Designing Monte Carlo Simulations in R

Luke W. Miratrix
James E. Pustejovsky

Table of contents

Welcome **9**

Welcome **9**
License 10
About the authors 10
Acknowledgements 12

I An Introductory Look **13**

1 Introduction **15**
1.1 Some of simulation's many uses 16
1.2 The perils of simulation as evidence 21
1.3 Simulating to learn 23
1.4 Why R? 24
1.5 Organization of the text 25

2 Programming Preliminaries **27**
2.1 Welcome to the tidyverse 27
2.2 Functions 28
2.3 \> (Pipe) dreams 35
2.4 Recipes versus Patterns 36
Exercises 36

3 An initial simulation **39**
3.1 Simulating a single scenario 41
3.2 A non-normal population distribution 43
3.3 Simulating across different scenarios 44
3.4 Extending the simulation design 47
Exercises 48

II Structure and Mechanics of a Simulation Study **51**

4 Structure of a simulation study **53**
4.1 General structure of a simulation 53

4.2 Tidy, modular simulations . . . 55
4.3 Skeleton of a simulation study . . . 56
Exercises . . . 62

5 Case Study: Heteroskedastic ANOVA and Welch **63**
5.1 The data-generating model . . . 66
5.2 The hypothesis testing procedures . . . 70
5.3 Running the simulation . . . 71
5.4 Summarizing test performance . . . 72
Exercises . . . 74

6 Data-generating processes **77**
6.1 Examples . . . 77
6.2 Components of a DGP . . . 79
6.3 A statistical model is a recipe for data generation . . . 82
6.4 Plot the artificial data . . . 84
6.5 Check the data-generating function . . . 86
6.6 Example: Simulating clustered data . . . 88
6.7 Sometimes a DGP is all you need . . . 96
6.8 More to explore . . . 101
Exercises . . . 102

7 Estimation procedures **111**
7.1 Writing estimation functions . . . 112
7.2 Including Multiple Data Analysis Procedures . . . 114
7.3 Validating an Estimation Function . . . 118
7.4 Handling errors, warnings, and other hiccups . . . 125
Exercises . . . 136

8 Running the Simulation Process **143**
8.1 Repeating oneself . . . 143
8.2 One run at a time . . . 144
8.3 Reparameterizing . . . 147
8.4 Bundling simulations with `simhelpers` . . . 148
8.5 Seeds and pseudo-random number generators . . . 150
8.6 Conclusions . . . 153
Exercises . . . 154

9 Performance Measures **157**
9.1 Measures for Point Estimators . . . 159
9.2 Measures for Variance Estimators . . . 166
9.3 Measures for Confidence Intervals . . . 171
9.4 Measures for Inferential Procedures (Hypothesis Tests) . . . 174
9.5 Relative or Absolute Measures? . . . 179
9.6 Estimands Not Represented By a Parameter . . . 182
9.7 Uncertainty in Performance Estimates . . . 185

9.8 Performance Calculations with the `simhelpers` Package 193
9.9 Concluding thoughts 197
Exercises 198

III Systematic Simulations 203

10 Simulating across multiple scenarios 205
10.1 Simulating across levels of a single factor 206
10.2 Simulating across multiple factors 213
10.3 Using pmap to run multifactor simulations 214
10.4 When to calculate performance metrics 217
10.5 Summary 222
Exercises 223

11 Designing multifactor simulations 227
11.1 Choosing factors and parameter combinations 228
11.2 Case Study: A multifactor evaluation of cluster RCT estimators 231
11.3 Further principles of design 238
11.4 Summary 240
Exercises 241

12 Exploring and presenting simulation results 243
12.1 Tabulation 244
12.2 Visualization 249
12.3 Modeling 261
12.4 Reporting 265

13 Building good visualizations 267
13.1 Subsetting and Many Small Multiples 268
13.2 Bundling 271
13.3 Aggregation 275
13.4 Comparing true SEs with standardization 279
13.5 The Bias-SE-RMSE plot 284
13.6 Assessing the quality of the estimated SEs 287
13.7 Assessing confidence intervals 291
13.8 Conclusions 294
Exercises 294

14 Special Topics on Reporting Simulation Results 297
14.1 Using regression to analyze simulation results 297
14.2 Using regression trees to find important factors 307
14.3 Analyzing results with few iterations per scenario 309
14.4 Multilevel Meta Regressions 313
14.5 What to do with warnings in simulations 316
14.6 Conclusions 318

15 Simulations as evidence **319**
15.1 Making simulations relevant . . . 320
15.2 Case study: Using copulas to generate calibrated data . . . 324
15.3 Simulation Sections in Methods Papers . . . 332
15.4 Concluding Thoughts . . . 339

IV Computational Considerations **341**

16 Organizing a simulation project **343**
16.1 Simulation project structure . . . 343
16.2 Well structured code files . . . 345
16.3 Principled directory structures . . . 348
16.4 Saving simulation results . . . 350
16.5 Summary . . . 354

17 Debugging and Testing **355**
17.1 Debugging with `print()` . . . 355
17.2 Debugging with `debug()` . . . 356
17.3 Debugging with `browser()` . . . 358
17.4 Protecting functions with assertions . . . 359
17.5 Unit testing . . . 361

18 Parallel Processing **367**
18.1 Parallel on your computer . . . 368
18.2 Parallel on a virtual machine . . . 369
18.3 Parallel on a cluster . . . 370
18.4 Concluding Thoughts . . . 378

V Complex Data Structures **379**

19 Using simulation as a power calculator **381**
19.1 Getting design parameters from pilot data . . . 382
19.2 The data generating process . . . 382
19.3 Running the simulation . . . 386
19.4 Evaluating power . . . 388
19.5 Using and exploring packages for power calculations . . . 393
19.6 Concluding Thoughts . . . 396

20 Simulation under the Potential Outcomes Framework **397**
20.1 Finite vs. Superpopulation inference . . . 398
20.2 Generating data under the potential outcome framework . . . 399
20.3 Finite sample performance measures . . . 402
20.4 Nested finite simulation procedure . . . 404
20.5 Superpopulation performance measures . . . 406

20.6 Calibrating simulations within the potential outcomes framework . . . 407
20.7 Concluding Thoughts . . . 409

References **411**

References **411**

Appendices **419**

A Coding Reference **419**
A.1 How to repeat yourself . . . 419
A.2 Default arguments for functions . . . 423
A.3 Profiling Code . . . 424
A.4 Optimizing code (and why you often shouldn't) . . . 429

Index **437**

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. Despite the ubiquity of simulation work, most quantitative researchers get little formal training in the design and implementation of Monte Carlo simulations. As a result, the simulation studies presented in academic journal articles are highly variable in terms of their high-level logic, scope, programming, and presentation. Although there has long been discussion of simulation design and implementation among statisticians and methodologists, the available guidance is scattered across many different disciplines, and much of it is focused on mechanics and computing tools, rather than on principles.

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! Our focus 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.

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!

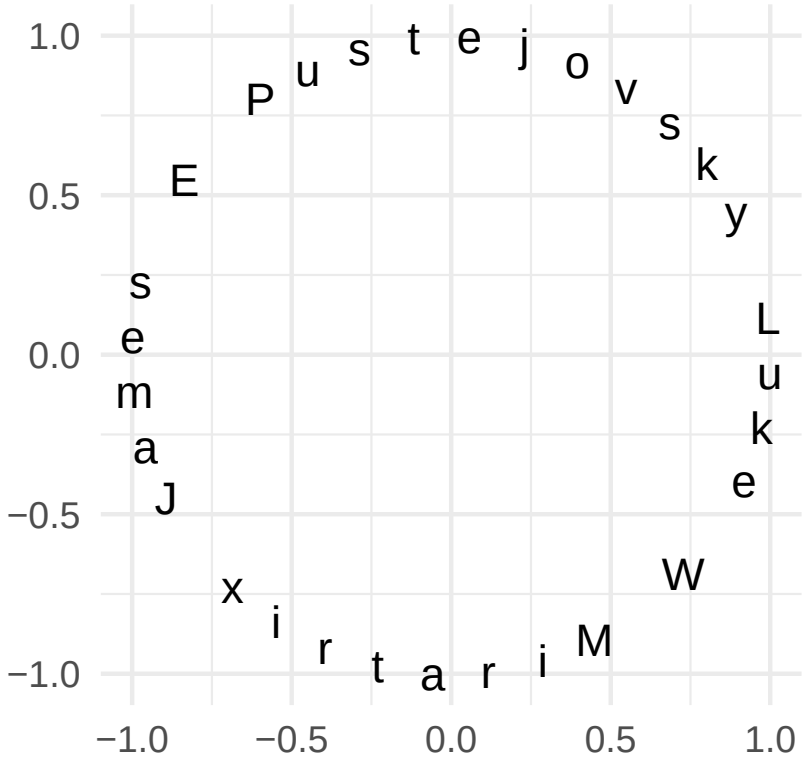
License

This book is licensed to you under Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

The code samples in this book are licensed under Creative Commons CC0 1.0 Universal (CC0 1.0), i.e. public domain.

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. Solutions for most of the exercises were completed and written up by Nathan Kessler, who also flagged several issues of clarity and accuracy in the text; thank you, Nathan, for your work on this book! Some parts of this book are based on memos or other writings generated for various purposes, some of which were written by others. This has been attributed throughout.

Part I

An Introductory Look

1

Introduction

Monte Carlo simulations are a tool for studying the behavior of random processes, such as the behavior of a statistical estimation procedure when applied to a sample of data. Within quantitatively oriented fields, researchers developing new statistical methods or evaluating the use of existing methods nearly always use Monte Carlo simulations as part of their research process. In the context of methodological development, researchers use simulations in a way analogous to how a chef would use their test kitchen to develop new recipes before putting them on the menu, how a manufacturer would use a materials testing laboratory to evaluate the safety and durability of a new consumer good before bringing it to market, or how an astronaut would prepare for a spacewalk by practicing the process in an underwater mock-up. Simulation studies provide a clean and controlled environment for testing out data analysis approaches before putting them to use with real empirical data.

More broadly, Monte Carlo studies are an essential tool in many different fields of science—climate science, engineering, and education research are three examples—and are used for a variety of different purposes. Simulations are used to model complex stochastic processes such as weather patterns (Jones et al., 2012; Robert and Casella, 2010); to generate parameter estimates from complex statistical models, as in Markov Chain Monte Carlo sampling (Gelman et al., 2013); and even to estimate uncertainty in statistical summaries, as in bootstrapping (Davison and Hinkley, 1997). In this book, we shall focus on using simulation for the development and validation of methods for data analysis, which are everyday concerns within the fields of statistics and quantitative methodology. However, we also believe that many of the general principles of simulation design and execution that we will discuss are broadly applicable to these other purposes, and we note connections as they occur.

At a very high level, Monte Carlo simulation provides a method for understanding the performance of a statistical model or data analysis method under conditions where the truth is known and can be controlled. The basic approach for doing so is as follows:

1. Create artificial data using random number generators based on a specific statistical model, or more generally, a *Data-Generating Process* (DGP).

2. Apply one or more data-analysis procedures to the artificial data. (These procedures might be something as simple as calculating a difference in sample means or fitting a regression model, or it might be an involved, multi-step procedure involving cleansing the data of apparent outliers, imputing missing values, applying a machine-learning algorithm, and carrying out further calculations on the predictions of the algorithm.)
3. Repeat Steps 1 and 2 many times.
4. Summarize the results across these repetitions in order to understand the general trends or patterns in how the method works.

Simulation is useful because one can control the data-generating process and therefore fully know the truth—something that is almost always uncertain when analyzing real, empirical data. Having full control of the data-generating process makes it possible to assess how well a procedure works by comparing the estimates produced by the data analysis procedure against this known truth. For instance, we can see if estimates from a statistical procedure are consistently too high or too low (i.e., whether an estimator is systematically biased). We can also compare multiple data analysis procedures by assessing the degree of error in each set of results to determine which procedure is generally more accurate when applied to the same collection of artificial datasets.

This basic process of simulation can be used to investigate an array of different questions that arise in statistics and quantitative methodology. To seed the field, we now give a high-level overview of some of the major use cases. We then discuss some of the major limitations and common pitfalls of simulation studies, which are important to keep in mind as we proceed.

1.1 Some of simulation's many uses

Monte Carlo simulations allow for rapid exploration of different data analysis procedures and, even more broadly, different approaches to designing studies and collecting measurements. Simulations are especially useful because they provide a means to answer questions that are difficult or impossible to answer by other means. Many statistical models and estimation methods *can* be analyzed mathematically, but only by using asymptotic approximations that describe how the methods work as sample size increases towards infinity. In contrast, simulation methods provide answers for specific, finite sample sizes. Thus, they allow researchers to study models and estimation methods where relevant mathematical formulas are not available, not easily applied, or not sufficiently accurate.

Circumstances where simulations are helpful—or even essential—occur in a

range of different situations within quantitative research. To set the stage for our subsequent presentation, consider the following areas where one might find need of simulation.

1.1.1 Comparing statistical approaches

One of the more common uses of Monte Carlo simulation is to compare alternative statistical approaches to analyzing the same type of data. In the academic literature on statistical methodology, authors frequently report simulation studies comparing a newly proposed method against more traditional approaches, to make a case for the utility of their method. As Dr. Little puts it in his article *Ten simple powerful ideas for the statistical scientist* (with "Embrace Well-Designed Simulation Experiments" being one of the ten!), "thoughtfully designed frequentist simulation experiments can cast useful light on the properties of a method" (Little, 2013).

A classic example of simulation for such evaluation is Brown and Forsythe (1974), who 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. A subsequent study by Mehrotra (1997) built on Brown and Forsythe's work, proposing a more refined method and using simulations to demonstrate that it is superior to the existing methods. We explore the Brown and Forsythe (1974) study in the case study of Chapter 5.

Comparative simulation can also have a practical application: In many situations, more than one data analysis approach is possible for addressing a given research question (or estimating a specified target parameter). Simulations comparing multiple approaches can be quite informative and can help to guide the design of an analytic plan (such as plans included in a pre-registered study protocol). For instance, researchers designing a multi-site randomized experiment might wonder whether they should use an analytic model that allows for variation in the site-specific impact estimates (Miratrix et al., 2021) or a simpler model that treats the impact as homogeneous across sites. What are the practical benefits and costs of using the more complex model? In the ideal case, simulations can identify best practices for how to approach analysis of a certain type of data and can surface trade-offs between competing approaches that occur in practice.

1.1.2 Assessing performance of complex pipelines

In practice, statistical methods are often used as part of a multi-step workflow. For instance, in a regression model, one might 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 heteroskedasticity-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

quite difficult to analyze mathematically, but it is straight-forward to simulate (see, for example, Long and Ervin, 2000). In particular, simulations are a straight-foward way to assess whether a proposed workflow is *valid*—that is, whether the conclusions from a pipeline are correct at a given level of certainty.

Beyond just evaluating the performance characteristics of a workflow, simulating a multi-step workflow can actually be used as a technique for *conducting* statistical inference with real data. Data analysis approaches such as randomization inference and bootstrapping involve repeatedly simulating data and putting it through an analytic pipeline, in order to assess the uncertainty of the original estimate based on real data. In bootstrapping, the variation in a point estimate across replications of the simulation is used as 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. See the first few chapters of Davison and Hinkley (1997) or Efron (2000) for further discussion of bootstrapping, and see Good (2013) or Lehmann et al. (1975) for more on permutation inference.

1.1.3 Assessing performance under misspecification

Many statistical estimation procedures are known to perform well when the assumptions they entail are correct. However, data analysts must also be concerned with the *robustness* of estimation procedures—that is, their performance when one or more of the assumptions is violated to some degree. For example, in a multilevel model, how important is the assumption that the random effects are normally distributed? What about normality of the individual-level error terms? What about homoskedasticity of the individual-level error terms? Quantitative researchers routinely contend with such questions when analyzing empirical data, and simulation can provide some answers.

Similar concerns arise for researchers considering the trade-offs between methods that make relatively stringent assumptions versus methods that are more flexible or adaptive. When the true data-generating process meets stringent assumptions (e.g., a treatment effect that is constant across the population of participants), what are the potential gain of exploiting such structure in the estimation process? Conversely, what are the costs (in terms of computation time or precision) of using more flexible methods that do not impose strong assumptions? A researcher designing an analytic plan would want to be well-informed of such trade-offs and, ideally, would want to situate their understanding in the context of the empirical phenomena that they study. 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 for data of a given, finite size is more difficult to quantify. Although mathematical theory can inform us about "asymptopia," empirical researchers live in a world of finite sample sizes, where it can be difficult to gauge if one's real data is large enough that the asymptotic approximations apply. For example, this is of particular concern with hierarchical data structures that include only 20 to 40 clusters—a common circumstance in many randomized field trials in education research. Simulation is a tractable approach for assessing the small-sample performance of such estimation methods or for determining minimum required sample sizes for adequate performance.

One example of a simulation investigating questions of finite-sample behavior comes from Long and Ervin (2000), whose evaluated the performance of heteroskedasticity-robust standard errors (HRSE) in linear regression models. Asymptotic analysis indicates that HRSEs work well (in the sense of providing correct assessments of uncertainty) in sufficiently large samples (White, 1980), but what about in realistic contexts where small samples occur? Long and Ervin (2000) use extensive simulations to investigate the properties of different versions of HRSEs 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 found in typical social science applications. Via simulation, they provided compelling evidence about a problem without having to wade into a technical (and potentially inaccessible) mathematical analysis of the problem.

1.1.5 Conducting Power Analyses

During the process of proposing, seeking funding for, and planning an empirical research study, researchers need to justify the design of the study, including the size of the sample that they aim to collect. Part of such justifications may involve a *power analysis*, or an approximation of the probability that the study will show a statistically significant effect, given assumptions about the magnitude of true effects and other aspects of the data-generating process. Researchers may also wish to compare the power of different possible designs in order to inform decisions about how to carry out the proposed study given a set of monetary and temporal constraints.

Many guidelines and tools are available for conducting power analysis for various research designs, including software such as PowerUp! (Dong and Maynard, 2013), the Generalizer (Tipton, 2013), G*Power (Faul et al., 2009), and PUMP (Hunter et al., 2024). These tools use analytic formulas for power,

which are often derived using approximations and simplifying assumptions about a planned design. Simulation provides a very general-purpose alternative for power calculations, which can avoid such approximations and simplifications. By repeatedly simulating data based on a hypothetical process and then analyzing data following a specific protocol, one can *computationally* approximate the power to detect an effect of a specified size.

Using simulation instead of analytic formulas allows for power analyses that are more nuanced and more tailored to the researcher's circumstance than what can be obtained from available software. For example, simulation can be useful for the following:

- When estimating power in multi-site, block- or cluster-randomized trials, the formulas implemented in available software assume that sites are of equal size and that outcome distributions are unrelated to the size of each site. 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 build in assumptions about anticipated rates of attrition in cluster-randomized trials, under the assumption that attrition is completely at random and unrelated to anything. However, researchers might anticipate that, in practice, attrition will be related to baseline characteristics. Simulation can be used to assess how this might 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 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 useful to generate realistic data structures and conduct power analysis via simulation.

1.1.6 Simulating processess

Yet another common use for Monte Carlo simulation is as a way to emulate a complex process as a means to better understand it or to evaluate the consequences of modifying it. 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.

Another example of process simulation arises in education research. Some large school districts such as New York City have centralized lotteries for school

assignment, which entail having families rank schools by order of preference. The central office then assigns students to schools via a lottery procedure where each student gets a lottery number that breaks ties when there are too many students desiring to go to certain schools. Students' school assignments are therefore based in part on random chance, but the the process is quite complex: each student has some probability of assignment to each school on their list, but the probabilities depend on their choices and the choices of other students.

The school lottery process creates a natural experiment, based on which researchers can estimate the causal impact of being assigned to one school vs. another. A defensible analysis of the process requires knowing the probabilities of school assignment. Abdulkadiroğlu et al. (2017) conducted such an evaluation using the school lottery process in New York City. They calculated school assignment probabilities via simulation, by running the school lottery over and over, changing only students' lottery numbers, and recording students' school assignments in each repetition of the process. Simulating the lottery process a large number of times provided precise estimates of each students' assignment probabilities, based on which Abdulkadiroğlu et al. (2017) were able to estimate causal impacts of school assignment.

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) simulated the process of firing teachers depending on their estimated value-added scores. Based on their simulations, which model firing different proportions of teachers, they suggest that firing substantial portions of the teacher workforce annually would substantially improve student test scores. Their work offers a clear illustration of how simulations can be used to examine the potential consequences of various policy decisions, assuming the underlying assumptions hold true. This example also brings home a core concern of simulation: we only learn about the world we are simulating, and the relevance of simulation evidence to the real world is by no means guaranteed.

1.2 The perils of simulation as evidence

Simulation has the potential to be a powerful tool for investigating quantitative methods. However, evidence from simulation studies is also fundamentally limited in certain ways, and thus very susceptible to critique. The core advantage of simulation studies is that they allow for evaluation of data analysis methods under *specific and exact conditions*, avoiding the need for approximation. The core limitation of simulations stems from this same property: they provide information about the performance of data analysis methods under specified conditions, but provide no guarantee that patterns of performance

hold in general. One can partially address questions of generalization by examining a wide range of conditions, looking to see whether a pattern holds consistently or changes depending on features of the data-generating process. Even this strategy has limitations, though. Except for very simple processes, we can seldom consider every possible set of conditions.

As we will see in later chapters, the design of a simulation study typically entails making choices over very large spaces of possibility. This flexibility leaves lots of room for discretion and judgement, and even for personal or professional biases (Boulesteix et al., 2020). Due to this flexibility, simulation findings are held in great skepticism by many. The following motto summarizes the skeptic's concern:

> Simulations are doomed to succeed.

As this motto captures, 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." If our goal is to show something that we already believe is correct (e.g., that our fancy new estimation procedure is better than existing methods), we could probably find a way to align our simulation with our intuition.[1]

Critiques of simulation studies often revolve around the *realism*, *relevance*, or *generality* of the data generating process. Are the simulated data realistic, in the sense that they follow similar patterns to what one would see in real empirical data? Are the explored aspects of the simulation relevant to what we would expect to find in practice? Was the simulation systematic in exploring a wide variety of scenarios, so that general conclusions are warranted?

We see at least three principles for addressing such questions in one's own work. Perhaps most fundamental is to be transparent in one's methods and reasoning: explicitly state what was done, and provide code so that others can reproduce one's results or tweak them to test variations of the data-generating process or alternative analysis strategies. Another important component of a robust argument is to systematically vary the conditions under examination. This is facilitated by writing code in a way to make it easy to simulate across a range of different data-generating scenarios. Once that is in place, one can systematically explore myriad scenarios and report all of the results. An aspiration of the simulation architect should be to explore the boundary conditions that separate where preferred methods work and where they break or fail. Finally, one can draw on relevant statistical theory to support the design of

[1] A comment from James: I recall attending seminars in the statistics department during graduate school, where guest speakers usually presented both some theory and some simulation results. A few years into my graduate studies, I realized that the simulation part of the presentation could nearly always be replaced with a single slide that said "we did some simulations and showed that our new method works better than old methods under conditions that we have cleverly selected to be favorable for our approach." I hope that my own work is not as boring or predictable as my memory of these seminars.

a simulation and interpretation of its results. Mathematical analysis might indicate that some features of a data-generating process will have a strong influence on the performance of a method, while other features will not matter at all when sample sizes are sufficiently large. Well designed simulations will examine conditions that are motivated by or complement what is known based on existing statistical theory.

In addition to these principles, methodologists have proposed broader changes in practice to counter the potential for bias in methodological simulation studies. Morris et al. (2019) introduced a formal framework, called ADEMP, to guide the reporting of methodological simulations. Boulesteix et al. (2013) argued for greater use of neutral comparison studies, in which the performance of alternative statistical methods are compared under a range of relevant conditions by methodological researchers who do not have vested interests in any specific method (see also Boulesteix et al., 2017). Further, Siepe et al. (2024) argue for more routine pre-registration of methodological simulations to bring greater transparency and reduce the possibility of bias arising from flexibility in their design.

1.3 Simulating to learn

Most of the examples of Monte Carlo simulation that we have mentioned thus far are drawn from formal methodological research, published in methodologically focused research journals. If you do not identify as a methodologist, you may be wondering whether there is any benefit to learning how to do simulations—what's the point, if you are never going to conduct methodological research studies or use simulation to aid in planning an empirical study? However, we believe that simulation is an incredibly useful tool—and well worth learning, even outside the context of formal methodological research—for at least two reasons.

First, in order to do any sort of quantitative data analysis, you will need to make decisions about what methods to use. Across fields, existing guidance about data analysis practice is almost certainly informed by simulation research of some form, whether well-designed and thorough or haphazard and poorly reasoned. Consequently, having a high-level understanding of the logic and limitations of simulation will help you to be a critical consumer of methods research, even if you do not intend to conduct methods research of your own.

Second, we believe conducting simulations deepens one's understanding of the logic of statistical modeling and statistical inference. Learning a new statistical model (such as generalized linear mixed models) or analytic technique

(such as multiple imputation by chained equations) requires taking in *a lot* of detailed information, from the assumptions of the model to the interpretation of parameter estimates to the best practices for estimation and what to do if some part of the process goes off. To thoroughly digest all these details, we have found it invaluable to *simulate* data based on the model under consideration. This usually requires translating mathematical notation into computer code, an exercise which makes the components of the model more tangible than just a jumble of Greek letters. The simulated data is then available for inspection, summarizing, graphing, and further calculation, all of which can aid comprehension and interpretation. Moreover, the process of simulating yields a dataset which can then be used to practice implementing the analysis procedure and interpreting the results. We have found that building a habit of simulating is a highly effective way to learn new models and methods, worthwhile even if one has no intention of carrying out methodological research. We might even go so far as to argue that *whatever you might think, you don't really understand a statistical model until you've done a simulation of it.*

1.4 Why R?

This book aims not only to introduce the conceptual principles of Monte Carlo simulation, but also to provide a practical guide to actually *conducting* simulation studies (whether for personal learning purposes or for formal methodological research). And conducting simulations requires writing computer code (sometimes, lots of code!). The computational principles and practices that we will describe are very general, not specific to any particular programming language, but for purposes of demonstrating, presenting examples, and practicing the process of developing a simulation, it helps to be specific. To that end, we will be using R, a popular programming language that is widely used among statisticians, quantitative methodologists, and data scientists. Our presentation will assume that readers are comfortable with writing R scripts to carry out tasks such as cleaning variables, summarizing data, creating data-based graphics, and running regression models (or more generally, estimating statistical models).

We have chosen to focus on R (rather than some other programming language) because both of us are intimately familiar with R and use it extensively in our day-to-day work. Simply put, it is much easier to write in your native language than in one in which you are less fluent. But beyond our own habits and preferences, there are several more principled reasons for using R.

R is free and open source software, which can be run under many different operating systems (Windows, Mac, Linux, etc.). This is advantageous not only because of the cost, but also because it means that anyone with a computer—

anywhere in the world—can access the software and could, if they wanted, re-run our provided code for themselves. This makes R a good choice for practicing transparent and open science processes.

There is a very large, active, and diverse community of people who use, teach, and develop R. It is used widely for applied data analysis and statistical work in such fields as education, psychology, economics, epidemiology, public health, and political science, and is widely taught in quantitative methods and applied statistics courses. Integral to the appeal of R is that it includes tens of thousands of contributed packages, which extend the core functionality of the language in myriad ways. New statistical techniques are often quickly available in R, or can be accessed through R interfaces. Increasingly, R can also be used to interface with other languages and platforms, such as running Python code via the `reticulate` package, running Stan programs for Bayesian modeling via `RStan`, or calling the h2o machine learning library using the `h2o` package (Fryda et al., 2014). The huge variety of statistical tools available in R makes it a fascinating place to learn and practice.

R does have a persistent reputation as being a challenging and difficult language to use. This reputation might be partly attributable to its early roots, having been developed by highly technical statisticians who did not necessarily prioritize accessibility, legibility of code, or ease of use. However, as the R community has grown, the availability of introductory documentation and learning materials has improved drastically, so that it is now much easier to access pedagogical materials and find help.

R's reputation also probably partly stems from being a decentralized, open source project with many, many contributors. Contributed R packages vary hugely in quality and depth of development; there are some amazingly powerful tools available but also much that is half-baked, poorly executed, or flat out wrong. Because there is no central oversight or quality control, the onus is on the user to critically evaluate the packages that they use. For newer users especially, we recommend focusing on more established and widely used packages, seeking input and feedback from more knowledgeable users, and taking time to validate functionality against other packages or software when possible.

A final contributor to R's intimidating reputation might be its extreme flexibility. As both a statistical analysis package and a fully functional programming language, R can do many things that other software packages cannot, but this also means that there are often many different ways to accomplish the same task. In light of this situation, it is good to keep in mind that knowing a single way to do something is usually adequate—there is no need to learn six different words for hello, when one is enough to start a conversation.

On balance, we think that the many strengths of R make it worthwhile to learn and continue exploring. For simulation, in particular, R's facility to easily write

functions (bundles of commands that you can easily call in different manners), to work with multiple datasets in play at the same time, and to leverage the vast array of other people's work all make it a very attractive language.

1.5 Organization of the text

We think of simulation studies as falling on a spectrum of formality. On the least formal end, we may use simulations to learn a statistical model, investigating questions purely to satisfy our own curiosity. On the most formal end, we may conduct carefully designed, pre-registered simulations that compare the performance of competing statistical methods across an array of data-generating conditions designed to inform practice in a particular research area.

Our central goal is to help you learn to work anywhere along this spectrum, from the most casual to the most principled. To support that goal, the book is designed with a spiral structure, where we first present simpler and less formal examples that illustrate high-level principles, then revisit the component pieces, dissecting and exploring them in greater depth. We defer discussion of concepts that are relevant only to more formal use-cases (such as the ADEMP framework of Morris et al. (2019)) until later chapters. We have also included many case studies throughout the book; these are designed to make the topics under discussion tangible, and are also designed to provide chunks of code that you emulate or even directly copy and use for your own purposes.

We divided the book into several parts. In Part I (which you are reading now), we lay out our case for learning simulation, introduces some guidance on programming, and presents an initial, very simple simulation to set the stage for later discussion of design principles. Part II lays out the core components (generating artificial data, applying analytic procedures, executing the simulations, and analyzing the simulation results) for simulating a single scenario. It then presents some more involved case studies that illustrate the principles of modular, tidy simulation design. Part III moves to multifactor simulations, meaning simulations that look at more than one scenario or context. Multifactor simulation is central to the design of more formal simulation studies because it is only by evaluating or comparing estimation procedures across multiple scenarios that we can begin to understand their general properties.

The book closes with two final parts. Part IV covers some technical challenges that are commonly encountered when programming simulations, including reproducibility, parallel computing, and error handling. Part V covers three extensions to specialized forms of simulation: simulating to calculate power, simulating within a potential outcomes framework for causal inference, and the parametric bootstrap. The specialized applications underscore how sim-

ulation can be used to answer a wide variety of questions across a range of contexts, thus connecting back to the broader purposes discussed above. The book also includes appendices with some further guidance on writing R code and pointers to further resources.

2

Programming Preliminaries

In this chapter, we introduce some essential programming concepts that may be less familiar to readers, but which are central to how we approach writing code for simulation studies. We also explain some of the rationale and reasoning behind how we present example code throughout the book.

2.1 Welcome to the tidyverse

Layered on top of R are a collection of contributed packages that make data wrangling and management much, much easier. This collection is called `tidyverse` and it includes popular packages such as `dplyr`, `tidyr`, and `ggplot2`. We use methods from the "tidyverse" throughout the book because it facilitates writing clean, concise code. In particular, we make heavy use of the `dplyr` package for group-wise data manipulation, the `purrr` package for functional programming, and the `ggplot2` package for statistical graphics. The 1st edition or 2nd edition of the free online textbook *R for Data Science* provide an excellent, thorough introduction to these packages, along with much more background on the tidyverse. We will cite portions of this text throughout the book.

Loading the tidyverse packages is straightforward:

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

(The second line is to turn off some of the persistent warnings generated by the `dplyr` function `summarize()`.) These lines of code appear in the header of nearly every script we use.

2.2 Functions

If you are comfortable using R for data analysis tasks, you will be familiar with many of R's functions. R has function to do things like calculate a summary statistic from a list of numbers (e.g., `mean()`, `median()`, or `sd()`), calculate linear regression coefficient estimates from a dataset (`lm()`), or count the number of rows in a dataset (`nrow()`). In the abstract, a function is a little machine for transforming ingredients into outputs, like a microwave (put a bag of kernels in and it will return hot, crunchy popcorn), a cheese shredder (put a block of mozzarella in and it transforms it into topping for your pizza), or a washing machine (put in dirty clothes and detergent and it will return clean but damp clothes). A function takes in pieces of information specified by the user (the inputs), follows a set of instructions for transforming or summarizing those inputs, and then returns the result of the calculations (the outputs).

A function can do nearly anything as long as the calculation can be expressed in code—it can even produce output that is random. For example, the `rnorm()` function takes as input a number `n` and returns that many random numbers, drawn from a standard normal distribution:

```
rnorm(3)
```

```
[1] 0.691978592 0.004965465 0.125490190
```

Each time the function is called, it returns a different set of numbers:

```
rnorm(3)
```

```
[1] -0.90259046 -0.06195651 -0.26001624
```

The `rnorm()` function also has further input arguments that let the user specify the mean and standard deviation of the distribution from which numbers are drawn:

```
rnorm(3, mean = 10, sd = 0.5)
```

```
[1] 10.15607 10.22331 10.17459
```

In writing code for simulations, we will make extensive use of this particular function and other functions that produce sequences of random numbers. We will have more to say about random number generation later.

2.2.1 Rolling your own

In R, you can create your own function by specifying the pieces of input information, the steps to follow in transforming the inputs, and the result to return as output. Learning to write your own functions to carry out calculations is

an immensely useful skill that will greatly enhance your ability to accomplish a range of tasks. Writing custom functions is also central to our approach to coding Monte Carlo simulations, and so we highlight some of the key considerations here. Chapter 19 of R for Data Science (1st edition) provides an in-depth discussion of how to write your own functions.

Here is an example of a custom function called `one_pval()`:

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

The first line specifies that we are creating a function that takes inputs `N`, `mn`, and `sd`. These are called the *parameters*, *inputs*, or *arguments* of the function. The remaining lines inside the curly brackets are called the *body* of the function. These lines specify the instructions to follow in transforming the inputs into an output:

1. Generate a random sample of `N` observations from a normal distribution with mean `mn` and store the result in `vals`.
2. Use the built-in function `t.test()` to compute a one-sample t-test for the null hypothesis that the population mean is zero, then store the result in `tt`.
3. Extract the p-value from the t-test store the result in `pvalue`.
4. Return `pvalue` as output.

Having created the function, we can then use it with any inputs that we like:

```
one_pval( 100, 5, 1 )
```

```
[1] 1.357973e-66
```

```
one_pval( 10, 0.3, 1 )
```

```
[1] 0.2619148
```

```
one_pval( 10, 0.3, 0.2 )
```

```
[1] 0.0002720652
```

In each case, the output of the function is a p-value from a simulated sample of data. The function produces a different answer each time because its instructions involve generating random numbers each time it is called. In essence, our custom function is just a short-cut for carrying out its instructions. Writing it saves us from having to repeatedly write or copy-paste the lines of code inside its body.

2.2.2 A dangerous function

Writing custom functions will prove to be crucial for effectively implementing Monte Carlo simulations. However, designing custom functions does take practice to master. It also requires a degree of care above and beyond what is needed just to use R's built-in functions.

One of the common mistakes encountered in writing custom functions is to let the function depend on information that is not part of the input arguments. For example, consider the following script, which includes a nonsensical custom function called `funky()`:

```
secret <- 3

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

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

  return(funky_output)
}

funky(3, 2, 5)
```

```
[1] 5.5
```

`funky` takes inputs `input1`, `input2`, and `input3`, but its instructions also depend on the quantity `secret`. What happens if we change the value of `secret`?

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

```
[1] 102.5
```

Even though we give it the same arguments as previously, the output of the function is different. This sort of behavior is confusing. Unless the function involves generating random numbers, we would generally expect it to return the exact same output if we give it the same inputs. Even worse, we get a rather cryptic error if the value of `secret` is not compatible with what the function expects:

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

```
Error in `input3 * ratio + secret`:
! non-numeric argument to binary operator
```

If we are not careful, we will end up with very confusing code that can very

easily lead to unintended results and errors.

To avoid this issue, it is important for functions to only use information that is explicitly provided to it through its arguments. This is the principle of *isolating the inputs*. If the result of a function is supposed to depend on a quantity, then we should include that quantity among the input arguments. We can fix our example function by including `secret` as an argument:

```
secret <- 3

funkier <- function(input1, input2, input3, secret) {

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

  return(funky_output)
}

funkier(3, 2, 5, 3)
```

```
[1] 5.5
```

Now the output of the function is always the same, regardless of the value of other objects in R:

```
secret <- 100
funkier(3, 2, 5, 3)
```

```
[1] 5.5
```

The *input parameter* `secret` holds sway here, even though there is also an object with the same name.[1] If we want to try 100, we have to do so *explicitly*:

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

```
[1] 102.5
```

When writing your own functions, it may not be obvious that your function depends on external quantities and does not isolate the inputs. In our experience, one of the best ways to detect this issue is to clear the R environment and start from a fresh palette, run the code to create the function, and call the function (perhaps more than once) to ensure that it works as expected. Here is an illustration of what happens when we follow this process with our problematic custom function:

[1] To learn more about how R determines which values to use when executing a function, see Section 6.4 of Advance R.

```
# clear environment
rm(list=ls())

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

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

  return(funky_output)
}

# test the function
funky(3, 2, 5)
```

```
Error in `funky()`:
! object 'secret' not found
```

We get an error because the external quantity is not available. Here is the same process using our corrected function:

```
# clear environment
rm(list=ls())

# create function
funkier <- function(input1, input2, input3, secret) {

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

  return(funky_output)
}


# test the function
funkier(3, 2, 5, secret = 3)
```

```
[1] 5.5
```

2.2.3 Using Named Arguments

When calling a function (whether it is a built-in function or a custom function that you developed), you can specify which values correspond to which arguments using argument names. Using argument names greatly enhances

the readability and flexibility of function calls. When you specify inputs by name, R matches the values to the arguments based on the names rather than the order in which they appear. This feature is particularly useful in complex functions with many optional arguments.

For example, consider the function `one_pval()` from Section 2.2.1:

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

You could call `one_pval()` with *named* arguments in any order:

```
result <- one_pval(sd = 0.5, mn = 2, N = 500)
```

In this call, R knows which value to assign to each argument, so we could list the arguments however we like.

Without naming, we would have to specify the arguments in the exact order that they appear in the function definition:

```
result <- one_pval( 500, 2, 0.5 )
```

Without the argument names, this line of code is harder to follow—you have to know more about the design of the function to understand how it is being used in this instance.

Getting in the habit of using named arguments will help you avoid errors. If you pass arguments without naming, and in the wrong order, you can end up with very strange results that are hard to diagnose. Even if you get it right, if someone later changes the function (say by adding a new argument in the middle of the list), your code will suddenly break with no explanation.

2.2.4 Argument Defaults

Default arguments allow you to specify typical values for parameters that a user might not need to change every time they use the function. This can make the function easier to use and less error-prone because the defaults ensure that the function behaves sensibly even when some arguments are not explicitly provided. For example, let us revise the `one_pval()` function from above to use default arguments:

```
one_pval <- function( N = 10, mn = 0, sd = 1 ) {
  vals <- rnorm( N, mean = mn, sd = sd )
  tt <- t.test( vals )
  pvalue <- tt$p.value
```

```
  return(pvalue)
}
```

Now our function has a default for `N` of 10, for `mn` of 0, and for `s` of 1. This means a user can run the function simply by calling `one_pval()` without any inputs. Doing so will generate a p-value from a sample of 10 observations with a mean of zero and a standard deviation of 1.

Once we have our function with defaults, we can call the function while specifying only the inputs that differ from the default values:

```
bigger_sample <- one_pval( N = 50 )
```

The function will use its default values for `mn` and `s`. Using defaults lets the user call the function more succinctly.

Later chapters will have much more to say about the process of writing custom functions, as well as many further illustrations and examples.

2.2.5 Function skeletons

In discussing how to write functions for simulations, we will often present *function skeletons*. By a skeleton, we mean code that creates a function with a specific set of input arguments, but where the body is left partially or fully unspecified. Here is a cursory example of a function skeleton:

```
run_simulation <- function( N, J, mu, sigma, tau ) {
  # simulate data
  # apply estimation procedure
  # repeat
  # summarize results
  return(results)
}
```

In subsequent chapters, we will use function skeletons to outline the organization of code for simulation studies. The skeleton headers make clear what the inputs to the function need to be. Sometimes, we will leave comments in the body of the skeleton to sketch out the general flow of calculations that need to happen. Depending on the details of the simulation, the specifics of these steps might be quite different, but the general structure will often be quite consistent. Finally, the last line of the skeleton indicates the value that should be returned as output of the function. Thus, skeletons are kind of like Mad Libs, but with R code instead of parts of speech.

2.3 \> (Pipe) dreams

Many of the functions from `tidyverse` packages are designed to make it easy to use them in sequence via the `|>` symbol, or *pipe*.[2] The pipe allows us to *compose* several functions, meaning to write a chain of several functions as a sequence, where the result of each function becomes the first input to the next function. In code written with the pipe, the order of function calls follows like a story book or cake recipe, making it easier to see what is happening at each step in the sequence.

Consider the hypothetical functions `f()`, `g()`, and `h()`, and suppose we want to do a calculation that involves composing all three functions. One way to write this calculation is

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

We have to store the result of each intermediate step in an object, and it takes a careful read of the code to see that we are using `res1` as input to `g()` and `res2` as input to `h()`.

Alternately, we could try to write all the calculations as one line:

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

This is a mess. It takes very careful parsing to see that the `b` argument is called as part of `g()` and the `c` argument is part of h()‘, and the order in which the functions appear is not the same as the order in which they are calculated.

With the pipe we can write the same calculation as

```
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()
```

This addresses the all the issues with our previous attempts: the order in which the functions appear is the same as the order in which they are executed; the additional arguments are clearly associated with the relevant functions; and

[2] The pipe is a relatively recent addition to R's basic syntax. Prior to its inclusion in base R, the `magrittr` package provided—and still provides—a pipe symbol `%>%` that works similarly but has some additional syntactic nuances. We use base R's `|>` because it is always available, even without loading any additional packages. To learn more about the nuanced `%>%` pipe and similar operators, see the magrittr package.

there is only a single object holding the results of the calculations. Pipes are a very nice technique for writing clear code that is easy for others to follow.[3]

2.4 Recipes versus Patterns

As we will elaborate in subsequent chapters, we follow a modular approach to writing simulations, in which each component of the simulation is represented by its own custom function or its own object in R. This modular approach leads to code that always has the same broad structure and where the process of implementing the simulation follows a set sequence of steps. We start by coding a data-generating process, then write one or more data-analysis methods, then determine how to evaluate the performance of the methods, and finally implement an experimental design to examine the performance of the methods across multiple scenarios. Over the next several chapters, we will walk through this process several times.

Although we always follow the same broad process, the case studies that we will present are *not* intended as a cookbook that must be rigidly followed. In our experience, the specific features of a data-generating model, estimator, or research question sometimes require tweaking the template or switching up how we implement some aspect of the simulation. And sometimes, it might just be a question of style or preference. Because of this, we have purposely constructed the examples presented throughout the book to use different variations of our central theme rather than always following the exact same style and structure. We hope that presenting these variants and adaptations will both expand your sense of what is possible and also help you to recognize the core design principles—in other words, to distinguish the forest from the trees. Of course, we would welcome and encourage you to take any of the code verbatim, tweak and adapt it for your own purposes, and use it however you see fit. Adapting a good example is usually much easier than starting from a blank screen.

Exercises

Exercise 2.1. Revise the `one_pval()` function to use `|>` instead of storing the simulated sample in `vals`.

[3] Chapter 3.4 of R for Data Science (2nd edition) provides more discussion and examples of how to use `|>`. Chapter 18 of R for Data Science (1st edition) provides more discussion and examples of how to use `magrittr`'s `%>%`.

Exercise 2.2. Modify the `one_pval()` function to return a `tibble()` that includes separate columns for the test statistic (called `tt$statistic`), the p-value, and the lower and upper end-points of the confidence interval (called `tt$conf.int`).

Exercise 2.3. Modify the `one_pval()` function so that the sample of data is generated from a non-central t distribution by substituting R's `rt()` function in place of `rnorm()`. Make sure to modify the arguments (and argument names) of `one_pval()` to allow the user to specify the non-centrality and degrees of freedom parameters of the non-central t distribution.

Exercise 2.4. The non-central t distribution is usually parameterized in terms of non-centrality parameter δ and degrees of freedom ν, and these parameters determine the mean and spread of the distribution. Specifically, the mean of the non-central t distribution is

$$\mathrm{E}(T) = \delta \times \sqrt{\frac{\nu}{2}} \times \frac{\Gamma((\nu-1)/2)}{\Gamma(\nu/2)},$$

where $\Gamma()$ is the gamma function (called `gamma()` in R). Create a version of the `one_pval()` function that generates data based on a non-central t distribution, but where the input arguments are `mn` for the mean and `df` for the degrees of freedom. Here is a function skeleton to get started:

```
one_pval <- function( N = 10, mn = 5, df = 4) {

  # generate data from non-central t distribution
  # vals <-

  # calculate one-sample t-test
  tt <- t.test( vals )

  # compile results into a tibble and return
  # res <-

  return(res)
}
```

Exercise 2.5. Modify `one_pval()` to allow the user to specify a hypothesized value for the population mean, to use when calculating the one-sample t-test results.

3

An initial simulation

To begin learning about simulation, a good starting place is to examine a small, concrete example. This example illustrates how simulation involves replicating the data-generating and data-analysis processes, followed by aggregating the results across replications. Our little example encapsulates the bulk of our approach to Monte Carlo simulation, touching on all the main components involved. In subsequent chapters we will look at each of these components in greater detail. But first, let us look at a simulation of a very simple statistical problem.

The one-sample t-test is one of the most basic methods in the statistics literature. It tests a null hypothesis that a population mean of some variable is equal to a specific value by comparing the mean of a sample of data to the hypothesized value. If the sample average is discrepant (very different) from the null value, relative to how uncertain we are about our estimate, then the hypothesis is rejected. The test can also be used to generate a confidence interval for the population mean. If the sample consists of independent observations and the variable is normally distributed in the population, then the confidence interval will have exact coverage, in the sense that 95% intervals will include the population mean in 95 out of 100 tries. But what if the population variable is not normally distributed?

To find out, let us look at the coverage of the t-test's 95% confidence interval for the population mean when the method's normality assumption is violated. *Coverage* is the chance of a confidence interval capturing the true parameter value. To examine coverage, we will simulate many samples from a non-normal population with a specified mean, calculate a confidence interval based on each sample, and see how many of the confidence intervals cover the known true population mean.

Before getting to the simulation, let's look at the data-analysis procedure we will be investigating. Here is the result of conducting a t-test on some fake data, generated from a normal distribution:

```
# make fake data
dat <- rnorm( n = 10, mean = 4, sd = 2 )

# conduct the test
```

```
tt <- t.test( dat )
tt
```

```
    One Sample t-test

data:  dat
t = 6.0878, df = 9, p-value = 0.0001819
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 2.164025 4.723248
sample estimates:
mean of x
 3.443636
```

```
# examine the confidence interval
tt$conf.int
```

```
[1] 2.164025 4.723248
attr(,"conf.level")
[1] 0.95
```

We generated data with a true (population) mean of 4. Did we capture it? To check, we can use the `findInterval()` function, which checks to see where the first number lies relative to the range given in the second argument. Here is an illustration of the syntax:

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

```
[1] 0
```

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

```
[1] 1
```

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

```
[1] 2
```

The `findInterval()` returns a `1` if the value of the first argument value is in the interval specified in the second argument.

We can apply it to check whether our estimated CI covers the true population mean:

```
findInterval( 4, tt$conf.int )
```

```
[1] 1
```

In this instance, `findInterval()` is equal to `1`, which means our CI captured the true population mean of 4.

Here is the full code for simulating data, computing the data-analysis procedure, and evaluating the result:

```
# make fake data
dat <- rnorm( n = 10, mean = 4, sd = 2 )

# conduct the test
tt <- t.test( dat )

# evaluate the results
findInterval( 4, tt$conf.int ) == 1
```

```
[1] TRUE
```

The above code captures the basic form of a single simulation trial: make the data, analyze the data, decide how well we did. The code also illustrates a good way to figure out the details of a simulation: start by figuring out what a single iteration of the simulation might look like. Starting by mucking about this way also allows us to test and develop our code in an interactive, exploratory fashion. For instance, we can play with `findInterval()` to figure out how to use it to determine whether our confidence interval captured the truth. Once we have arrived at working code for a single iteration, we are in a good position to start writing functions to implement the actual simulation. For now, we have generated data from a normal distribution; we will later revise the code to generate data from a non-normal population distribution.

3.1 Simulating a single scenario

We can estimate the coverage of the confidence interval by repeating the above data-generating and data-analysis processes many, many times. R's `replicate()` function is a handy way to repeatedly call a line of code. Its first input argument is `n`, the number of times to repeat the calculation, followed by `expr`, which is one or more lines of code to be called. We can use `replicate` to repeat our simulation process 1000 times in a row, each time generating a new sample of 10 observations from a normal distribution with mean of 4 and a standard deviation of 2. For each replication, we store the result of using `findInterval()` to check whether the confidence interval includes the population mean of 4.

```
rps <- replicate( 1000, {
  dat <- rnorm( n = 10, mean = 4, sd = 2 )
  tt <- t.test( dat )
  findInterval( 4, tt$conf.int )
})
```

```
head(rps, 20)
```

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

To see how well we did, we can look at a table of the results stored in `rps` and calculate the proportion of replications that the interval covered the population mean:

```
table( rps )
```

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

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

```
[1] 0.957
```

We got about 95% coverage, which is good news. In 27 out of the 1000 replications, the interval was too high (so the population mean was below the interval) and in 16 out of the 1000 replications, the interval was too low (so the population mean was above the interval).

It is important to recognize that this set of simulations results, and our coverage rate of 95.7%, itself has some uncertainty in it. Because we only repeated the simulation 1000 times, what we really have is a *sample* of 1000 independent replications, out of an infinite number of possible simulation runs. Our coverage of 95.7% is an *estimate* of what the true coverage would be, if we ran more and more replications. The source of uncertainty of our estimate is called *Monte Carlo simulation error (MCSE)*. We can actually assess the Monte Carlo simulation error in our simulation results using standard statistical procedures for independent and identically distributed data. Here we use a proportion test to check whether the estimated coverage rate is consistent with a true coverage rate of 95%:

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

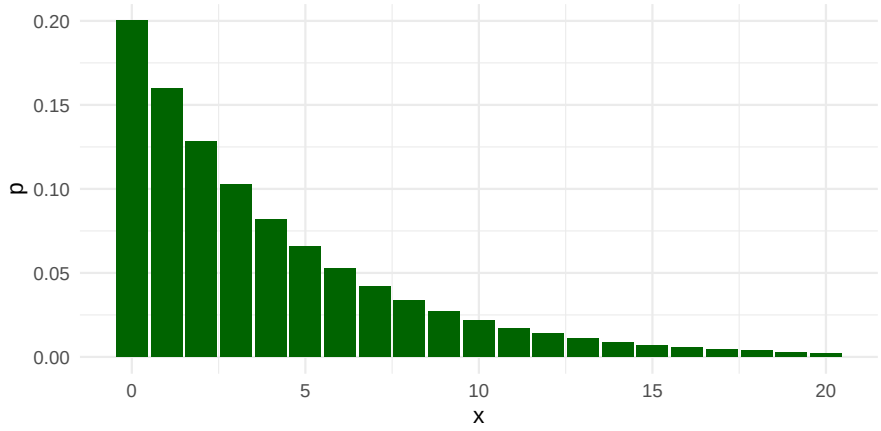
```
    1-sample proportions test with continuity correction

data:  sum(covered) out of length(covered), 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
```

The test indicates that our estimate is consistent with the possibility that the true coverage rate is 95%, just as it should be. Things working out should hardly be surprising. Mathematical theory tells us that the t-test is exact for normally distributed population variables, and we generated data from a normal distribution. In other words, all we have found so far is that the confidence intervals follow theory when the assumptions of the method are met.

3.2 A non-normal population distribution

To see what happens when the normality assumption is violated, let us now look at a scenario where the population variable follows a geometric distribution. The geometric distribution is usually written in terms of a probability parameter p, so that the distribution has a mean of $(1-p)/p$. We will use a geometric distribution with a mean of 4 by setting $p = 1/5$. Here is the population distribution of the variable:



The distribution is highly right-skewed, which suggests that the normal confidence interval might not work very well.

Now let's revise our previous simulation code to use the geometric distribution:

```
rps <- replicate( 1000, {
  dat <- rgeom( n = 10, prob = 1/5 )
  tt <- t.test( dat )
  findInterval( 4, tt$conf.int )
```

```
})
table( rps )
```

```
rps
  0   1   2
  8 892 100
```

Our confidence interval is often entirely too low (such that the population mean is above the interval) and very rarely does our interval fall fully above the population mean. Furthermore, our coverage rate is not the desired 95%:

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

```
[1] 0.892
```

To take account of Monte Carlo error, we will again do a proportion test. The following test result calculates a confidence interval for the true coverage rate under the scenario we are examining:

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

```

    1-sample proportions test with continuity correction

data:  sum(covered) out of length(covered), null probability 0.95
X-squared = 69.605, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is not equal to 0.95
95 percent confidence interval:
 0.8707042 0.9102180
sample estimates:
    p
0.892
```

Our coverage is *too low*; the confidence interval based on the t-test misses the the true value more often than it should. We have learned that the t-test can fail when applied to non-normal (skewed) data.

3.3 Simulating across different scenarios

So far, we have looked at coverage rates of the confidence interval under a single, specific scenario, with a sample size of 10, a population mean of 4, and a geometrically distributed variable. We know from statistical theory (specifically, the central limit theorem) that the confidence interval should work better if the sample size is big enough. But how big does it have to get? One

way to examine this question is to expand the simulation to look at several different scenarios involving different sample sizes. We can think of this as a one-factor experiment, where we manipulate sample size and use simulation to estimate how confidence interval coverage rates change.

To implement such an experiment, we first write our own function that executes the full simulation process for a given sample size:

```
ttest_CI_experiment = function( n ) {

  rps <- replicate( 1000, {
    dat <- rgeom( n = n, prob = 1/5 ) # simulate data
    tt <- t.test( dat )               # analyze data
    findInterval( 4, tt$conf.int )    # evaluate coverage
  })

  coverage <- mean( rps == 1 )       # summarize results

  return(coverage)
}
```

The code inside the body of the function is identical to what we have used above, with the sample size as a function argument, `n`, which enables us to easily run the code for different sample sizes. With our function in hand, we can now run the simulation for a single scenario just by calling it:

```
ttest_CI_experiment(n = 10)
```

```
[1] 0.885
```

Even though the sample size is still `n = 10`, the simulated coverage rate is a little bit different from what we found previously. That is because there is some Monte Carlo error in each simulated coverage rate.

Our task is now to use this function for several different values of n. We could just do this by copy-pasting and changing the value of `n`:

```
ttest_CI_experiment(n = 10)
```

```
[1] 0.922
```

```
ttest_CI_experiment(n = 20)
```

```
[1] 0.91
```

```
ttest_CI_experiment(n = 30)
```

```
[1] 0.914
```

However, this will quickly get cumbersome if we want to evaluate many different sample sizes. A better approach is to use a mapping function from the

`purrr` package.[1] The `map_dbl()` function takes a list of values and calls a function for each value in the list. This accomplishes the same thing as using a `for` loop to iterate through a list of items (if you happen to be familiar with these), but is more succinct.

To proceed, we first create a list of sample sizes to test out:

```
ns <- c(10, 20, 30, 40, 60, 80, 100, 120, 160, 200, 300)
```

Now we can use `map_dbl()` to evaluate the coverage rate for each sample size:

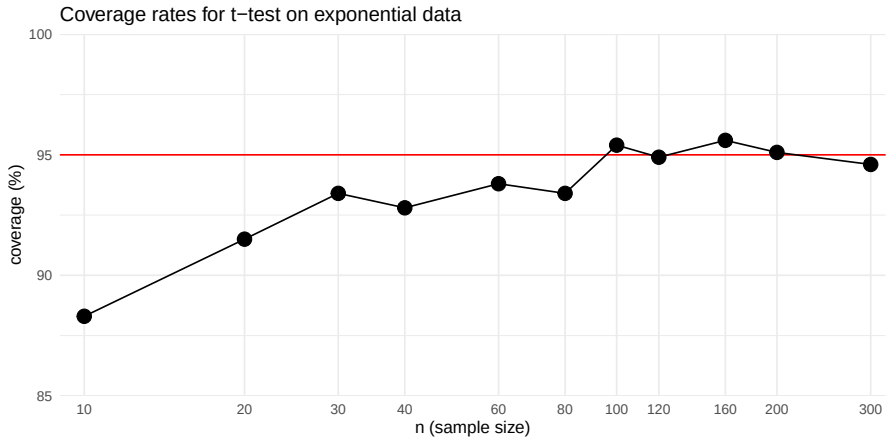
```
coverage_est <- map_dbl( ns, ttest_CI_experiment)
```

This code will run our experiment for each value in `ns`, and then return a vector of the estimated coverage rates for each of the sample sizes.

We advocate for depicting simulation results graphically. To do so, we store the simulation results in a dataset and then create a line plot using a log scale for the horizontal axis:

```
res <- tibble(
  n = ns,
  coverage = 100 * coverage_est
)

ggplot( res, aes( x = n, y = coverage ) ) +
  geom_hline( yintercept=95, col="red" ) +
  # A reference line for nominal coverage rate
  geom_line() +
  geom_point( size = 4 ) +
  scale_x_log10( breaks = ns, minor_breaks = NULL) +
  labs(
    title="Coverage rates for t-test on exponential data",
    x = "n (sample size)",
    y = "coverage (%)"
  ) +
  coord_cartesian(xlim = c(9,320), ylim=c(85,100), expand = FALSE) +
  theme_minimal()
```

[1] See Section 21.5 of R for Data Science (1st edition), which provides an introduction to mapping. Alternately, readers familiar with the `*apply()` family of functions from Base R might prefer to use `lapply()` or `sapply()`, which do essentially the same thing as `purrr::map_dbl()`.



We can see from the graph that the confidence interval's coverage rate improves as sample size gets larger. For sample sizes over 100, the interval appears to have coverage quite close to the nominal 95% level. Although the general trend is pretty clear, the graph is still a bit messy because each point is an *estimated* coverage rate, with some Monte Carlo error baked in.

3.4 Extending the simulation design

So far, we have executed a simple simulation to assess how well a statistical method works in a given circumstance, then expanded the simulation by running a single-factor experiment in which we varied the sample size to see how the method's performance changes. In our example, we found that coverage is below what it should be for small sample sizes, but improves for sample sizes in the 100's.

This example captures all the major steps of a simulation study, which we outlined at the start of Chapter 1. We generated some hypothetical data according to a fully-specified data-generating process: we did both a normal distribution and a geometric distribution, each with a mean of 4. We applied a defined data-analysis procedure to the simulated data: we used a confidence interval based on the t distribution. We assessed how well the procedure worked across replications of the data-generating and data-analysis processes: in this case we focused on the coverage rate of the confidence interval. After creating a function to implement this whole process for a single scenario, we investigated how the performance of the confidence interval changed depending on sample size.

In simulations of more complex models and data-analysis methods, some or all of the steps in the process might have more moving pieces or entail more

complex calculations. For instance, we might want to compare the performance of different approaches to calculating a confidence interval. We might also want to examine how coverage rates are affected by other aspects of the data-generating process, such as looking at different population mean values for the geometric distribution—or even entirely different distributions. With such additional layers of complexity, we will need to think systematically about each of the component parts of the simulation. In the next chapter, we introduce an abstract, general framework for simulations that is helpful for guiding simulation design and managing all the considerations involved.

Exercises

Exercise 3.1. The simulation function we developed in this chapter runs 1000 replications of the data-generating and data-analysis process, which leads to some Monte Carlo error in the reported results. Modify the `ttest_CI_experiment()` function to make the number of replications an input argument, then re-run the simulation and re-create the graph of the results with $R = 10,000$ or even $R = 100,000$. Is the graph more regular than the one in the text, above? Use your improved results to estimate what sample size would be large enough to give coverage of at least 94% (so only 1% off of desired). Is this answer much different from if you had used the figure given in the text?

Exercise 3.2. Modify the `ttest_CI_experiment()` function to make the p parameter an input argument. Repeat the one-factor simulation, but use $p = 1/10$. Make sure your function is comparing coverage to the population mean of $(1 - p)/p$. How do the coverage rates change when p is so small?

Exercise 3.3. Below is a partially completed modified version of the `ttest_CI_experiment()` function that should create a tibble that includes the estimated coverage rate, the average interval length, and a confidence interval for the coverage rate:

```
ttest_CI_experiment_full = function( n ) {

  lotsa_CIs <- replicate( 1000, {
    # simulate data
    dat <- rgeom( n = n, prob = 1/5)
    # analyze data
    tt <- t.test( dat )
    # return CI
    tibble(lower = tt$conf.int[1], upper = tt$conf.int[2])
```

```
  }, simplify = FALSE ) %>%
    bind_rows()

  # summarize results
  # <calculate coverage>
  # <calculate average interval length>
  # <calculate a 95% confidence interval for the true coverage rate>

  return(coverage)
}
```

Complete the function by writing code to compute the estimated coverage rate and average confidence interval length. Also calculate a 95% confidence interval for the true coverage rate (you can use `prop.test()` on your set of simulation coverage indicators to obtain this, treating your R simulation replicates as a random sample in its own right). This confidence interval captures what we call *Monte Carlo error*, which will depend on the number of simulation trials you run. Your modified function should return a one-row tibble with the coverage rate, average confidence interval length, and the lower and upper limits of a CI for the true coverage rate.

Exercise 3.4. Using your work from Exercise 3.3, re-run the simulations to obtain a data frame with each row being a simulation scenario and columns of sample size, estimated coverage, low end of the estimate's confidence interval, high end of the interval, and average confidence interval length. You will likely want to use `map()` and then `bind_rows()` on your list of results; see Section 8.1 for more information about these techniques. Use your resulting set of results to create a graph that depicts the estimated coverage rates as a function of sample size. Make your graph include the 95% confidence intervals also, so that the Monte Carlo simulation error in the estimated coverage rates is represented in the graph. We recommend using the `ggplot2` function `geom_pointrange()` to plot the confidence intervals.

Exercise 3.5. Modify `ttest_CI_experiment()` so that the user can specify the population mean of the data-generating process. Also let the user specify the number of replications to use. Here is a function skeleton to use as a starting point:

```
ttest_CI_experiment <- function( n, pop_mean, reps) {

  pop_prob <- 1 / (pop_mean + 1)

  # <fill in the rest>

  return(coverage)
```

```
}
```

Exercise 3.6. Using the modified function from Exercise 3.5, implement a two-factor simulation study for several different values of `n` and several different population means. One way to do this is to run a few one-factor simulations, each with a different population mean. You can store them in a series of datasets, `res1`, `res2`, `res3`, etc. Then use `bind_rows( size1 = res1, ..., .id = "mean" )` to combine the datasets into a single dataset. Make a plot of your results, with `n` on the x-axis, coverage on the y-axis, and different lines for different population means.

Part II

Structure and Mechanics of a Simulation Study

4

Structure of a simulation study

Monte Carlo simulation is a very flexible tool that researchers use to study a vast array of different models and topics. Within the realm of methodological research, most simulations share a common structure, nearly always involving the same set of steps or component pieces. In learning to design your own simulations, it is very useful to recognize the core components that most simulation studies share. Identifying these components will help you to organize your work and structure the coding tasks involved in writing a simulation.

In this chapter, we outline the component structure of a methodological simulation study, highlighting the four steps involved in a simulation of a single scenario and the three additional steps involved in multifactor simulations. We then describe a strategy for implementing simulations that mirrors the same component structure, where each step in the simulation is represented by a separate function or object. We call this strategy ***tidy, modular simulation***. Finally, we show how the tidy, modular simulation strategy informs the structure and organization of code for a simulation study, walking through basic code skeletons (cf. Section 2.2.5) for each of the steps in a single-scenario simulation.

4.1 General structure of a simulation

The four main steps involved in a simulation study, introduced in Chapter 1, are summarized in the top portion of Table 4.1.

Table 4.1: Steps in the Simulation Process

	Step	Description
1	**Generate**	Generate a sample of artificial data based on a specific statistical model or data-generating process.
2	**Analyze**	Apply one or more data-analysis procedures, estimators, or workflows to the artificial data.

	Step	Description
3	**Repeat**	Repeat steps (1) & (2) R times, recording R sets of results.
4	**Summarize**	Assess the performance of the procedure across the R replications.
5	**Design**	Specify a set of conditions to examine
6	**Execute**	Run the simulation for each condition in the design.
7	**Synthesize**	Compare performance across conditions.

In the simple t-test example presented in Chapter 3, we put each of these steps into action with R code:

- We used the geometric distribution as a data-generating process;
- We used the confidence interval from a one-sample t-test as the data-analysis procedure;
- We repeatedly simulated the confidence intervals with R's `replicate()` function; and
- We summarized the results by estimating the fraction of the intervals that covered the population mean.

We also saw that it was helpful to wrap all of these steps up into a single function, so that we could run the simulation across multiple sample sizes.

These four initial steps are common and shared across nearly all simulations. In our first example, each of the steps was fairly simple, sometimes involving only a single line of code. More generally, each of the steps might be quite a bit more complex. The data-generating process might involve a more complex model with multiple variables or multilevel structure. The data analysis procedure might involve solving a multidimensional optimization problem to get parameter estimates, or might involve a data analysis workflow with multiple steps or contingencies. Further, we might use more than one metric for summarizing the results across replications and describing the performance of the data analysis procedure. Because each of the four steps involves its own set of choices, it will useful to recognize them as distinct from one another.

In methodological research projects, we usually want to examine simulation results across an array of different conditions that differ not only in terms of sample size, but also in other parameters or features of the data-generating process. Running a simulation study across multiple conditions entails several further steps, which are summarized in the bottom portion of Table 4.1. We will first need to specify the factors and specific conditions to be examined in our experimental design. We will then need to execute the simulation for each of the conditions and store all the results for further analysis. Finally, we will need to find ways to synthesize or make sense of the main patterns in the results across all of the conditions in the design.

Just as with the first four steps, it is useful to recognize these further steps as distinct from one another, each involving its own set of choices and techniques. The design step requires choosing which parameters and features of the data-generating process to vary, as well as which specific values to use for each factor that is varied. Executing a simulation might require a lot of computing power, especially if the simulation design has many conditions or the data analysis procedure takes a long time to compute. How to effectively implement the execution step will therefore depend on the computing requirements and available resources. Finally, many different techniques can be used to synthesizing findings from a multifactor simulation. Which ones are most useful will depend on your research questions and the choices you make in each of the preceeding steps.

4.2 Tidy, modular simulations

It is apparent from Table 4.1 that writing a simulation in R involves a large space of possibilities and will requiring making a bunch of decisions. Considering the number of choices to be made, it is critical to stay organized and to approach the process systematically. Recognizing the components of a simulation is the starting point. Next is to see how to translate the components into R code.

In our own methodological work, we have found it very useful to always follow the same approach to writing code for a simulation. We call this approach *tidy, modular simulation*. It involves two simple principles:

1. Implement each component of a simulation as a distinct function or object.
2. Store all results in rectangular data sets.

Writing separate functions for each component step of a simulation has several benefits. The first is the practical benefit of turning the coding process from a monolithic (and potentially daunting) activity into a set of smaller, discrete tasks. This lets us focus on one task at a time and makes it easier to see progress. Second, following this principle makes for code that is easier to read, test, and debug. Rather than having to scan through an entire code file to understand how the data analysis procedure is implemented, we can quickly identify the function that implements it, then focus on understanding the workings of that function. Likewise, if another researcher wanted to test out the data analysis procedure on a dataset of their own, they could do so by running the corresponding function rather than having to dissect an entire script. Third, writing separate code for each component makes it possible to

tweak the code or swap out pieces of the simulation, such as by adding additional estimation methods or trying out a data-generating process involving different distributional assumptions. We already saw this in Chapter 3, where we modified our initial data-generating process to use a geometric distribution rather than a normal distribution. In short, following the first principle makes for simpler, more robust code that is easier to navigate, easier to test, and easier to extend.

The second principle of tidy, modular simulation is to store all results in rectangular datasets, such as the base R `data.frame` object or the tidyverse `tibble` object.[1] This principle applies to any and all output, including the simulated data from Step 1, the results of data analysis procedures from Step 2, full sets of replicated simulation results from Step 3, and summarized results from Step 4. A primary benefit of following this principle is that it facilitates working with the output of each stage in the simulation process. If you are comfortable using R to analyze real data, you you can use the same skills and tools to examine simulation output as long as it is in tabular form. Many of the data processing and data analysis tools available in R work with—or even require—rectangular datasets. Thus, using rectangular datasets makes it easier to inspect, summarize, and visualize the output.

4.3 Skeleton of a simulation study

The principles of tidy simulation imply that code for a simulation study should usually follow the same broad outline and organization of Table 4.1, with custom functions for each step in process. We will describe the outlines of simulation code using function skeletons to illustrate the inputs and outputs of each component. These skeletons skip over all the specific details, so that we can see the structure more clearly. We will first examine the structure of the code for simulating one specific scenario, then consider how to extend the code to systematically explore a variety of scenarios, as in a multifactor simulation.

In code skeletons, the structure of the first four steps in a simulation looks like this:

```
# Generate (data-generating process)

generate_data <- function( model_params ) {
  # stuff
  return(data)
```

[1] Wickham (2014) provides a broader introduction to the concept of tidy data in the context of data-analysis tasks.

```
}

# Analyze (data-analysis procedure)

analyze <- function( data ) {
  # stuff
  return(one_result)
}

# Repeat

one_run <- function( model_params ) {
  dat <- generate_data( model_params )
  one_result <- analyze(dat)
  return(one_result)
}

results <- replicate(R, one_run( params ))

# Summarize (calculate performance measures)

assess_performance <- function( results, model_params ) {
  # stuff
  return(performance_measures)
}

assess_performance(results, model_params)
```

The code above shows the full skeleton of a simulation. It involves four functions, where the outputs of one function get used as inputs in subsequent functions. We will now look at the inputs and outputs of each function to see how they align with the four steps in the simulation process. Subsequent chapters examine each piece in much greater detail—putting meat on the bones of each function skeleton, to torture our metaphor—and discuss specific statistical issues and programming techniques that are useful for designing each component.

Besides illustrating the skeletal framework of a simulation, readers might find it useful to use it as a template from which to start writing their own code. The `simhelpers` package includes the function `create_skeleton()`, which will open a new R script that contains a template for a simulation study, with sections corresponding to each component:

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

The template that appears is a slightly more elaborate version of the code

above, with the main difference being that it also includes some additional lines of code to wire the pieces together for a multifactor simulation. Starting from this template, you will already be on the road to writing a tidy, modular simulation.

4.3.1 Data-Generating Process

The first step in a simulation is specifying a data-generating process. This is a hypothetical model for how data might arise, involving measurements or observations of one or more variables. The bare-bones skeleton of a data-generating function looks like the following:

```
generate_data <- function( model_params ) {
  # stuff
  return(data)
}
```

The function takes as input a set of model parameter values, denoted here as `model_params`. Based on those model parameters, the function generates a hypothetical dataset as output. Generating our own data based on a model allows us to know what the answer is (e.g., the true population mean or the true average effect of an intervention), so that we have benchmark against which to compare the results of a data analysis procedure that generates noisy estimates of the true value.

In practice, `model_params` will usually not be just one input but rather multiple arguments. These arguments might include inputs such as the population mean for a variable, the standard deviation of a distribution of treatment effects, or a parameter controlling the degree of skewness in the population distribution. Many data-generating processes involving multiple variables, such as the response variable and predictor variables in a regression model. In such instances, the inputs of `generate_data()` might also include parameters that determine the degree of dependence or correlation between variables. Further, the `generate_data()` inputs will also usually include arguments relating to the sample size and structure of the hypothetical dataset. For instance, in a simulation of a multilevel dataset where individuals are nested within clusters, the inputs might include an arguments to specify the total number of clusters and the average number of individuals per cluster. We discuss the inputs and form of the data-generating function further in Chapter 6.

4.3.2 Data Analysis Procedure

The second step in a simulation is specifying a data analysis procedure or set of procedures. The bare-bones skeleton of a data-generating function looks like the following:

```
analyze <- function( data ) {
  # stuff
  return(one_result)
}
```

The function should take a single dataset as input and produce a set of estimates or results (e.g., point estimates, standard errors, confidence intervals, p-values, predictions, etc.). Because we will be using the function to analyze hypothetical datasets simulated from the data-generating process, the `analyze()` function needs to work with `data` inputs that are produced by the `generate_data()` function. Thus, the code in the body of `analyze()` can assume that `data` will include relevant variables with specific names.

Inside the body of the function, `analyze()` includes code to carry out a data analysis procedure. This might involve generating a confidence interval, as in the example from Chapter 3. In another context, it might involve estimating an average growth rate along with a standard error, given a dataset with longitudinal repeated measurements from multiple individuals. In still another context, it might involve generating individual-level predictions from a machine learning algorithm. In simulations that involve comparing multiple analysis methods, we might write an `analyze()` function for each of the methods of interest, or (generally less preferred because it is less modular) we might write one function that does the calculations for all of the methods together.

A well-written estimation method should, in principle, work not only on a simulated, hypothetical dataset but also on a real empirical dataset that has the same format (i.e., appropriate variable names and structure). Because of this, the inputs of the `analyze()` function should not typically include any information about the parameters of the data-generating process. To be realistic, the code for our simulated data-analysis procedure should not make use of anything that the analyst could not know when analyzing a real dataset. Thus, `analyze()` has an argument for the sample dataset but not for `model_params`. We discuss the form and content of the data analysis function further in Chapter 7.

4.3.3 Repetition

The third step in a simulation is to repeatedly evaluate the data-generating process and data analysis procedure. In practice, this amounts to repeatedly calling `generate_data()` and then calling `analyze()` on the result. Here is the skeleton from our simulation template:

```
one_run <- function( model_params ) {
  dat <- generate_data( model_params )
  one_result <- analyze(dat)
```

```
  return(one_result)
}

results <- simhelpers::repeat_and_stack(R, one_run( params ))
```

We first create a helper function called `one_run()`, which takes `model_params` as input. Inside the body of the function, we call the `generate_data()` function to simulate a hypothetical dataset. We pass this dataset to `analyze()` and return a small dataset containing the results of the data-analysis procedure. The `one_run()` method is like a coordinator or dispatcher of the system: it generates the data, calls all the evaluation methods we want to call, combines all the results, and hands them back for recording. Making a helper method such as `one_run()` can be useful because it facilitates debugging.

Once we have the `one_run()` helper function, we need a way to call it repeatedly. As with many things in R, there are a variety of different ways to do something over and over. In the above skeleton, we use the `repeat_and_stack()` function from `simhelpers`.[2] In the first argument, we specify the number of times to repeat the process. In the second argument, we specify an expression that evaluates `one_run()` for specified values of the model parameters stored in `params`. The `repeat_and_stack()` function then evaluates the expression repeatedly, `R` times in all, and then stacks up all of the replications into a big dataset, with one or more rows per replication.[3]

We go into further detail about how to approach running the simulation process in Chapter 8. Among other things, we will illustrate how to use the `bundle_sim()` function from the `simhelpers` package to automate the process of coding this step, thereby avoiding the need to write a `one_run()` helper function.

---

[2] In the example from Chapter 3, we used the `replicate()` function from base R to repeat the process of generating and analyzing data. This function is a fine alternative to the `repeat_and_stack()` approach demonstrated in the skeleton. The only drawback is that it requires some further work to combine the results across replications. Here is a different version of the skeleton, which uses `replicate()` instead of `repeat_and_stack()`:

```
results_list <- replicate(n = R, expr = {
  dat <- generate_data( params )
  one_result <- analyze(dat)
  return(one_result)
}, simplify = FALSE)

results <- purrr::list_rbind(results_list)
```

This version of the skeleton does not create a `one_run()` helper function, but instead puts the code from the body of `one_run()` directly into the `expr` argument of `replicate()`. To learn about other ways of repeatedly evaluating the simulation process, see Appendix Section 8.1.

[3] If you would prefer the output as a list rather than a stacked dataset, set `repeat_and_stack()`'s optional argument `stack = FALSE`.

4.3.4 Performance summaries

The fourth step in a simulation is to summarize the distribution of simulation results across replications. Here is the skeleton from our simulation template:

```
assess_performance <- function( results, model_params ) {
  # stuff
  return(performance_measures)
}

assess_performance(results, params)
```

The `assess_performance()` function takes `results` as input. `results` should be a dataset containing all of the replications of the data-generating and analysis process. In contrast to the `analyze()` function, `assess_performance()` also needs to know the true parameter values of the data-generating process, so it needs to have `model_params` as its other input. The function then uses both of these inputs to calculate performance measures and returns a summary of the performance measures in a dataset.

Performance measures are the metrics or criteria used to assess the performance of a statistical method across repeated samples from the data-generating process. For example, we might want to know how close an estimator gets to the target parameter, on average. We might want to know if a confidence interval captures the true parameter the right proportion of the time, as in the simulation from Chapter 3. Performance is defined in terms of the sampling distribution of estimators or analysis results, across an infinite number of replications of the data-generating process. In practice, we use many replications of the process, but still only a finite number. Consequently, we actually *estimate* the performance measures and need to attend to the Monte Carlo error in the estimates. We discuss the specifics of different performance measures and assessment of Monte Carlo error in Chapter 9.

4.3.5 Multifactor simulations

Thus far, we have sketched out the structure of a modular, tidy 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 the observations follow geometric distribution. However, we rarely want to examine a method only in a single context. Typically, we want to explore how well a procedure works across a range of different contexts. If we choose conditions in a structured and thoughtful manner, we will be able to examine broad trends and potentially make more general claims about the behaviors of the data-analysis procedures under investigation. Thus, it is helpful to think of simulations as akin to a designed experiment: in seeking to understand the properties of one or more procedures, we test them under a variety of different scenarios to see how they perform, then seek to identify more general patterns that hold beyond the

specific scenarios examined. This is the heart of simulation for methodological evaluation.

To implement a multifactor simulation, we will follows the same principles of modular, tidy simulation. In particular, we will take the code developed for simulating a single context and bundle it into a function that can be evaluated for any and all scenarios of interest. Simulation studies often 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), and these too can be represented in an object, distinct from the other components of the simulation. We will discuss multiple-scenario simulations in Part III (starting with Chapter 10), after we more fully develop the core concepts and techniques involved in simulating a single context.

Exercises

Exercise 4.1. Look back at the t-test simulation presented in Chapter 3. The code presented there did not entirely follow the formal structure outlined in this chapter. Revise the code by creating separate functions for each of four components in the simulation skeleton. Using the functions, re-run the simulation and recreate one or more graphs from Exercise 3.1 or Exercise 3.4.

5

Case Study: Heteroskedastic ANOVA and Welch

In this chapter, we present another detailed example of a simulation study to demonstrate how to put the principles of tidy, modular simulation into practice. To illustrate the process of programming a simulation, we reconstruct the simulations from Brown and Forsythe (1974). We will also use this case study as a recurring example in some of the following chapters.

Brown and Forsythe (1974) studied methods for null hypothesis testing in studies that measure a characteristic X on samples from each of several groups. They 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 the characteristic X. A researcher obtains samples of size $n_1, ..., n_G$ from each of the groups and takes measurements of the characteristic for each sampled unit. Let x_{ig} denote the measurement from unit i in group g, for $i = 1, ..., n_g$ for each $j = 1, ..., G$. The researcher's goal is to use the sample data to test the hypothesis that the population means are all equal

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

If the population *variances* were all equal (so $\sigma_1^2 = \sigma_2^2 = \cdots = \sigma_G^2$), we could use a conventional one-way analysis of variance (ANOVA) to conduct this 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 the null of equal group means, allowing for the possibility that variances could differ across groups?

To tackle this question, Brown and Forsythe evaluated two different hypothesis testing procedures, developed by James (1951) and Welch (1951), both of which avoid the assumption that all groups have equal equality of variances. Brown and Forsythe also evaluated the conventional one-way ANOVA F-test as a benchmark, even though this procedure maintains the assumption of equal variances. They also proposed and evaluated a new procedure of their own devising.[1] Their simulation involved comparing the performance of these

[1] This latter piece makes Brown and Forsythe's study a prototypical example of a statistical methodology paper: find some problem that current procedures do not perfectly solve,

Table 5.1: Simulation scenarios explored by Brown and Forsythe (1974)

Scenario	Groups	Sample Sizes	Standard Deviations
A	4	4,4,4,4	1,1,1,1
B	4		1,2,2,3
C	4	4,8,10,12	1,1,1,1
D	4		1,2,2,3
E	4		3,2,2,1
F	4	11,11,11,11	1,1,1,1
G	4		1,2,2,3
H	4	11,16,16,21	1,1,1,1
I	4		3,2,2,1
J	4		1,2,2,3
K	6	4,4,4,4,4,4	1,1,1,1,1,1
L	6		1,1,2,2,3,3
M	6	4,6,6,8,10,12	1,1,1,1,1,1
N	6		1,1,2,2,3,3
O	6		3,3,2,2,1,1
P	6	6,6,6,6,6,6	1,1,2,2,3,3
Q	6	11,11,11,11,11,11	1,1,2,2,3,3
R	6	16,16,16,16,16,16	1,1,2,2,3,3
S	6	21,21,21,21,21,21	1,1,2,2,3,3
T	10	20,20,20,20,20,20,20,20,20,20	1,1,1.5,1.5,2,2,2.5,2.5,3,3

different hypothesis testing procedures (the methods) under a range of conditions (different data generating processes) with different sample sizes and different degrees of heteroskedasticity. They looked at the different scenarios shown as Table 5.1, varying number of groups, group size, and amount of variation within each group. In all, there are a total of 20 scenarios, covering conditions with between 10 and 6 groups.

When evaluating hypothesis testing procedures, there are two main performance metrics of interest: type-I error rate and power. The type-I error rate is the rate at which a test rejects the null hypothesis when the null hypothesis is actually true. To apply a hypothesis testing procedure, one has to specify

invent something to do a better job, and then do simulations and/or math to build a case that the new procedure is better.

Table 5.2: Portion of "Table 1" reproduced from Brown and Forsythe (1974)

	ANOVA F test			B									2.4
Scenario	10A	10.2	4.9	0.9	7.8	3.4	0.5	9.6	4.5	0.8	13.3	7.9	
B	12.0	6.7	1.7	8.9	4.1	0.7	10.3	4.7	0.8	13.8	8.1	2.7	
C	9.9	5.1	1.1	9.5	4.8	1.0	10.8	5.7	1.6	12.1	6.7	2.1	
D	5.9	3.0	0.6	10.3	5.7	1.4	9.8	4.9	0.9	10.8	5.6	1.3	
E	21.9	14.4	5.6	11.0	6.2	1.8	11.3	6.5	2.0	12.9	7.7	2.9	
F	10.1	5.1	1.0	9.8	5.7	1.5	10.0	5.0	0.9	10.6	5.5	1.1	
G	11.4	6.3	1.8	10.7	5.7	1.5	10.1	5.0	1.1	10.6	5.4	1.3	
H	10.3	4.9	1.1	10.3	5.1	1.0	10.2	5.0	1.0	10.5	5.3	1.2	
I	17.3	10.8	3.9	11.1	6.2	1.8	10.5	5.5	1.2	10.9	5.8	1.3	
J	7.3	4.0	1.0	11.5	6.5	1.8	10.6	5.4	1.1	10.9	5.6	1.1	
K	9.6	4.9	1.0	7.3	3.4	0.4	11.4	6.1	1.4	14.7	9.5	3.8	

a desired, or nominal, type-I error rate, often denoted as the α-level. For a specified α, a valid or well-calibrated test should have an actual type-I error rate less than or equal to the nominal level, and ideally should be very close to nominal. Power is how often a test correctly rejects the null when it is indeed false. It is a measure of how sensitive a method is to violations of the null.

Brown and Forsythe estimated 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"). Our Table 5.2 reports some of their results with respect to Type I error. For a well-calibrated hypothesis testing method, the reported numbers should be very close to the desired alpha levels, as listed at the top of the table. We can compare the four tests to each other across each row, where each row is a specific scenario defined by a specific data generating process. Looking at ANOVA, for example, we see some scenarios with very elevated rates. For instance, in Scenario E, the ANOVA F-test has 21.9% rejection when it should only have 10%. In contrast, the ANOVA F works fine under scenario A, which is what one would expect because all the groups have the same variance. Brown and Forsythe's choice of scenarios here illustrates a broader design principle: to provide a full picture of the performance of a method or set of methods, it is wise to always evaluate them under conditions where we expect things to work, as well as conditions where we expect them to not work well.

To replicate the Brown and Forsythe simulation, we will first write functions to generate data for a specified scenario and evaluate data of a given structure. We will then use these functions to evaluate the hypothesis testing procedures

in a specific scenario with a particular set of parameters (e.g., sample sizes, number of groups, and so forth). We will then scale up to execute the simulations for a range of scenarios that vary the parameters of the data-generating model, just as reported in Brown and Forsythe's paper.

5.1 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 starting point for developing a function. For example, say that we have four groups with means of 1, 2, 5, 6; variances of 3, 2, 5, 1; and sample sizes of 3, 6, 2, 4:

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

Following Brown and Forsythe (1974), we will assume that the measurements are normally distributed within each sub-group of the population. The following code generates a vector of group id's and a vector of simulated measurements:

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

# group id factor
group <- factor(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
   <fct> <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 we have the pieces needed to generate a small dataset consisting of group memberships and the measured characteristic:

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

```
# A tibble: 15 x 2
   group      x
   <fct>  <dbl>
 1 1     -0.236
 2 1      1.15
 3 1      2.36
 4 2      2.10
 5 2      0.156
 6 2      1.28
 7 2      2.06
 8 2      2.33
 9 2      3.15
10 3      6.29
11 3      4.05
12 4      5.22
13 4      5.29
14 4      7.99
15 4      6.28
```

We have followed the strategy of first constructing a dataset with parameters for each observation in each group, making heavy use of base R's `rep()` function to repeat values in a list. We then called `rnorm()` to generate `N` observations in all. This works because `rnorm()` is *vectorized*; if you give it a vector (or vectors) of parameter values, it will generate each subsequent

observation according to the next entry in the vector. As a result, the first `x` value is simulated from a normal distribution with mean `mu_long[1]` and standard deviation `sd_long[1]`, the second `x` is simulated using `mu_long[2]` and `sd_long[2]`, and so on.

As usual, there are many different and legitimate ways of doing this in R. For instance, instead of using `rep()` to do it all at once, we could generate observations for each group separately, then stack the observations into a dataset. Do not worry about trying to writing code the "best" way—especially when you are initially putting a simulation together. If you can find a way to accomplish your task at all, then that's often enough (and you should feel good about it!).

5.1.1 Now make a function

Because we will need to generate datasets over and over, we will wrap our code in a function. The inputs to the function will be the parameters of the model that we specified at the very beginning: the set of population means `mu`, the population variances `sigma_sq`, and sample sizes `sample_size`. We make these quantities arguments of the data-generating function so that we can make datasets of different sizes and shapes:

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

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

  group <- factor(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)
}
```

The function is simply the code we built previously, all bundled up. We developed the function by first writing code to make the data-generating process to work once, the way we want, and only then turning the final code into a function for later reuse.

Once we have turned the code into a function, we can call it to get a new set of simulated data. For example, to generate a dataset with the same parameters as before, we can do:

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

str(sim_data)
```

```
tibble [15 x 2] (S3: tbl_df/tbl/data.frame)
 $ group: Factor w/ 4 levels "1","2","3","4": 1 1 1 2 2 2 2 2 2 3 ...
 $ x    : num [1:15] 1.436 0.95 1.338 3.195 0.779 ...
```

To generate one with population means of zero in all the groups, but the same group variances and sample sizes as before, we can do:

```
sim_data_null <- generate_ANOVA_data(
  mu = c( 0, 0, 0, 0 ),
  sigma_sq = sigma_sq,
  sample_size = sample_size
)

str(sim_data)
```

```
tibble [15 x 2] (S3: tbl_df/tbl/data.frame)
 $ group: Factor w/ 4 levels "1","2","3","4": 1 1 1 2 2 2 2 2 2 3 ...
 $ x    : num [1:15] 1.436 0.95 1.338 3.195 0.779 ...
```

Following the principles of tidy, modular simulation, we have written a function that returns a rectangular dataset for further analysis. Also note that the dataset returned by `generate_ANOVA_data()` only includes the variables `group` and `x`, but not `mu_long` or `sd_long`. This is by design. Including `mu_long` or `sd_long` would amount to making the population parameters available for use in the data analysis procedures, which is not something that happens when analyzing real data.

5.1.2 Cautious coding

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.

Developing code in this way is not without hazards. In particular, after we have created our function, our workspace is left with a variable `mu` in it and our function also has a parameter named `mu`. Inside the function, R will use the parameter `mu` first, but this is potentially confusing. Another potential source of confusion are lines such as `mu = mu`, which means "set the function's

parameter called `mu` to the variable called `mu`." These are different things (with the same name).

Once you have built a function, one way to check that it is working properly is to comment out the initial code (or delete it), clear out the workspace (or restart R), and then re-run the code that uses the function. If things still work, then you can be somewhat confident that you have successfully bundled your code into the function. Once you bundle your code, you can also do a search and replace to change the variable names inside your function to something more generic, to better clarify the distinction betwen object names and argument names.

5.2 The hypothesis testing procedures

Brown and Forsythe considered four different hypothesis testing procedures for heteroskedastic ANOVA, but we will focus on just two of the tests for now. We start with the conventional one-way ANOVA that mistakenly assumes homoskedasticity. R's `oneway.test` function will calculate this test automatically:

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

anova_F <- oneway.test(x ~ group, data = sim_data, var.equal = TRUE)
anova_F
```

```
    One-way analysis of means

data:  x and group
F = 3.1337, num df = 3, denom df = 11, p-value = 0.0695
```

We can use the same function to calculate Welch's test by setting `var.equal = FALSE`:

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

```
    One-way analysis of means (not assuming equal variances)

data:  x and group
```

```
F = 16.131, num df = 3.0000, denom df = 3.4645, p-value
= 0.01606
```

The main results we need here are the p-values of the tests, which will let us assess 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-values from the one-way ANOVA test and Welch test:

```
ANOVA_Welch_F <- function(data) {
  anova_F <- oneway.test(x ~ group, data = data, var.equal = TRUE)
  Welch_F <- oneway.test(x ~ group, data = data, var.equal = FALSE)

  result <- tibble(
    ANOVA = anova_F$p.value,
    Welch = Welch_F$p.value
  )

  return(result)
}
```

```
ANOVA_Welch_F(sim_data)
```

```
# A tibble: 1 x 2
   ANOVA  Welch
   <dbl>  <dbl>
1 0.0695 0.0161
```

Following our tidy, modular simulation principles, this function returns a small dataset with the p-values from both tests. Eventually, we might want to use this function on some real data. Our estimation function does not care if the data are simulated or not; we call the input `data` rather than `sim_data` to reflect this.

As an alternative to this function, we could instead write code to implement the ANOVA and Welch tests ourselves. This has some potential advantages, such as avoiding any extraneous calculations that `oneway.test` does, which take time and slow down our simulation. However, there are also drawbacks to doing so, including that writing our own code takes *our* time and opens up the possibility of errors in our code. For further discussion of the trade-offs, see Section A.4, where we do implement these tests by hand and see what kind of speed-ups we can obtain.

5.3 Running the simulation

We now have functions that implement steps 2 and 3 of the simulation. Given some parameters, `generate_ANOVA_data` produces a simulated dataset and, given some data, `ANOVA_Welch_F` calculates p-values two different ways. We now want to know which way is better, and by how much. To answer this question, we will need to repeat the chain of generate-and-analyze calculations a bunch of times. To facilitate repetition, we first put the components together into a single function:

```
one_run = function( mu, sigma_sq, sample_size ) {
  sim_data <- generate_ANOVA_data( mu = mu, sigma_sq = sigma_sq, sample_size = sample_size
  ANOVA_Welch_F(sim_data)
}

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

```
# A tibble: 1 x 2
     ANOVA   Welch
     <dbl>   <dbl>
1 0.000460 0.00132
```

This function implements a single simulation trial by generating artificial data and then analyzing the data, ending with a tidy dataset that has results for the single run.

We next call `one_run()` over and over; see Appendix Section 8.1 for some discussion of options. The following uses `repeat_and_stack()` from `simhelpers` to evaluate `one_run()` 4 times and then stack the results into a single dataset:

```
library(simhelpers)

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

```
# A tibble: 4 x 2
      ANOVA    Welch
      <dbl>    <dbl>
1 0.0000566 0.000238
2 0.000996  0.0226
3 0.0000974 0.00356
4 0.00583   0.0477
```

Voila! We have simulated p-values!

5.4 Summarizing test performance

We now have all the pieces in place to reproduce the results from Brown and Forsythe (1974). We first 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. To evaluate the type-I error rate, we need to simulate data from a process where the population means are indeed all equal. Arbitrarily, let's start with $G = 4$ groups and set all of the means equal to zero:

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

In the fifth row of Table 1 (Scenario E in our Table 5.1), 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 `map_dfr` to simulate 10,000 p-values:

```
p_vals <- repeat_and_stack(10000,
  one_run(
    mu = mu,
    sigma_sq = sigma_sq,
    sample_size = sample_size
  )
)
```

We can estimate the rejection rates by summarizing across these replicated p-values. The rule is that the null is rejected if the p-value is less than α. To get the rejection rate, we calculate the proportion of replications where the null is rejected:

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

```
[1] 0.1416
```

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

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

```
[1] 0.1416
```

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

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

```
[1] 0.0651
```

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 have not 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*). In Chapter 9, we will look more at how to estimate and control the Monte Carlo simulation error in performance measures.

So there you have it! Each part of the simulation is a distinct block of code, and together we have a modular simulation that can be easily extended to other scenarios or other tests. The exercises at the end of this chapter ask you to extend the framework further. In working through them, you will get to experience first-hand how the modular code that we have started to develop is easier to work with than a single, monolithic block of code.

Exercises

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

Exercise 5.1 (Other α's)**.** Table 1 from Brown and Forsythe (1974) 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 and compare to the table.

Exercise 5.2 (Compare results)**.** 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 report results?

Exercise 5.3 (Power)**.** In the primary paper, Table 1 is about Type I error and Table 2 is about power. A portion of Table 2 follows:

In the table, the sizes of the four groups are 11, 16, 16, and 21, for all the scenarios. Try simulating the **power levels** for a couple of sets of parameter values from Table 5.3. Use 10,000 replications. How do your results compare to the results reported in the Table?

Table 5.3: Portion of "Table 2" reproduced from Brown and Forsythe (1974)

Variances	Means	Brown's F	B	F's F*	Welch's W
1,1,1,1	0,0,0,0	4.9	5.1	5.0	
	1,0,0,0	68.6	67.6	65.0	
3,2,2,1	0,0,0,0	-	6.2	5.5	
	1.3,0,0,1.3	-	42.4	68.2	

Exercise 5.4 (Wide or long?)**.** Instead of making `ANOVA_Welch_F` return a single row with the columns for the p-values, one could instead return a dataset with one row for each test. The "long" approach is often nicer when evaluating more than two methods, or when each method returns not just a p-value but other quantities of interest. For our current simulation, we might also want to store the F statistic, for example. The resulting dataset would then look like the following:

```
ANOVA_Welch_F_long(sim_data)
```

```
# A tibble: 2 x 3
  method Fstat   pvalue
  <chr>  <dbl>    <dbl>
1 ANOVA   15.4 0.000295
2 Welch   10.2 0.0332
```

Modify `ANOVA_Welch_F()` to return output in the long format, update your simulation code, and then use `group_by()` plus `summarise()` to calculate rejection rates of both tests. The `group_by()` method divides your data into distinct groups so you can easily conduct an operation on each. Here is a typical way that `group_by()` might be used for calculating performance measures:

```
sres <-
  res %>%
  group_by( method ) %>%
  summarise( rejection_rate = mean( pvalue < 0.05 ) )
```

Exercise 5.5 (Other tests)**.** The `onewaytests` package in R includes functions for calculating Brown and Forsythe's F^* test and James' test for differences in population means. Modify the data analysis function `ANOVA_Welch_F` (or, better yet, `ANOVA_Welch_F_long` from Exercise 5.4) to also include results from these hypothesis tests. Re-run the simulation to estimate the type-I error rate of all four tests under Scenarios A and B of Table 5.1.

Exercise 5.6 (Methodological extensions)**.** What other methodological questions might you want to investigate about the one-way ANOVA problem? List

two or three questions that could be investigated with further simulations.

Exercise 5.7 (Power analysis)**.** Suppose you were conducting an experimental study involving several groups (perhaps $G = 3$ or 4 or 5). Because of logistical constraints, one of the groups will need to include at least 40% of the full sample, and you anticipate that this group might have more variable outcomes than the other groups. Your collaborators want to know how large a total sample they will need to recruit in order to have a high probability of detecting significant differences between groups. How could you use the functions you've developed to answer this question? List out the steps in your approach, and make note of any further information or assumptions you will need in order to answer the question.

6

Data-generating processes

As we saw in Chapter 4, the first step of a simulation is creating artificial data based on some process where we know (and can control) the truth. This step is what we call the data generating process (DGP). Think of it as a recipe for cooking up artificial data, which can be applied over and over, any time we're hungry for a new dataset. Like a good recipe, a good DGP needs to be complete—it cannot be missing ingredients and it cannot omit any steps. Unlike cooking or baking, however, DGPs are usually specified in terms of a statistical model, or a set of equations involving constants, parameter values, and random variables. More complex DGPs, such as those for hierarchical data or other latent variable models, will often involve a series of several equations that describe different dimensions or levels of the model, which need to be followed in sequence to produce an artificial dataset.

In this chapter, we will look at how to instantiate a DGP as an R function (or perhaps a set of functions). Designing DGPs and implementing them in R code involves making choices about what aspects of the model we want to be able to control and how to set up the parameters of the model. We start by providing a high-level overview of DGPs and discussing some of the choices and challenges involved in designing them. We then demonstrate how to write R functions for DGPs. In the remainder, we present detailed examples involving a hierarchical DGP for generating data on students nested within schools and an item response theory DGP for generating data on responses to individual test items.

6.1 Examples

Before diving in, it is helpful to consider a few examples that we will return to throughout this and subsequent chapters.

6.1.1 Example 1: One-way analysis of variance

We have already seen one example of a DGP in the ANOVA example from Chapter 5. Here, we consider observations on some variable X drawn from a population consisting of G groups, where group g has population mean μ_g and population variance σ_g^2 for $g = 1, ..., G$. A simulated dataset consists of n_g observations from each group $g = 1, ..., G$, where X_{ig} is the measurement for observation i in group g. The statistical model for these data can be written as follows:

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

for $i = 1, ..., n_g$ and $g = 1, ..., G$. Alternately, we could write the model as

$$X_{ig} \sim N(\mu_g, \sigma_g^2).$$

6.1.2 Example 2: Bivariate Poisson model

As a second example, suppose that we want to understand how the usual Pearson sample correlation coefficient behaves with non-normal data and also to investigate how the Pearson correlation relates to Spearman's rank correlation coefficient. To look into such questions, one DGP we might entertain is a bivariate Poisson model, which is a distribution for a pair of counts, C_1, C_2, where each count follows a Poisson distribution and where the pair of counts may be correlated. We will denote the expected values of the counts as μ_1 and μ_2 and the Pearson correlation between the counts as ρ.

To simulate a dataset based on this model, we would first need to choose how many observations to generate. Call this sample size N. One way to generate data following a bivariate Poisson model is to generate *three* independent Poisson random variables for each of the N observations:

$$\begin{aligned} Z_0 &\sim Pois\left(\rho\sqrt{\mu_1\mu_2}\right) \\ Z_1 &\sim Pois\left(\mu_1 - \rho\sqrt{\mu_1\mu_2}\right) \\ Z_2 &\sim Pois\left(\mu_2 - \rho\sqrt{\mu_1\mu_2}\right) \end{aligned}$$

and then combine the pieces to create two dependent observations:

$$\begin{aligned} C_1 &= Z_0 + Z_1 \\ C_2 &= Z_0 + Z_2. \end{aligned}$$

An interesting feature of this model is that the range of possible correlations is constrained: only positive correlations are possible and, because each of the independent pieces must have a non-negative mean, the maximum possible correlation is $\sqrt{\frac{\min\{\mu_1,\mu_2\}}{\max\{\mu_1,\mu_2\}}}$.

6.1.3 Example 3: Hierarchical linear model for a cluster-randomized trial

Cluster-randomized trials are randomized experiments where the unit of randomization is a *group* of individuals, rather than the individuals themselves. For example, suppose we have a collection of schools and the students within them. A cluster-randomized trial involves randomizing the *schools* into treatment or control conditions and then measuring an outcome such as academic performance on the multiple students within the schools. Typically, researchers will be interested in the extent to which average outcomes differ across schools assigned to different conditions, which captures the impact of the treatment relative to the control condition. We will index the schools using $j = 1, ..., J$ and let n_j denote the number of students observed in school j. Say that Y_{ij} is the outcome measure for student i in school j, for $1 = 1, ..., n_j$ and $j = 1, ..., J$, and let Z_j be an indicator equal to 1 if school j is assigned to the treatment condition and otherwise equal to 0.

A widely used approach for estimating impacts from cluster-randomized trials is heirarchical linear modeling (HLM). One way to write an HLM is in two parts. First, we consider a regression model that describes the distribution of the outcomes across students within school j:

$$Y_{ij} = \beta_{0j} + \epsilon_{ij}, \qquad \epsilon_{ij} \sim N(0, \sigma^2_\epsilon),$$

where β_{0j} is the average outcome across students in school j. Second, we allow that the school-level average outcomes differ by a treatment effect γ_1 and that, for schools within each condition, the average outcomes follow a normal distribution with variance σ^2_u. We can write these relationships as a regression equation for the school-specific average outcome:

$$\beta_{0j} = \gamma_0 + \gamma_{10} Z_j + u_{0j}, \quad u_{0j} \sim N(0, \tau^2),$$

where γ_0 is the average outcome among schools in the control condition.

If we only consider the first stage of this model, it looks a bit like the one-way ANOVA model from the previous example: in both cases, we have multiple observations from each of several groups.
The main distinction is that the ANOVA model treats the G groups as a fixed set, whereas the HLM treats the set of J schools as sampled from a larger population of schools and includes a regression model describing the variation in the school-level average outcomes.

6.2 Components of a DGP

A DGP involves a statistical model with parameters and random variables, but it also often includes further details as well, beyond those that we would

consider to be part of the model as we would use it for analyzing real data. In statistical analysis of real data, we often use models that describe only *part* of the distribution of the data, rather than its full, multivariate distribution. For instance, when conducting a regression analysis, we are analyzing the distribution of an outcome or response variable, conditional on a set of predictor variables. In other words, we take the predictor variables as *given* or *fixed*, rather than modeling their distribution. When using an item response theory (IRT) model, we use responses to a set of items to estimate individual ability levels, given the set of items on the test. We do not (usually) model the items themselves. In contrast, if we are going to *generate* data for simulating a regression model or IRT model, we need to specify distributions for these additional features (the predictors in a regression model, the items in an IRT model); we can no longer just take them as given.

In designing and discussing DGPs, it is helpful to draw distinctions between the components of the focal statistical model and the remaining components of the DGP that are taken as given when analyzing real data. A first relevant distinction is between structural features, covariates, and outcomes (or more generally, endogenous quantities):

- **Structural features** are quantities, such as the per-group sample sizes in the one-way ANOVA example, that describe the structure of a dataset but do not enter directly into the focal statistical model. When analyzing real data, we usually take the structural features as they come, but when simulating data, we will need to make choices about the structural features. For instance, in the HLM example involving students nested within schools, the number of students in each school is a structural feature. To simulate data based on HLM, we will need to make choices about the number of schools and the distribution of the number of students in each school (e.g., we might specify that school sizes are uniformly distributed between specified minimum and maximum sizes), even though we do not have to consider these quantities when estimating a hierarchical model on real data.
- **Covariates** are variables in a dataset that we typically take as given when analyzing real data. For instance, in the one-way ANOVA example, the group assignments of each observation is a covariate. In the HLM example, covariates would include the treatment indicators $Z_1, ..., Z_J$. In a more elaborate version of the HLM, they might also include variables such as student demographic information, measures of past academic performance, or school-level characteristics such as the school's geographic region or treatment assignment. When analyzing real data, we condition on these quantities, but when specifying a DGP, we will need to make choices about how they are distributed (e.g., we might specify that students' past academic performance is normally distributed).
- **Outcomes and endogenous quantities** are the variables whose distribution is described by the focal statistical model. In the one-way ANOVA example, the outcome variable consists of the measurements X_{ig} for $i = 1, ..., n_g$

and $g = 1, ..., G$. In the bivariate Poisson model, the outcomes consist of the component variables Z_1, Z_2, Z_3 and the observed counts C_1, C_2 because all of these quantities follow distributions that are specified as part of the focal model. The focal statistical model specifies the distribution of these variables, and we will be interested in estimating the parameters controlling their distribution.

Note that the focal statistical model only determines this third component of the DGP. The focal model consists of the equations describing what we would aim to estimate when analyzing real data. In contrast, the full statistical model also includes additional elements specify how to generate the structural features and covariates—the pieces that are taken as given when analyzing real data. Table 6.1 contrasts the role of structural features, covariates, and outcomes in real data analysis versus in simulations.

Table 6.1: Real Data Analysis versus Simulation

Component	Real world	Simulation world
Structural features	We obtain data of a given sample size, sizes of clusters, etc.	We specify sample sizes, we specify how to generate cluster sizes
Covariates	Data come with covariates	We specify how to generate covariates
Outcomes	Data come with outcome variables	We generate outcome data based on a focal model
Parameter estimation	We estimate a statistical model to learn about the unknown parameters	We estimate a statistical model and compare the results to the true parameters

For a given DGP, the full statistical model might involve distributions for structural features, distributions for covariates, and distributions for outcomes given the covariates. Each of these distributions will involve parameters that control the properties of the distribution (such as the average and degree of variation in a variable). We think of these parameters as falling into one of three categories: focal, auxiliary, or design.

- **Focal** parameters are the quantities that we care about and seek to estimate in real data analysis. These are typically parts of the focal statistical model, such as the population means $\mu_1, ..., \mu_G$ in the one-way ANOVA model, the correlation between counts ρ in the bivariate Poisson model, or the treatment effect γ_1 in the HLM example.
- **Auxiliary** parameters are the other quantities that go into the focal statistical model or some other part of the DGP, which we might not be substantively interested in when analyzing real data but which nonetheless affect

the analysis. For instance, in the one-way ANOVA model, we would consider the population variances $\sigma_1^2, ..., \sigma_G^2$ to be auxiliary if we are not interested in investigating how they vary from group to group. In the bivariate Poisson model we might consider the average counts μ_1 and μ_2 to be auxiliary parameters.

- **Design** parameters are the quantities that control how we generate structural features of the data. For instance, in a cluster-randomized trial, the fraction of schools assigned to treatment is a design parameter that can be directly controlled by the researchers. Additional design parameters might include the minimum and maximum number of students per school. Typically, in a real data analysis, we would not directly estimate such parameters because we take the distribution of structural features as given.

It is evident from this discussion that DGPs can involve *many* moving parts. One of the central challenges in specifying DGPs is that the performance of estimation methods will generally be affected by the *full* statistical model—including the design parameters and distribution of structural features and covariates—even though they are not part of the focal model.

6.3 A statistical model is a recipe for data generation

Once we have decided on a full statistical model and written it down in mathematical terms, we need to translate it into code. A function that implements a data-generating model should have the following form:

```
generate_data <- function(
  focal_parameters, auxiliary_parameters, design_parameters
) {

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

  return(sim_data)
}
```

The function takes a set of parameter values as input, simulates random numbers and does calculations, and produces as output a set of simulated data. Typically, the inputs will consist of multiple parameters, and these will include not only the focal model parameters, but also the auxiliary parameters, sample sizes, and other design parameters. The output will typically be a dataset, mimicking what one would see in an analysis of real data. In some cases, the output data might be augmented with some other latent quantities (normally unobserved in the real world) that can be used later to assess whether an estimation procedure produces results that are close to the truth.

We have already seen an example of a complete DGP function in the case study on one-way ANOVA (see Section 5.1). In this case study, we developed the following function to generate data for a single outcome from a set of G groups:

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

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

  group <- factor(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)
}
```

This function takes both the focal model parameters (`mu`, `sigma_sq`) and other design parameters that one might not think of as parameters per-se (`sample_size`). When simulating, we have to specify quantities that we take for granted when analyzing real data.

How would we write a DGP function for the bivariate Poisson model? The equations in Section 6.1 give us the recipe, so it just a matter of re-expressing them in code. For this model, the only design parameter is the sample size, N; the sole focal parameter is the correlation between the variates, ρ; and the auxiliary parameters are the expected counts μ_1 and μ_2. Our function should have all four of these quantities as inputs and should produce as output a dataset with two variables, C_1 and C_2. Here is one way to implement the model:

```
r_bivariate_Poisson <- function(N, mu1, mu2, rho = 0) {

  # covariance term, equal to E(Z_3)
  EZ3 <- rho * sqrt(mu1 * mu2)

  # Generate independent components
  Z1 <- rpois(N, lambda = mu1 - EZ3)
  Z2 <- rpois(N, lambda = mu2 - EZ3)
  Z3 <- rpois(N, lambda = EZ3)

  # Assemble components
  dat <- data.frame(
    C1 = Z1 + Z3,
```

```
    C2 = Z2 + Z3
  )

  return(dat)
}
```

Here we generate 5 observations from the bivariate Poisson with $\rho = 0.5$ and $\mu_1 = \mu_2 = 4$:

```
r_bivariate_Poisson(5, rho = 0.5, mu1 = 4, mu2 = 4)
```

```
  C1 C2
1  6  6
2  0  0
3  5  3
4  4  4
5  2  1
```

6.4 Plot the artificial data

The whole purpose of writing a DGP is to produce something that can be treated just as if it were real data. Considering that is our goal, we should act like it and engage in data analysis processes that we would apply whenever we analyze real data. In particular, it is worthwhile to create one or more plots of the data generated by a DGP, just as we would if we were exploring a new real dataset for the first time. This exercise can be very helpful for catching problems in the DGP function (about which more below). Beyond just debugging, constructing graphic visualizations can be a very effective way to *study* a model and strengthen your understanding of how to interpret its parameters.

In the one-way ANOVA example, it would be conventional to visualize the data with box plots or some other summary statistics for the data from each group. For exploratory graphics, we prefer plots that include representations of the raw data points, not just summary statistics. The figure below uses a density ridge-plot, filled in with points for each observation in each group. The plot is based on a simulated dataset with 50 observations in each of five groups.

Here is a plot of 30 observations from the bivariate Poisson distribution with means $\mu_1 = 10, \mu_2 = 7$ and correlation $\rho = .65$ (points are jittered slightly to avoid over-plotting):

Plots like these are useful for building intuitions about a model. For instance,

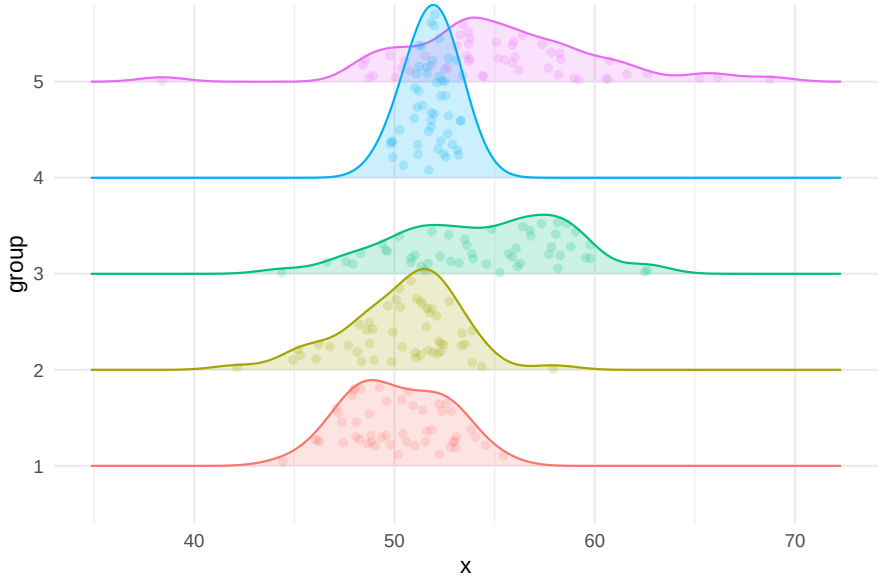


Figure 6.1: Densities of five heteroskedastic groups for the one-way ANOVA example.

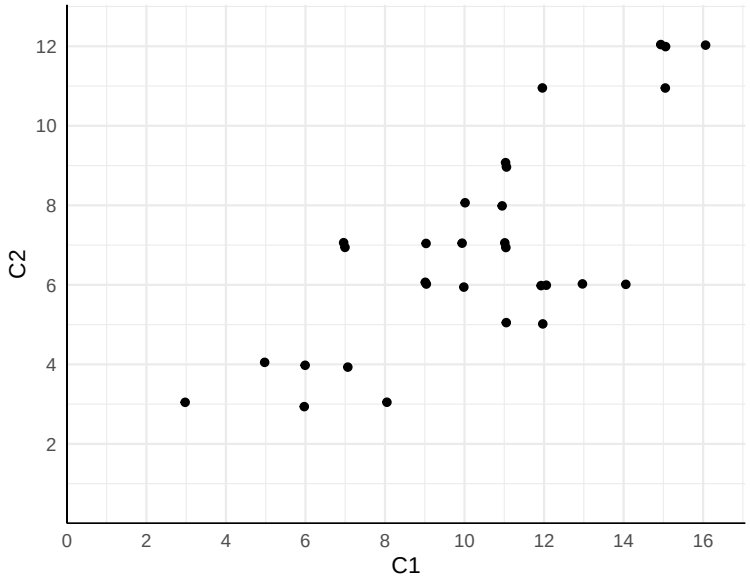


Figure 6.2: $N = 30$ observations from the bivariate Poisson distribution with $\mu_1 = 10, \mu_2 = 7, \rho = .65$.

we can inspect Figure 6.2 to get a sense of the order of magnitude and range of the observations, as well as the likelihood of obtaining multiple observations with identical counts. Depending on the analysis procedures we will apply to the dataset, we might even create plots of transformations of the dataset, such as a histogram of the differences $C_2 - C_1$ or a scatterplot of the rank transformations of C_1 and C_2.

6.5 Check the data-generating function

An important part of programming in R—especially when writing custom functions—is finding ways to test and check the correctness of your code. Just writing a data-generating function is not enough. It is also *critical* to test whether the output it produces is correct. How best to do this will depend on the particulars of the DGP being implemented.

For many DGPs, a broadly useful strategy is to generate a very large sample of data—one so large that the sample distribution should very closely resemble the population distribution. One can then test whether features of the sample distribution closely align with corresponding parameters of the population model.

For the heteroskedastic ANOVA problem, one basic thing we can do is check that the simulated data from each group follows a normal 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:

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

check_data <- generate_ANOVA_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)
  ) %>%
```

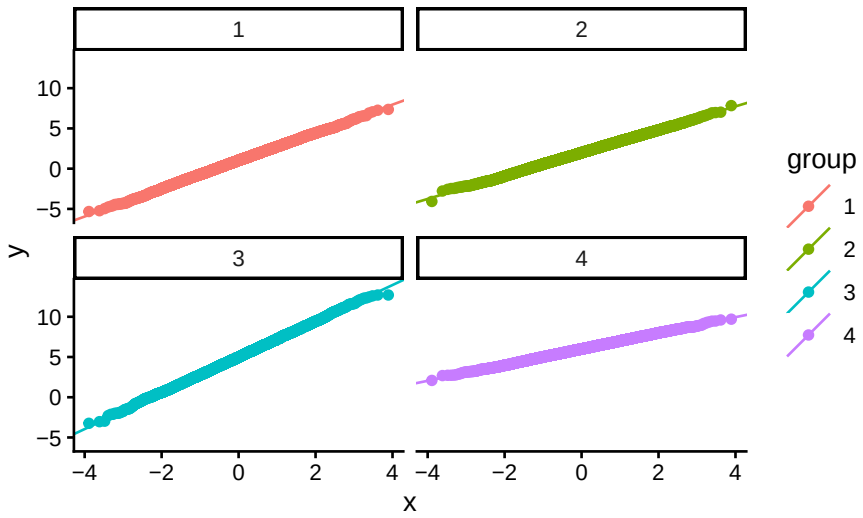
```
  mutate(mu = mu, sigma2 = sigma_sq) %>%
  relocate( group, n, mean, mu, var, sigma2 )
chk
```

```
# A tibble: 4 x 6
  group     n  mean    mu   var sigma2
  <fct> <int> <dbl> <dbl> <dbl>  <dbl>
1 1     10000  1.01     1 2.99       3
2 2     10000  1.98     2 2.04       2
3 3     10000  5.00     5 4.91       5
4 4     10000  5.99     6 0.996      1
```

It seems 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, color = group ) +
  facet_wrap( ~ group ) +
  stat_qq() + stat_qq_line()
```



This diagnostic looks good too. Here, these checks may seem a bit silly, but most bugs are silly—at least once you find them! In models that are even a little bit more complex, it is quite easy for small things such as a sign error to slip into your code. Even simple checks such as these can be quite helpful in catching such bugs.

6.6 Example: Simulating clustered data

Writing code for a complicated DGP can feel like a daunting task, but if you first focus on a recipe for how the data is generated, it is often not too bad to then convert that recipe into code. We now illustrate this process with a detailed case study involving a more complex data-generating process Recent literature on multisite trials (where, for example, students are randomized to treatment or control within each of a series of schools) has explored how variation in the strength of effects across schools can affect how different data-analysis procedures behave (e.g., Miratrix et al., 2021; Bloom et al., 2016). In this example, we are going to extend this work to explore best practices for estimating treatment effects in cluster randomized trials. In particular, we will investigate what happens when the treatment impact for each school is related to the size of the school.

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

In designing a simulation study, we need to find a DGP that will allow us to address the specific questions we are interested in investigating. For instance, in the one-way ANOVA example, we wanted to see how different degrees of within-group variation impacted the performance of several hypothesis-testing procedures. We therefore needed a data generation process that allowed us to control the extent of within-group variation.

To figure out what DGP to use for simulating data from a cluster-randomized trial, we need to consider how we are going to use those data in our simulation study. Because we are interested in understanding what happens when school-specific effects are related to school size, we will need data with the following features:

a) observations for students in each of several schools;
b) schools are different sizes and have different mean outcomes;
c) school-specific treatment effects correlate with school size; and
d) schools are assigned to different treatment conditions.

A given dataset will consist of observations for individual students in schools, with each student having a school id, a treatment assignment (shared for all in the school), and an outcome. A good starting point for building a DGP is to first sketch out what a simulated dataset should look like. For this example, we need data like the following:

schoolID	Z	size	studentID	Y
1	1	24	1	3.6
1	1	24	3	1.0
1	etc	etc	etc	etc
1	1	24	24	2.0
2	0	32	1	0.5
2	0	32	2	1.5
2	0	32	3	1.2
etc	etc	etc	etc	etc

When running simulations, it is good practice to look at simple scenarios along with complex ones. This lets us not only identify conditions where some aspect of the DGP is important, but also verify that the feature does *not* matter under scenarios where we know it should not. Given this principle, we land on the following points:

- We need a DGP that lets us generate schools that are all the same size or that are all different sizes.
- Our DGP should allow for variation in the school-specific treatment effects.
- We should have the option to generate school-specific effects that are related or unrelated to school size.

6.6.2 A model for a cluster RCT

DGPs are expressed and communicated using mathematical models. In developing a DGP, we often start by considering the model for the outcomes (along with its focal parameters), which covers some but not all of the steps in the full recipe for generating data. It is helpful to write down the equations for the outcome model and then note what further quantities need to be generated (such as structural features and covariates). Then we can consider how to generate these quantities with auxiliary models.

Section 6.1.3 introduced a basic multilevel model for a cluster-randomized trial. This model had two parts, starting with a model for our student outcome:

$$Y_{ij} = \beta_{0j} + \epsilon_{ij} \text{ with } \epsilon_{ij} \sim N(0, \sigma^2_\epsilon)$$

where Y_{ij} is the outcome for student i in school j, β_{0j} is the average outcome in school j, and ϵ_{ij} is the residual error for student i in school j. The model was completed by specifying how the school-level outcomes vary as a function of treatment assignment:

$$\beta_{0j} = \gamma_0 + \gamma_1 Z_j + u_j, \quad u_j \sim N(0, \sigma^2_u).$$

This model has a constant treatment effect: if a school is assigned to treatment, then the outcomes of all students in the school are raised by γ_1. But we also

want to allow the size of impact to vary by school size. This suggests we will need to elaborate the model to include a treatment-by-size interaction term.

One approach for allowing the school-specific impacts to depend on school size is to introduce school size as a predictor, as in

$$\beta_{0j} = \gamma_0 + \gamma_1 Z_j + \gamma_2 \left(Z_j \times n_j\right) + u_j.$$

A drawback of this approach is that changing the average size of the schools will change the average treatment impact. A more interpretable approach is to allow treatment effects to depend on the *relative* school sizes. To do this, we can define a covariate that describes the deviation in the school size relative to the average size. Thus, let

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

where $\bar{n}$ is the overall average school size. Using this covariate, we then revise our equation for our school j to:

$$\beta_{0j} = \gamma_0 + \gamma_1 Z_j + \gamma_2 \left(Z_j \times S_j\right) + u_j.$$

If γ_2 is positive, then bigger schools will have larger treatment impacts. Because S_j is centered at 0, the overall average impact across schools will be simply γ_1. (If S_j was not centered at zero, then the overall average impact would be some function of γ_1 and γ_2.)

Putting all of the above together, we now have an HLM to describe the distribution of outcomes conditional on the covariates and structural features:

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

Substituting the second equation into the first leads to a single equation for generating the student-level outcomes (or what is called the reduced form of the HLM):

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

The parameters of this focal model are the mean outcome among control schools (γ_0), the average treatment impact (γ_1), the school-size by treatment interaction term (γ_2), the amount of school-level variation (σ_u^2), and the amount of within-school variation (σ_ϵ^2).

There are several ways that we could elaborate this model further. For one, we might want to include a main effect for S_j, so that average outcomes in the absence of treatment are also dependent on school size. For another, we might revise the model to allow for school-to-school variation in treatment impacts that is not explained by school size. For simplicity, we do not build in these further features, but see the exercises at the end of the chapter.

So far we have a mathematical model analogous to what we would write if we were *analyzing* the data. To *generate* data, we also need a way to generate the structural features and covariates involved in the model. First, we need to know the number of clusters (J) and the sizes of the clusters (n_j, for $j = 1, ..., J$). For illustrative purposes, we will generate size sizes from a uniform distribution with average school size $\bar{n}$ and a fixed parameter α that controls the degree of variation in school size. Mathematically,

$$n_j \sim \text{Unif}\left[(1-\alpha)\bar{n}, (1+\alpha)\bar{n}\right].$$

Equivalently, we could generate school sizes by taking

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

For instance, if $\bar{n} = 100$ and $\alpha = 0.25$ then schools would range in size from 75 to 125. This specification is nice because it is simple, with just two parameters, both of which are easy to interpret: $\bar{n}$ is the average school size and α is the degree of variation in school size.

To round out the model, we also need to define how to generate the treatment indicator, Z_j. To allow for different treatment allocations, we will specify a proportion p of clusters assigned to treatment. Because we are simulating a cluster-randomized trial, we do this by drawing a simple random sample (without replacement) of $p \times J$ schools out of the total sample of J schools, then setting $Z_j = 1$ for these schools and $Z_j = 0$ for the remaining schools. We will denote this process as $Z_1, ..., Z_J \sim SRS(p, J)$, where SRS stands for simple random sample.

Now that we have an auxiliary model for school sizes, let us look again at our treatment impact heterogeneity term:

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

where $U_j \sim \text{Unif}(-1, 1)$ is the uniform variable used to generate n_j. Because we have standardized by average school size, the importance of the covariate does not change as a function of average school size, but rather as a function of the relative variation parameter α. Setting up a DGP with standardized quantities will make it easier to interpret simulation results, especially if we are looking at results from multiple scenarios with different parameter values. To the extent feasible, we want the parameters of the DGP to change only one feature of the data, so that it is easier to isolate the influence of each parameter.

6.6.3 From equations to code

When sketching out the equations for the DGP, we worked from the lowest level of the model (the students) to the higher level (schools) and then to the

auxiliary models for covariates and structural features. For writing code to based on the DGP, we will proceed in the opposite direction, from auxiliary to focal and from the highest level to the lowest. First, we will generate the schools and their features:

- Generate school sizes
- Generate school-level covariates
- Generate school-level random effects

Then we will generate the students inside the schools:

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

The mathematical model gives us the details we need to execute with each of these steps.

Here is the skeleton of a DGP function with arguments for each of the parameters we might want to control, including defaults for each (see Section A.2 for more on function defaults):

```
gen_cluster_RCT <- function(
    J = 30,
    n_bar = 10,
    alpha = 0,
    p = 0.5,
    gamma_0 = 0, gamma_1 = 0, gamma_2 = 0,
    sigma2_u = 0, sigma2_e = 1
) {

  # generate schools sizes
  # Code (see below) goes here
}
```

Note that the inputs to this function are a mix of *model parameters* (`gamma_0`, `gamma_1`, `gamma_2`, representing coefficients in regressions), *auxilary parameters* (`sigma2_u`, `sigma2_e`, `alpha`, `n_bar`), and *design parameters* (`J`, `p`) that directly inform data generation. We set default arguments (e.g., `gamma_0=0`) so that we can ignore aspects of the DGP that we do not care about for the moment.

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

We first make the schools:

```
  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 for all schools
# (the random effects)
u0j <- rnorm( J, mean = 0, sd = sqrt(sigma2_u) )

# randomize schools (proportion p to treatment)
Zj <- ifelse( sample( 1:J ) <= J * p, 1, 0)

# Calculate schools intercepts
S_j <- (nj - n_bar) / n_bar
beta_0j <- gamma_0 + gamma_1 * Zj + gamma_2 * Zj * S_j + u0j
```

The code is a literal translation of the math we did before. Note the line with `sample(1:J) <= J*p`; this is a simple trick to generate a 0/1 indicator for control and treatment conditions.

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.

Next, we use the school characteristics to generate the individual-level variables:

```
# Make individual site membership
sid <- as.factor( rep( 1:J, nj ) )

# Generate the individual-level errors and outcome
N <- sum( nj )
e <- rnorm( N, mean = 0, sd = sqrt(sigma2_e) )
Y <- beta_0j[sid] + e

# Bundle into a dataset
dd <- data.frame(
  sid = sid,
  Z = Zj[ sid ],
  Yobs = Y
)
```

A key piece here is the `rep()` function that takes a list and repeats each element of the list a specified number of times. In particular, `rep()` repeats each number $(1, 2, \ldots, J)$, the corresponding number of times as listed in `nj`. Putting the code above into the function skeleton will produce a complete DGP function (view the complete function here). We can then call the function as so:

```
dat <- gen_cluster_RCT(
  J=3, n_bar = 5, alpha = 0.5, p = 0.5,
  gamma_0 = 0, gamma_1 = 0.2, gamma_2 = 0.2,
```

```
  sigma2_u = 0.4, sigma2_e = 1
)

head( dat )
```

```
  sid Z        Yobs
1   1 1 -0.18235214
2   1 1 -0.50136685
3   1 1  1.10960242
4   2 0  0.07182754
5   2 0  0.83878390
6   3 0  1.06898694
```

With this function, we can control the average size of the clusters (`n`), the number of clusters (`J`), the proportion treated (`p`), the average outcome in the control group (`gamma_0`), the average treatment effect (`gamma_1`), the school size by treatment interaction (`gamma_2`), the amount of cross site variation (`sigma2_u`), the residual variation (`sigma2_e`), and the amount of school size variation (`alpha`). The next step is to test the code, making sure it is doing what we think it is. In fact, it is not–there is a subtle bug that only appears under some specifications of the parameters; see the exercises for more on diagnosing and repairing this error.

6.6.4 Standardization in the DGP

One difficulty with the current implementation of the model is that the magnitude of the different parameters are inter-connected. For instance, raising or lowering the within-school variance (σ_u^2) will increase the overall variation in Y, and therefore affect our the interpretation of the treatment effect parameters, because a given value of γ_1 will be less consequential if there is more overall variation. We can fix this issue by standardizing the model parameters. Standardization will allow us to reduce the set of parameters we might want to manipulate and will ensure that varying the remaining parameters only affects one aspect of the DGP.

For a continuous, normally distributed outcome variable, a common approach to scaling is to constrain the overall variance of the outcome to a fixed value, such as 1 or 100. The magnitude of the other parameters of the model can then be interpreted relative to this scale. Often, we can also constrain the mean of the outcome to a fixed value, such as setting $\gamma_0 = 0$ without affecting the interpretation of the other parameters.

With the current model, the variance of the outcome across students in the

control condition is

$$\begin{aligned}\text{Var}(Y_{ij}|Z_j = 0) &= \text{Var}(\beta_{0j} + \epsilon_{ij}|Z_j = 0) \\ &= \text{Var}(\gamma_0 + \gamma_1 Z_j + \gamma_2 Z_j S_j + u_j + \epsilon_{ij}|Z_j = 0) \\ &= \text{Var}(\gamma_0 + u_j + \epsilon_{ij}) \\ &= \sigma_u^2 + \sigma_\epsilon^2.\end{aligned}$$

To ensure that the total variance is held constant, we can redefine the variance parameters in terms of the intra-class correlation (ICC). The ICC is defined as

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

The ICC measures the degree of between-group variation as a proportion of the total variation of the outcome. It plays an important role in power calculations for cluster-randomized trials. If we want the total variance of the outcome to be 1, we need to set $\sigma_u^2 + \sigma_\epsilon^2 = 1$, which the implies that $ICC = \sigma_u^2$, and $\sigma_\epsilon^2 = 1 - ICC$. Thus, we can call our DGP function as follows:

```
ICC <- 0.3
dat <- gen_cluster_RCT(
  J = 30, n_bar = 20, alpha = 0.5, p = 0.5,
  gamma_0 = 0, gamma_1 = 0.3, gamma_2 = 0.2,
  sigma2_u = ICC, sigma2_e = 1 - ICC
)
```

Manipulating the ICC rather than separately manipulating σ_u^2 and σ_ϵ^2 will let us change the degree of between-group variation without affecting the overall scale of the outcome.

A further consequence of setting the overall scale of the outcome to 1 is that the parameters controlling the treatment impact can now be interpreted as standardized mean difference effect sizes. The standardized mean difference for a treatment impact is defined as the average impact over the standard deviation of the outcome among control observations.[1] Letting δ denote the standardized mean difference parameter,

$$\delta = \frac{E(Y|Z_j = 1) - E(Y|Z_j = 0)}{SD(Y|Z_j = 0)} = \frac{\gamma_1}{\sqrt{\sigma_u^2 + \sigma_\epsilon^2}}$$

Because we have constrained the total variance to 1, γ_1 is equivalent to δ. This equivalence holds for any value of γ_0, so we do not have to worry about manipulating γ_0 in the simulations—we can simply leave it at its default value. The γ_2 parameter only influences outcomes for those in the treatment

[1] An alternative definition is based on 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.

condition, with larger values inducing more variation. Standardizing based on the total variance also makes γ_2 somewhat easier to interpret because this parameter the same units as γ_1. Specifically, γ_2 is the expected difference in the standardized treatment impact for schools that differ in size by $\bar{n}$ (the overall average school size). More broadly, standardizing by a stable parameter (the variation among those in the control condition) rather than something that changes in reaction to other parameters (which would be the case if we standardized by overall variation or pooled variation) will lead to more interpretable quantities.

6.7 Sometimes a DGP is all you need

We have introduced the data-generating process as only the first step in developing a simulation study. Indeed, there are many more considerations to come, which we will describe in subsequent chapters. However, this first step is still very useful in its own right, even apart from the other components of a simulation. Sometimes, a data-generating function is all you need to learn about a statistical model.

A data-generating function provides a very effective way to *study* a statistical model, such as a model that you might be learning about in a course or through self-study. Writing code based on a model is a much more *active* process than listening to a lecture or reading a book or paper. Coding requires you to make the mathematical notation tangible and can help you to notice details of the model that might be easily missed through listening or reading alone.

Suppose we are taking a first course psychometrics and have just been introduced to item response theory (IRT) models for binary response items. Our instructor has just laid out a bunch of notation:

- We have data from a sample of N individuals, each of whom responds to a set of M test items.
- We let $X_{im} = 1$ if respondent i answers item m correctly, with $X_{im} = 0$ otherwise, for $i = 1, ..., N$ and $m = 1, ..., M$.
- We imagine that each respondent has a latent ability θ_i on whatever domain the test measures.

Our instructor next puts up a slide showing the equation for what they say is a three-parameter IRT model:

$$Pr(X_{im} = 1) = \gamma_m + (1 - \gamma_m) g\left(\alpha_m[\theta_i - \beta_m]\right),$$

where $g(x)$ is the cumulative logistic curve: $g(x) = e^x/(1+e^x)$. The instructor explains that α_m is a positive-valued "discrimination parameter", β_m is a difficulty parameter that can be any value, and γ_m is a guessing probability

parameter between 0 and 1. They further explain that the guessing parameter is often hard to estimate and so might get treated as fixed and known, based on the number of response options on the item. For instance, if all the items have four options, then we might take $\gamma_m = \frac{1}{4}$ for $m = 1, ..., M$. Finally, they explain that the ability parameters are assumed to follow a standard normal distribution, so $\theta_i \sim N(0, 1)$ in the population.

What is this madness? It certainly is a lot of notation to follow. To make sense of all the moving pieces, let's try simulating from the model using more-or-less arbitrary parameters. To begin, we will need to pick a test length M and a sample size N. Let's use $N = 7$ participants and $M = 4$ items for starters:

```
N <- 7
M <- 4
```

DGPs usually follow a hierarchy, with different elements being defined in terms of other elements, and so on until some elements do not depend on anything. Following this structure, we first generate the ability parameters because they follow a standard normal distribution that does not depend on anything else:

```
thetas <- rnorm(N)
```

Now we need sets of parameters $\alpha_m, \beta_m, \gamma_m$ for every item $m = 1, ..., M$. Where do we get these?

For a particular fixed-length test, the set of item parameters would depend on the features of the actual test questions. But we are not (yet) dealing with actual testing data, so we will need to make up an auxiliary model for these parameters. Perhaps we could just simulate some values? Each item requires three numbers; the easiest way to generate them is to let them all be independent of each other, so we do that.

- **Difficulty:** Arbitrarily, let's draw the difficulty parameters from a normal distribution with mean $\mu_\beta = 0$ and standard deviation $\tau_\beta = 1$.
- **Discrimination:** The discrimination parameters have to be greater than zero, and values near $\alpha_m = 1$ make the model simplify (in other words, if $\alpha_1 = 1$ then we can drop the parameter from the model), so let's draw them from a gamma distribution with mean $\mu_\alpha = 1$ and standard deviation $\tau_\alpha = 0.2$. In looking up `rgamma()` we see that it takes shape and rate, so we need to convert our mean and standard deviation before we can generate values. A bit of poking on Wikipedia gives us our transform: shape is equal to $\mu_\alpha^2 \tau_\alpha^2 = 0.2^2$ and rate is equal to $\mu_\alpha \tau_\alpha^2 = 0.2^2$.
- **Guessing:** Finally, we imagine that all the test questions have four possible responses, and therefore set $\gamma_m = \frac{1}{4}$ for all the items, just like the instructor suggested.

Here is the code following from this line of reasoning:

```
betas <- rnorm(M, mean = 0, sd = 1.5)             # difficulty
alphas <- rgamma(M, shape = 0.2^2, rate = 0.2^2) # discrimination
gammas <- rep(1 / 4, M)                           # guessing
```

A three-parameter IRT model describes the probability that a given respondent with ability θ_i answers item m correctly. To generate data based on the model, we need to produce scores on every item for every respondent, leading to a matrix of N respondents by M items. To simplify this calculation, we write a function to compute the item probabilities for a single respondent:

```
r_scores <- function(theta_i, alphas, betas, gammas, M) {
  pi_i <- gammas + (1 - gammas) * plogis(alphas * (theta_i - betas))
  scores <- rbinom(M, size = 1, prob = pi_i)
  names(scores) <- paste0("q",1:M)
  return(scores)
}

r_scores( thetas[1],
          alphas = alphas, betas = betas,
          gammas = gammas, M = M )
```

```
q1 q2 q3 q4
 0  1  0  1
```

The first line of the function calculates the probability of correctly answering all M items, and stores the result in a vector `pi_i`. This code is simply the math turned into R syntax, nothing more, nothing less. In the next line, we simulate binary outcomes by flipping coins with the specified vector of probabilities. Finally, we assign names to each item so that we can keep track of which score is for which item. Now, we use `map()` to run the function on each respondent:

```
map_dfr(
  thetas, .f = r_scores,
  alphas = alphas, betas = betas, gammas = gammas, M = M
)
```

```
# A tibble: 7 x 4
     q1    q2    q3    q4
  <int> <int> <int> <int>
1     1     0     1     1
2     1     0     1     1
3     0     0     0     1
4     0     0     0     1
5     1     1     0     0
6     0     0     0     1
7     1     0     1     1
```

To make it easier to generate datasets with different characteristics, we bundle the above code chunks into a single data-generating function. To do so, we have to decide what the input parameters should be. We know we need to at least specify N and M. We made assumptions about the item parameters, but we had to specify means and standard deviations for the difficulty and discrimination parameter distributions, so it is natural to allow those to be inputs also. Our data-generating function will then be

```
r_3PL_IRT <- function(
    N, M = 4,
    diff_M = 0, diff_SD = 1,
    disc_M = 1, disc_SD = 0.2,
    item_options = 4
) {
  # generate ability parameters
  thetas <- rnorm(N)

  # generate item parameters

  alphas <- rgamma(
    M,
    shape = disc_M^2 * disc_SD^2,
    rate = disc_M * disc_SD^2
  )
  betas <- rnorm(M, mean = diff_M, sd = diff_SD)
  gammas <- rep(1 / item_options, M)

  # simulate item responses
  test_scores <- map_dfr(
   thetas, .f = r_scores,
    alphas = alphas, betas = betas, gammas = gammas, M = M
  )

  # calculate total score
  test_scores$total <- rowSums(test_scores)

  return(test_scores)
}

r_3PL_IRT(N = 7, M = 4)
```

```
# A tibble: 7 x 5
     q1    q2    q3    q4 total
  <int> <int> <int> <int> <dbl>
1     0     0     0     0     0
2     0     1     0     1     2
```

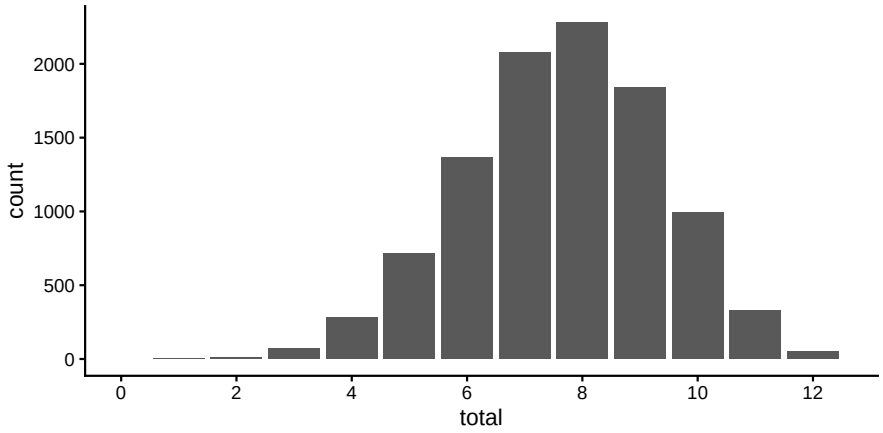
```
3     1     1     1     1     4
4     0     1     1     1     3
5     1     0     1     1     3
6     0     0     1     1     2
7     0     0     0     1     1
```

We have a DGP function! We next use it to understand our model. To get a sense of variation across items and participants, we generate a dataset with many, many participants taking a longer test with $M = 12$ items:

```
test_scores <- r_3PL_IRT(N = 10000, M = 12)
```

Having written a function for the 3-parameter logistic IRT model makes it easy to explore properties of the model. For instance, we can easily visualize the distribution of the total scores on the test:

```
ggplot(test_scores, aes(total)) +
  geom_bar() +
  scale_x_continuous(limits = c(0,12.5), breaks = seq(0,12,2))
```



Looking at item variation, we can examine the probability of correctly responding to each of the items by computing sample means for each item:

```
test_scores %>%
  summarise(across(starts_with("q"), mean))
```

```
# A tibble: 1 x 12
     q1    q2    q3    q4    q5    q6    q7    q8    q9   q10
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 0.622 0.618 0.620 0.619 0.761 0.626 0.624 0.618 0.622 0.627
# i 2 more variables: q11 <dbl>, q12 <dbl>
```

The percentage of correct responses varies from 61.8% to 76.1. What are the

correlations between individual items and the total score? Let's check:

```
test_scores %>%
  summarise(across(starts_with("q"), ~ cor(.x, total)))
```

```
# A tibble: 1 x 12
     q1    q2    q3    q4    q5    q6    q7    q8    q9   q10
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 0.261 0.294 0.284 0.285 0.358 0.304 0.278 0.274 0.279 0.284
# i 2 more variables: q11 <dbl>, q12 <dbl>
```

Note that simulating a new dataset will produce different results for these summaries because generating each dataset entails sampling a new set of items (with different difficulty and discrimination).

Writing a DGP function makes it possible to study a model through exploration, simply by examining datasets generated by wiggling the model parameters up or down. For instance, we could look at what happens to the distribution of total scores if the items covered a much broader range of difficulties (higher τ_α) or are mis-calibrated to be too difficult ($\mu_\tau > 0$). What if the items had varying numbers of response options, so the guessing parameters differ? Are there item parameter values that will produce highly unrealistic distributions of total scores? Exercises 6.11, 6.12, and 6.13 ask you to further explore these questions.

Overall, implementing the model as a data-generating function has allowed us to explore the model in a more active way than simply reading about it or listening to a lecture. We have a more visceral feel about how the different parameters would create variation in our data—and it would be easy to tweak those parameters to see how things changed to learn even more.

6.8 More to explore

This chapter has introduced the core components of a data-generating process and demonstrated how to move from formulating a full data-generating model to implementing the model in R code. Our main aim has been to provide enough scaffolding for you to get started with simulating from and exploring a variety of models. The exercises below will give you further practice in developing, testing, and extending DGP functions. Of course, we have not covered every consideration and challenge that arises when writing DGPs for simulations—there is *much* more to explore!

We will cover some further challenges in subsequent chapters. See, for example, Chapters 19 and 20. Yet more challenges will surely arise as you apply simulations in your own work. Even though such challenges might not align

with the examples or contexts that we have covered here, the concepts and principles that we have introduced should allow you to reason about potential solutions. In doing so, we encourage you adopt the attitude of a chef in a test kitchen by trying out your ideas with code. In the kitchen, there is often no way to know how a dish will turn out until you actually bake it and taste it. Likewise, in developing a DGP, the best way to understand and evaluate a DGP is to write it out in code and explore the datasets that you can produce with it.

Exercises

Exercise 6.1 (The Welch test on a 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. Because the t-distribution has variance of $\nu/(\nu-2)$), we add the scale factor of $(\nu-2)/\nu$ to achieve the target standard deviation specified in the `sd` input parameter.

```
r_tss <- function(n, mean, sd, df) {
  stopifnot( df > 2 ) # 2 or less has no defined variance
  mean + sd * sqrt( (df-2)/df ) * rt(n = n, df = df)
}

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

```
[1] 3.165957 4.476384 5.693423 4.697553 1.901471 3.050113
[7] 4.110213 3.269402
```

1. Modify the Welch simulation's `generate_ANOVA_data()` function from Section 5.1.1 to generate data from shifted-and-scaled t-distributions rather than from normal distributions. Include the degrees of freedom as an input argument. Note that, in your modified function, you will have to translate the original parameters (`mu`, `sigma_sq`) to appropriate parameters of `r_tss()`. With new function in hand, simulate a dataset with low degrees of freedom and plot it to see if the data includes outliers.

2. Now generate more data and calculate the means and standard deviations to see if they are correctly calibrated (i.e., generate a big dataset to ensure you get reliable mean and standard deviation estimates). Check `df` equal to 500, 5, 3, and 2.

3. Once you are satisfied that you have a correct DGP function, re-run the Type-I error rate calculations from Exercise 5.1 and Exercise 5.2 using a t-distribution with 5 degrees of freedom. Do the results change substantially?

Exercise 6.2 (Plot the bivariate Poisson)**.** In Section 6.3, we provided an example of a DGP function for the bivariate Poisson model. We demonstrated a plot of data simulated from this function in Section 6.4. Create a similar plot but for a much larger sample size of $N = 1000$.

With such a large dataset, it will likely be hard to distinguish individual observations because of over-plotting. Create a better visual representation of the same simulated dataset, such as a heatmap or a contour plot.

Exercise 6.3 (Check the bivariate Poisson function)**.** Although we presented a DGP function for the bivariate Poisson model, we have not demonstrated how to check that the function is correct—we've left that to you. Write some code to verify that the function `r_bivariate_Poisson()` works properly. Do this by generating a very large sample (say $N = 10^4$ or 10^5) and verifying the following:

- The sample means of C_1 and C_2 align with the specified population means.
- The sample variances of C_1 and C_2 are close to the specified population means (because for a Poisson distribution $\mathbb{E}(C_p) = \mathbb{V}(C_p)$ for $p = 1, 2$).
- The sample correlation aligns with the specified population correlation.
- The observed counts C_1 and C_2 follow Poisson distributions.

Exercise 6.4 (Add error-catching to the bivariate Poisson function)**.** In Section 6.1, we noted that the bivariate Poisson function as we described it can only produce a constrained range of correlations, which a maximum value that depends on the ratio of μ_1 to μ_2. Our current implementation of the model does not handle this aspect of the model very well:

```
r_bivariate_Poisson(5, rho = 0.6, mu1 = 4, mu2 = 12)
```

```
  C1 C2
1 NA  5
2 NA 12
3 NA 13
4 NA 18
5 NA  7
```

For this combination of parameter values, $\rho \times \sqrt{\mu_1 \mu_2}$ is larger than μ_1, which leads to simulated values for C_1 that are all missing. That makes it pretty hard to compute the correlation between C_1 and C_2.

Please help us fix this issue! Revise `r_bivariate_Poisson()` so that it checks for allowable values of ρ. If the user specifies a combination of parameters that does not make sense, make the function throw an error (using R's `stop()` function).

Exercise 6.5 (A bivariate negative binomial distribution)**.** One potential limitation of the bivariate Poisson distribution described above is that the variances of the counts are necessarily equal to the means. This limitation is inherited from the univariate Poisson distributions that each variate follows (for a Poisson distribution the mean and the variance are always the same).

One way to relax this limitation is to use distributions with marginals that are negative binomial rather than Poisson, thereby allowing for overdispersion. Cho et al. (2023) provides a method for constructing such a bivariate negative binomial distribution with latent, gamma-distributed components. Their algorithm involves first generating components from gamma distributions with specified shape and scale parameters:

$$\begin{aligned} Z_0 &\sim \Gamma(\alpha_0, \beta) \\ Z_1 &\sim \Gamma(\alpha_1, \beta) \\ Z_2 &\sim \Gamma(\alpha_2, \beta) \end{aligned}$$

for $\alpha_0, \alpha_1, \alpha_2 > 0$ and $\beta > 0$. They then simulate independent Poisson random variables as

$$\begin{aligned} C_1 &\sim Pois(Z_0 + Z_1) \\ C_2 &\sim Pois(\delta(Z_0 + Z_2)). \end{aligned}$$

The resulting count variables follow marginal negative binomial distributions with moments

$$\begin{aligned} \mathbb{E}(C_1) &= (\alpha_0 + \alpha_1)\beta & \mathbb{V}(C_1) &= (\alpha_0 + \alpha_1)\beta(\beta + 1) \\ \mathbb{E}(C_2) &= (\alpha_0 + \alpha_2)\beta\delta & \mathbb{V}(C_2) &= (\alpha_0 + \alpha_2)\beta\delta(\beta\delta + 1) \\ & & \mathrm{Cov}(C_1, C_2) &= \alpha_0\beta^2\delta. \end{aligned}$$

The correlation between C_1 and C_2 is thus

$$\mathrm{cor}(C_1, C_2) = \frac{\alpha_0}{\sqrt{(\alpha_0 + \alpha_1)(\alpha_0 + \alpha_2)}} \frac{\beta\sqrt{\delta}}{\sqrt{(\beta + 1)(\beta\delta + 1)}}.$$

1. Write a DGP function that implements this distribution given the α_0, α_1, α_2, β, and δ parameters. It should have the following form:

```
r_bivariate_Poisson_raw <- function(N, alpha0, alpha1, alpha2, beta, delta)
```

2. Write some code to check that the function produces data where each variate follows a negative binomial distribution and the means, variances, and correlation are as specified by the formula above. Check a few different values.

3. Now write a more user-friendly DGP function of the following form:

```
r_bivariate_Poisson_NB <- function(N, mu1, mu2, var1, var2, rho = 0)
```

where `mu1`, `mu2`, `var1`, and `var2` are the means and variances of the two variates, and `rho` is the correlation between them. You will have to do some math to convert the target parameters to the given α_0, α_1, α_2, β, and δ. Your new function could simply call the first one or it could be a separate, stand-alone function.

4. Write some test code to verify that the means, variances, and correlations of data generated from the function match the parameters specified.

5. Consider target means of $\mathbb{E}(C_1) = \mathbb{E}(C_2) = 10$ and target variances of $\mathbb{V}(C_1) = \mathbb{V}(C_2) = 15$. What are the minimum and maximum possible correlations between C_1 and C_2?

Exercise 6.6 (Another bivariate negative binomial distribution)**.** Another model for generating bivariate counts with negative binomial marginal distributions is by using Gaussian copulas. Here is a mathematical recipe for this distribution, which will produce counts with marginal means μ_1 and μ_2 and marginal variances $\mu_1 + \mu_1^2/p_1$ and $\mu_2 + \mu_2^2/p_2$. Start by generating variates from a bivariate standard normal distribution with correlation ρ:

$$\begin{pmatrix} Z_1 \\ Z_2 \end{pmatrix} \sim N\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}\right)$$

You can do this via the `rmvnorm()` function in the `mvtnorm` package. Now find $U_1 = \Phi(Z_1)$ and $U_2 = \Phi(Z_2)$, where $\Phi()$ is the standard normal cumulative distribution function (called `pnorm()` in R). Then generate the counts by evaluating U_1 and U_2 with the negative binomial quantile function, $F_{NB}^{-1}(x|\mu, p)$ with mean parameters μ and size parameter p (this function is called `qnbinom()` in R):

$$C_1 = F_{NB}^{-1}(U_1|\mu_1, p_1) \qquad C_2 = F_{NB}^{-1}(U_2|\mu_2, p_2).$$

The resulting counts will be correlated, but the correlation will not be equal to ρ.

1. Write a DGP function that implements this distribution. It should have the following form:

```
r_NB_copula <- function(N, mu1, mu2, var1, var2, rho = 0)
```

You will need to convert the target means and variances to the appropriate parameters for your `rnbinom()` call. Use $p = \mu^2/(\text{var} - \mu)$ to convert from mean and variance to the size parameter.

2. Write some code to check that the function produces data where each variate follows a negative binomial distribution.

3. Use your new function to create a graph showing the population correlation between the observed counts as a function of the input ρ. Following part 5 of Exercise 6.5, use $\mu_1 = \mu_2 = 10$ and $\text{var}_1 = \text{var}_2 = 15$. How does the range of achievable correlations compare to the range found in that problem?

Exercise 6.7 (Plot the data from a cluster-randomized trial)**.** Run `gen_cluster_RCT()` with parameter values of your choice to produce data from a simulated cluster-randomized trial. Create a plot of the data that illustrates how student-level observations are nested within schools and how schools are assigned to different treatment conditions.

Exercise 6.8 (Checking the Cluster RCT DGP)**.** In the following we run a few tests on the cluster RCT DGP code (`gen_cluster_RCT()`) to see if it works as advertised.

1. What is the variance of the outcomes generated by the model for the cluster-randomized trial if there are no treatment effects? (Try simulating data to check!) What other quick checks can you run on this DGP to make sure it is working correctly?

2. It is prudent to prevent nonsensical values of parameters. For example, what happens if you put in `alpha = 1` or give an `alpha` larger than 1? Does it generate data? It can be useful to put in some error-catching code to prevent such values from being used. For example, you can have `stopifnot( n_min > 0 )` at the top of your function. Implement this error-catching code and check that it works.

3. The code also rounds `alpha * n_bar`–but what happens if `alpha * n_bar` is not an integer? What will happen if you specify values for `alpha` and `n_bar` that do not multiply to a whole number? Is this ok, or could it cause problems? Explain why or why not.

4. In the `gen_cluster_RCT()` function, the following line of code generates the number of individuals per school:

```
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.

Exercise 6.9 (More school-level variation)**.** The DGP for the cluster-randomized trial allows for school-level treatment impact variation, but only to the extent that the variation is explained by school size. Change the `gen_cluster_RCT()` function to allow for school-level treatment impact variation that is not related to school size. Generate some data to show how it works. (If you did the prior problem, work from your corrected function.)

Exercise 6.10 (Cluster-randomized trial with baseline predictors)**.** Extend the DGP for the cluster-randomized trial to include an individual-level covariate X that is correlated with the outcome. Do this by modifying the model for student-level outcomes as

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

Keep the same β_1 for all schools. To implement this model as a DGP, you will have to decide how to generate X_{ij}.

1. Use words and equations to explain your auxiliary model for X_{ij}.

2. Implement the model by modifying `gen_cluster_RCT()` accordingly.

3. Use your implementation to find the unconditional variance of the outcome, $\text{Var}(Y|Z_j = 0)$, when $\beta_1 = 0.6$.

Exercise 6.11 (3-parameter IRT datasets)**.** A challenge that arises with the IRT model described in Section 6.7 is that the data are generated using *random parameters* (the person ability parameters $\theta_1, ..., \theta_N$ and item characteristics $\alpha_m, \beta_m, \gamma_m$ for $m = 1, ..., M$) that we simulated from auxiliary models. When applying IRT models to actual test data, an analyst will usually need to estimate at least some of these parameters. In particular, they might be fitting the model to *score* the student responses by estimating their latent ability parameters or fitting a model to *calibrate* the items by estimating the item characteristics. But each time we generate a new dataset with `r_3PL_IRT()`, those latent parameters will change.

In order to understand how well an estimation procedure performs on data generated by our function, we will need to keep track of these randomly generated parameter values, even though those values will not be known in real

data analysis contexts. Suppose, for example, that our main interest is in understanding how well we can recover the item characteristics.

1. Modify `r_3PL_IRT()` to return a named list containing two datasets (use R's `list()` function). The first entry in the list should be a dataset with the person-by-item responses, just as in the original function. The second entry in the list should be an $M \times 3$ dataset containing the item parameters.

2. An alternative strategy for implementing the three-parameter IRT DGP is to write two functions instead of one: a function to simulate a set of M item parameters and a function to simulate the person-by-item responses. Complete the following function skeletons to implement this strategy. Demonstrate how to use the functions to simulate a dataset.

```
r_3PL_items <- function(
  M,
  diff_M = 0, diff_SD = 1,
  disc_M = 1, disc_SD = 0.2,
  item_options = 4
) {
  # Code here
  return(item_parameter_data)
}

r_3PL_data <- function( N, item_data ) {
  # Generate item responses
  return(response_data)
}
```

3. Describe the benefits and limitations of the two approaches you have implemented.

4. For your preferred strategy, modify your function(s) to also return the latent person ability parameters.

Exercise 6.12 (Check the 3-parameter IRT DGP)**.** For one of the DGPs you wrote in Exercise 6.11, write code to check that the function is working properly. What features of the model will you check?

Exercise 6.13 (Explore the 3-parameter IRT model)**.** Use the modified DGP function(s) you wrote for Exercise 6.11 to explore the model. Simulate data and create visualizations to answer the following questions. Be sure to specify what your other parameters are, as you test how a target parameter of interest varies.

1. What happens to the distribution of total scores if the items cover a broad range of difficulties (i.e., a very high τ_α)?

2. What happens if the items are mis-calibrated to be systematically too difficult ($\mu_\alpha > 0$)?

3. Find at least one combination of parameter values that will produce very extreme or unrealistic distributions of total scores.

Exercise 6.14 (Random effects meta-regression)**.** Meta-analysis involves working with quantitative findings reported by previous studies on a particular topic, in the form of effect size estimates and their standard errors, to generate integrative summaries and identify patterns in the findings that might not be evident from any previous study considered in isolation. One model that is widely used in meta-analysis is the random effects meta-regression, which relates the effect sizes to known characteristics of the studies that are encoded in the form of predictors. Consider a collection of K studies. Let T_i denote an effect size estimate and s_i denote its standard error for study i. Let x_i be a quantitative predictor variable that represents some characteristic of study i (such as its year of publication). The random effects meta-regression model assumes

$$T_i = \beta_0 + \beta_1 x_i + u_i + e_i,$$

where $u_i \sim N(0, \tau^2)$ and $e_i \sim N(0, s_i^2)$. In this model, β_0 corresponds to the expected effect size when $x_i = 0$ and β_1 describes the expected difference in effect size per one unit difference in x_i. The first error u_i corresponds to the heterogeneity in the effect sizes above and beyond the variation explained by x_i, and the second error e_i corresponds to the sampling error, which we assume has known variance s_i^2.

1. Of the variables in the random effects meta-regression model, which are structural features, which are covariates, and which are outcomes? Of the parameters described above, which would you classify as focal, which as auxiliary, and which as design parameters?

2. In addition to the focal model described above, what further assumptions or auxiliary models will be needed in order to simulate data for a random effects meta-regression? Propose an auxiliary model for any needed quantities.

3. Using your proposed auxiliary model, write a function to generate data for a random effects meta-regression. Demonstrate your function by creating a plot based on a simulated dataset with $K = 30$ effect sizes.

4. Write code to check the properties of your data-generating function.

Exercise 6.15 (Meta-regression with selective reporting)**.** Vevea and Hedges (1995) proposed a meta-regression model that allows for the possibility that not all primary study results are published. They assume that primary study results are generated according to the random effects meta-regression model described in Exercise 6.14, but then only a subset of results are observed, where the probability of being included is a function of the result's one-sided p-value for the null hypothesis $H_0 : \delta \leq 0$ against alternative $H_A : \delta > 0$. Let $p_i = 1 - \Phi(T_i / s_i)$ be the one-sided p-value for study result i, where $Phi()$ is the standard normal cumulative distribution (`pnorm()` in R). In one version of the selection model, the selection probability follows a piece-wise constant distribution with

$$\Pr(T_i \text{ is observed}) = \begin{cases} 1 & \text{if} \quad 0 \leq p_i < .025 \\ \lambda_1 & \text{if} \quad .025 \leq p_i < .500 \\ \lambda_2 & \text{if} \quad .500 \leq p_i < 1 \end{cases}$$

for selection probabilities $0 \leq \lambda_1 \leq 1$ and $0 \leq \lambda_2 \leq 1$.

1. Modify your data-generating function from Exercise 6.14 to follow the Vevea-Hedges selection model. Ensure that the new data-generating function returns a total of K primary study results.

2. Use the new function to generate a large number of effect size estimates, all with $s_i = 0.35$, using parameters $\beta_0 = 0.1$, $\beta_1 = 0.0$, $\tau = 0.1$, $\lambda_1 = 0.5$, and $\lambda_2 = 0.2$. Plot the distribution of observed effect size estimates.

3. Create several further plots using different values for λ_1 and λ_2. How do these parameters affect the shape of the distribution?

4. Use the `selmodel()` function from the `metafor` package to estimate the Vevea-Hedges selection model based on one of your simulated datasets. (See Exercise 7.7 for example syntax.) How do the estimates compare to the model parameters you've specified?

7

Estimation procedures

We do simulation studies to understand how to analyze data. Thus, the central object of study is a *data analysis procedure*, or a set of steps or calculations carried out on a dataset. We want to know how well a procedure would work when applied in practice, and the data generating processes we described in Chapter 6 provides a stand-in for real data. To revisit our consumer product testing analogy from Chapter 1, the data analysis procedure is the product, and the data generating process is the set of trials to which we subject the product.

Depending on the research question, a data-analysis procedure might be very simple—as simple as just computing a sample correlation—or it might involve something more complex, such as fitting a multilevel model and generating a confidence interval for a specific coefficient. A data analysis procedure might even involve a combination of several components. For example, the procedure might entail first running a variable screening procedure and then fitting a random forest on the selected predictor variables. As another example, a data-analysis procedure might involve using multiple imputation for missingness on key variables, then fitting a statistical model, and then generating predicted values based on the model. For sake of brevity, we will use the term *estimation procedure* or just *estimator* to encompass all of these procedures, even ones that involve multi-step or multi-component procedures.

In this chapter, we demonstrate how to implement an estimator in the form of an R function, which we call an *estimation function*, so that its performance can eventually be evaluated by repeatedly applying it to artificial data. We start by describing the high-level design of an estimation function and illustrate with some simple examples in Section 7.1.

Depending on the research question, we will often be interested in comparing several competing estimators. In this case, we will create *several* functions that implement the estimators that we plan to compare. The easiest approach here is to implement each estimator in turn, and then bundle the collection in a final overall function. We describe how to do this in Section 7.2.

In Section 7.3, we describe strategies for validating the coded-up estimator before running a full simulation. We offer a few strategies for validating ones code, including checking against existing implementations, checking theoreti-

cal properties, and using simulations to check for bugs.

For a full simulation, an estimator needs to be reliable, not crashing and giving answers across the wide range of datasets to which it is applied. An estimator will need to be able to handle odd edge cases or pathological datasets that might be generated as we explore a full range of simulation scenarios. To allow for this, Section 7.4 demonstrates several methods for handling common computational problems, such as errors or warnings.

7.1 Writing estimation functions

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

```
estimator <- function(data) {

  # calculations/model-fitting/estimation procedures

  return(estimates)
}
```

An estimator function should take a dataset as input, fit a model or otherwise calculates an estimate, possibly with associated standard errors and so forth, and return these quantities as output. It can return point estimates of parameters, standard errors, confidence intervals, p-values, predictions, or other quantities. The calculations in the body of the function should be set up to use datasets that have the same structure (i.e., same dimensions, same variable names) as the output of the corresponding function for generating simulated data. However, in principle, we should also be able to run the estimation function on real data as well.

As a first example, suppose we want to evaluate a method for generating a confidence interval for Pearson's sample correlation coefficient when faced with non-normal data. For bivariate normal data, statistical theory provides some very useful facts. In particular, it tells us that applying Fisher's z-transformation of Pearson's correlation coefficient, which is equivalent to the hyperbolic arc-tangent function (`atanh()` in R), produces a statistic that is very close to normally distributed. It also tells us that the empirical standard error of the z-transformed correlation coefficient is simply $1/\sqrt{N-3}$, and thus independent of the correlation parameter. This makes z-transformation very useful for computing confidence intervals, which can then be back-transformed to the Pearson-r scale. However, these results are specific to bivariate normal distributions. Would this transformation work well in the face of non-normal data, such as the bivariate Poisson distribution we coded in Chapter 6?

For studying this question with simulation, a simple estimation function would take a dataset with two variables as input and compute the sample correlation and its z-transformation, compute confidence intervals for z, and then back-transform the confidence interval end-points. Here is an implementation of this sequence of calculations:

```
r_and_z <- function(data) {

  r <- cor(data$C1, data$C2)
  z <- atanh(r)
  se_z <- 1 / sqrt(nrow(data) - 3)
  ci_z <- z + c(-1, 1) * qnorm(.975) * se_z
  ci_r <- tanh(ci_z)

  data.frame( r = r, z = z, CI_lo = ci_r[1], CI_hi = ci_r[2] )
}
```

To check that the function returns a result of the expected form, we generate a small dataset using the `r_bivariate_Poisson()` function developed in the last chapter, then apply our estimation function to the result:

```
Pois_dat <- r_bivariate_Poisson(40, rho = 0.5, mu1 = 4, mu2 = 4)
r_and_z(Pois_dat)
```

```
          r        z     CI_lo     CI_hi
1 0.6371712 0.753397 0.4063077 0.7915666
```

Although it is a little cumbersome to do so, we could also apply the estimation function to a real dataset. Here is an example, which calculates the correlation between ratings of judicial integrity and familiarity with the law from the `USJudgeRatings` dataset (which is included in base R). For the function to work on this dataset, we first need to rename the relevant variables because our function takes a `data.frame` with two columns named `C1` and `C2`:

```
data(USJudgeRatings)

USJudgeRatings %>%
  dplyr::select(C1 = INTG, C2 = FAMI) %>%
  r_and_z()
```

```
         r        z     CI_lo     CI_hi
1 0.868858 1.328401 0.7692563 0.9272343
```

The function returns a valid result—a quite strong correlation in this case!

It is good practice to test out a newly-developed estimation function on real data as a check that it is working as intended. Testing on real data ensures that the estimation function is not using information outside of the dataset, such as by using known parameter values to construct an estimate. Applying

the function to a real dataset demonstrates that the function implements a procedure that could actually be applied in real data analysis contexts.

7.2 Including Multiple Data Analysis Procedures

Many simulations involve head-to-head comparisons between more than one data-analysis procedure. As a design principle, we generally recommend writing different functions for each estimation method one is planning on evaluating. Doing so makes it easier to add in additional methods as desired or to focus on just a subset of methods. Writing separate functions also leads to a code base that is flexible and useful for other purposes (such as analyzing real data). Finally (repeating one of our favorite mantras), separating functions makes debugging easier because it lets you focus attention on one thing at a time, without worrying about how errors in one area might propagate to others.

To see how this works in practice, we return to the case study from Section 6.6, where we developed a data-generating function for simulating a cluster-randomized trial with student-level outcomes but school-level treatment assignment. Our data-generating process allowed for varying school sizes and heterogeneous treatment effects that are correlated with school size. Several different procedures might be used to estimate an overall average effect from a clustered experiment. We will consider three different procedures:

- Fitting a multi-level regression model (also known as a hierarchical linear model),
- Fitting an ordinary least squares (OLS) regression model and applying cluster-robust standard errors, or
- Averaging the outcomes by school, then fitting a linear regression model on the mean outcomes.

All three of these methods are widely used and have some theoretical guarantees supporting their use. Education researchers tend to be more comfortable using multi-level regression models, whereas economists tend to use OLS with clustered standard errors.

We next develop estimation functions for each of these procedures, focusing on a simple model that does not include any covariates besides the treatment indicator. Each function needs to produce a point estimate, standard error, and p-value for the average treatment effect. To write estimation functions, it is useful to work with an example dataset that has the same structure as the data that will be simulated. To that end, we generate a sample dataset using a revised version of `gen_cluster_RCT()`, which corrects the bug discussed in Exercise 6.8:

```
dat <- gen_cluster_RCT(
  J=16, n_bar = 30, alpha = 0.8, p = 0.5,
  gamma_0 = 0, gamma_1 = 0.2, gamma_2 = 0.2,
  sigma2_u = 0.4, sigma2_e = 0.6
)
```

For the multi-level modeling strategy, there are several different existing packages that we could use. We will implement an estimator using the popular `lme4` package, via the `lmerTest` function which calculates a p-value for the average effect. Here is a basic implementation:

```
analysis_MLM <- function( dat ) {

  M1_test <- lmerTest::lmer( Yobs ~ 1 + Z + (1 | sid), data = dat )
  M1_summary <- summary(M1_test)$coefficients

  tibble(
    ATE_hat = M1_summary["Z","Estimate"],
    SE_hat = M1_summary["Z","Std. Error"],
    p_value = M1_summary["Z", "Pr(>|t|)"]
  )

}
```

The function fits a multi-level model with a fixed coefficient for the treatment indicator and random intercepts for each school. It outputs only the statistics in which we are interested.

Our function makes use of the `lmerTest` package. Rather than assuming that this package will be loaded, we call the relevant function using the package name as a prefix: `lmerTest::lmer()`. This way, we can run the function even if we have not loaded the package in the global environment. Referencing functions with their package prefix is also preferable to loading packages inside the function itself (e.g., with `require(lmerTest)`) because calling the function then does not change which packages are loaded in the global environment.

Here is a function implementing OLS regression with cluster-robust standard errors:

```
analysis_OLS <- function( dat, se_type = "CR2" ) {

  M2 <- estimatr::lm_robust(
    Yobs ~ 1 + Z, data = dat,
    clusters = sid,  se_type = se_type
  )

  tibble(
```

```
    ATE_hat = M2$coefficients[["Z"]],
    SE_hat = M2$std.error[["Z"]],
    p_value = M2$p.value[["Z"]]
  )
}
```

To get cluster-robust standard errors, we use the `lm_robust()` function from the `estimatr()` package, again calling only the relevant function using the package prefix rather than loading the whole package. A novel aspect of this estimation function is that it includes an additional intput argument, `se_type`, which allows us to control the type of standard error calculated by `lm_robust()`. Adding this option would let us use the same function to compute (and compare) different types of clustered standard errors for the average treatment effect estimate. We set a default option of `"CR2"`, just like the default of `lm_robust()`.

Sometimes an analytic procedure involves multiple steps. For example, the aggregation estimator first involves collapsing the data to a school-level dataset, and then analyzing at the school level. This is no problem:
we just wrap all the steps in a single estimation function. Once written, all we need to do is call the function—no matter how complicated the process inside. Here is the code for the aggregate-then-analyze approach:

```
analysis_agg <- function( dat, se_type = "HC2" ) {

  datagg <- dplyr::summarise(
    dat,
    Ybar = mean( Yobs ),
    n = n(),
    .by = c(sid, Z)
  )

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

  M3 <- estimatr::lm_robust(
    Ybar ~ 1 + Z, data = datagg,
    se_type = se_type
  )

  tibble(
    ATE_hat = M3$coefficients[["Z"]],
    SE_hat = M3$std.error[["Z"]],
    p_value = M3$p.value[["Z"]]
  )
}
```

Note the `stopifnot` command: it will throw an error if the condition is not true. This `stopifnot` ensures we do not have both treatment and control students within a single school–if we did, we would have more aggregated values than school ids due to the grouping! Putting *assert statements* in your code like this is a good way to guarantee you are not introducing weird and hard-to-track errors. A `stopifnot` statement halts your code as soon as something goes wrong, rather than letting that initial error flow on to further work, potentially creating odd results that are hard to understand. Here we are protecting ourselves from strange results if, for example, we messed up our DGP code to have treatment not nested within school, or if we were using data that did not actually come from a cluster randomized experiment. See Section 17.4 for more.

All of our functions produce output in the same format:

```
analysis_MLM( dat )
```

```
# A tibble: 1 x 3
  ATE_hat SE_hat p_value
    <dbl>  <dbl>   <dbl>
1  -0.176  0.383   0.653
```

```
analysis_OLS( dat )
```

```
# A tibble: 1 x 3
  ATE_hat SE_hat p_value
    <dbl>  <dbl>   <dbl>
1  -0.124  0.462   0.793
```

```
analysis_agg( dat )
```

```
# A tibble: 1 x 3
  ATE_hat SE_hat p_value
    <dbl>  <dbl>   <dbl>
1  -0.178  0.378   0.645
```

Ensuring that the output of all the functions is structured in the same way will make it easy to keep the results organized once we start running multiple iterations of the simulation. If each estimation method returns a dataset with the same variables, we can simply stack the results on top of each other. Here is a function that bundles all the estimation procedures together:

```
analyze_data <- function(
    data,
    CR_se_type = "CR2", agg_se_type = "HC2"
) {

  dplyr::bind_rows(
    MLM = analysis_MLM( data ),
```

```
    OLS = analysis_OLS( data, se_type = CR_se_type),
    agg = analysis_agg( data, se_type = agg_se_type),
    .id = "estimator"
  )

}
```

```
analyze_data(dat)
```

```
# A tibble: 3 x 4
  estimator ATE_hat SE_hat p_value
  <chr>       <dbl>  <dbl>   <dbl>
1 MLM        -0.176  0.383   0.653
2 OLS        -0.124  0.462   0.793
3 agg        -0.178  0.378   0.645
```

This is a common coding pattern for simulations that involve multiple estimators. Each procedure is expressed in its own function, then these are assembled together in a single function so that they can all easily be applied to the same dataset. Stacking the results row-wise will make it easy to compute performance measures for all methods at once. The benefit of stacking will become even more evident once we are working across multiple replications of the simulation process, as we will in Chapter 8.

7.3 Validating an Estimation Function

Just as with data-generating functions, it is critical to verify that an estimation function is correctly implemented. If an estimation function involves a known procedure that has been implemented in R or one of its contributed packages, then a straightforward way to do this is to compare your implementation to another existing implementation. For estimation functions that involve multi-step procedures or novel methods, other approaches to verification may be needed.

7.3.1 Checking against existing implementations

In Chapter 5, we wrote a function called `ANOVA_Welch_F()` for computing p-values from two different procedures for testing equality of means in a heteroskedastic ANOVA:

```
ANOVA_Welch_F <- function(data) {
  anova_F <- oneway.test(x ~ group, data = data, var.equal = TRUE)
  Welch_F <- oneway.test(x ~ group, data = data, var.equal = FALSE)
```

```
  result <- tibble(
    ANOVA = anova_F$p.value,
    Welch = Welch_F$p.value
  )

  return(result)
}
```

We can test this function by comparing its output against the built-in `oneway.test` function, as follows:

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

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 = 15.577, num df = 3.0000, denom df = 3.0993, p-value
= 0.02275
```

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

```
[1] TRUE
```

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

For the bivariate correlation example introduced in Section 7.1, we can check the output of `r_and_z()` against R's built-in `cor.test()` function:

```
Pois_dat <- r_bivariate_Poisson(15, rho = 0.6, mu1 = 14, mu2 = 8)

my_result <- r_and_z(Pois_dat) |> subset(select = -z)
my_result
```

```
          r     CI_lo     CI_hi
1 0.7474359 0.3810838 0.9109217
```

```
R_result <- cor.test(~ C1 + C2, data = Pois_dat)
R_result <- tibble(r = R_result$estimate[["cor"]],
```

```
                    CI_lo = R_result$conf.int[1],
                    CI_hi = R_result$conf.int[2])
R_result
```

```
# A tibble: 1 x 3
      r CI_lo CI_hi
  <dbl> <dbl> <dbl>
1 0.747 0.381 0.911
```

```
all.equal( as.numeric(R_result), as.numeric(my_result) )
```

```
[1] TRUE
```

This type of check is all the more useful here because `r_and_z()` uses our own implementation of the confidence interval calculations, rather than relying on R's built-in functions as we did with `ANOVA_Welch_F()`.

One might ask why you should bother implementing your own function if you already have a version implemented in R. In some cases, it might be possible to implement a faster version of a function because you can cut corners by skipping input data verification, providing fewer estimation options, or cutting out post-estimation processing. Any of these could help with the overall speed of your simulation. On the other hand, gains in computational efficiency might not be worth the human time it would take to implement the calculations from scratch. See Section A.4 for more discussion of this trade-off.

7.3.2 Checking novel procedures

Simulations are usually an integral part of projects to develop novel statistical methods. Checking estimation functions in such projects presents a challenge: if an estimator truly is new, how do you check that your code is correct? Effective methods for doing so will vary from problem to problem, but an over-arching strategy is to use theoretical results about the performance of the estimator to check that your implementation works as expected. For instance, we might work out the algebraic properties of an estimator for a special case and then check that the result of the estimation function agrees with our algebra. For some estimation problems, we might be able to identify theoretical properties of an estimator when applied to a very large sample of data and when the model is correctly specified. If we can find results about large-sample behavior, then we can test an estimation function by applying it to a very large sample and checking whether the resulting estimates are very close to specified parameter values. We illustrate each of these approaches using our functions for estimating treatment effects from cluster-randomized trials.

We start by testing an algebraic property. With each of the three methods we have implemented, the treatment effect estimator is a difference between the weighted average of the outcomes from students in each treatment condition;

the only difference between the estimators is in what weights are used. In the special case where all schools have the same number of students, the weights used by all three methods end up being the same: all three methods allocate equal weight to each school. Therefore, we know that there should be no difference between the three point estimates. Furthermore, a bit of algebra will show that the cluster-robust standard error from the OLS approach will end up being identical to the robust standard error from the aggregation approach. If there are also equal numbers of schools assigned to both conditions, then the standard error from the multilevel model will also be identical to the other standard errors.

Let's verify that our estimation functions produce results that are consistent with these theoretical properties. To do so, we will need to generate a dataset with equal cluster sizes, setting $\alpha = 0$:

```
dat <- gen_cluster_RCT(
  J=12, n_bar = 30, alpha = 0, p = 0.5,
  gamma_0 = 0, gamma_1 = 0.2, gamma_2 = 0.2,
  sigma2_u = 0.4, sigma2_e = 0.6
)
table(dat$sid) # verify equal-sized clusters
```

```
 1  2  3  4  5  6  7  8  9 10 11 12
30 30 30 30 30 30 30 30 30 30 30 30
```

```
analyze_data(dat)
```

```
# A tibble: 3 x 4
  estimator ATE_hat SE_hat p_value
  <chr>       <dbl>  <dbl>   <dbl>
1 MLM         0.456  0.401   0.282
2 OLS         0.456  0.401   0.282
3 agg         0.456  0.401   0.282
```

All three methods yield identical results. Now let's try equal school sizes but unequal allocation to treatment:

```
dat <- gen_cluster_RCT(
  J=12, n_bar = 30, alpha = 0, p = 2 / 3,
  gamma_0 = 0, gamma_1 = 0.2, gamma_2 = 0.2,
  sigma2_u = 0.4, sigma2_e = 0.6
)
analyze_data(dat)
```

```
# A tibble: 3 x 4
  estimator ATE_hat SE_hat p_value
  <chr>       <dbl>  <dbl>   <dbl>
```

```
1 MLM          0.292  0.309   0.367
2 OLS          0.292  0.260   0.303
3 agg          0.292  0.260   0.287
```

As expected, all three point estimators match, but the SE from the multilevel model is a little bit discrepant from the others.

We can also use large-sample theory to check the multilevel modeling estimator. If the model is correctly specified, then *all* the parameters of the model should be accurately estimated if the model is fit to a very large sample of data. To check this property, we will need access to the full model output, not just the selected results returned by `analysis_MLM()`. One way to handle this is to make a small tweak to the estimation function, adding an option to control whether to return the entire model or just selected results. Here is the tweaked function:

```
analysis_MLM <- function( dat, all_results = FALSE) {

  M1_test <- lmerTest::lmer( Yobs ~ 1 + Z + (1 | sid), data = dat )

  if (all_results) {
    return(summary(M1_test))
  }

  M1_summary <- summary(M1_test)$coefficients

  tibble(
    ATE_hat = M1_summary["Z","Estimate"],
    SE_hat = M1_summary["Z","Std. Error"],
    p_value = M1_summary["Z", "Pr(>|t|)"]
  )
}
```

Setting `all_results` to `TRUE` will return the entire function; keeping it at the default value of `FALSE` will return the same output as the other functions. Now let's apply the estimation function to a very large dataset, with variation in cluster sizes. We set `gamma_2 = 0` so that the estimation model is correctly specified:

```
dat <- gen_cluster_RCT(
  J=5000, n_bar = 20, alpha = 0.9, p = 2 / 3,
  gamma_0 = 2, gamma_1 = 0.30, gamma_2 = 0,
  sigma2_u = 0.4, sigma2_e = 0.6
)

analysis_MLM(dat, all_results = TRUE)
```

```
Linear mixed model fit by REML. t-tests use Satterthwaite's
```

```
  method [lmerModLmerTest]
Formula: Yobs ~ 1 + Z + (1 | sid)
   Data: dat

REML criterion at convergence: 241831

Scaled residuals:
    Min      1Q  Median      3Q     Max
-4.5540 -0.6605  0.0009  0.6593  4.3081

Random effects:
 Groups   Name        Variance Std.Dev.
 sid      (Intercept) 0.3924   0.6264
 Residual             0.5984   0.7736
Number of obs: 98739, groups:  sid, 5000

Fixed effects:
             Estimate Std. Error        df t value Pr(>|t|)
(Intercept) 2.005e+00  1.627e-02 4.960e+03   123.2   <2e-16 ***
Z           3.028e-01  1.992e-02 4.949e+03    15.2   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Correlation of Fixed Effects:
  (Intr)
Z -0.817
```

The intercept and treatment effect coefficient estimates are very close to their true parameter values, as are the estimated school-level variance and student-level residual variance. This all gives some assurance that the `analysis_MLM()` function is working properly.

Of course, it is important to bear in mind that these tests are only partial verifications. With the algebraic test, we have only checked that the functions seem to be working properly for scenarios with equal school sizes, but they still might have errors that only appear when school sizes vary. Likewise, `analysis_MLM()` seems to be working properly for very large datasets, but our test does not rule out the possibility of bugs that only crawl out when J is small. Our large-sample test also relies on the correctness of the `gen_cluster_RCT()` function; if we had seen a discrepancy between parameters and estimates from the multilevel model, it could have been because of a problem with the data-generation function rather than with the estimation function.

These limitations are typical of what can be accomplished through tests based on theoretical results, because theoretical results typically only hold under

specific conditions. After all, if we had comprehensive theoretical results, we would not need to simulate anything in the first place! Nonetheless, it is good to work through such tests to the extent that relevant theory is available for the problem you are studying.

7.3.3 Checking with simulations

Checking, debugging, and revising should not be limited to when you are initially developing estimation functions. It often happens that later steps in the process of conducting a simulation will reveal problems with the code for earlier steps. For instance, once you have run the data-generating and estimation steps repeatedly, calculated performance summaries, and created some graphs of the results, you might find an unusual or anomolous pattern in the performance of an estimator. This might be a legitimate result—perhaps the estimator really does behave weirdly or not work well—or it might be due to a problem in how you implemented the estimator or data-generating process. When faced with an unusual pattern, revisit the estimation code to double check for bugs and also think further about what might lead to the anomaly. Further exploration might lead you to a deeper understanding of how a method works and perhaps even an idea for how to improve the estimator or refine the data-generating process.

A good illustration of this process comes from one of Luke's past research projects (see Pashley et al. (2024)), in which he and other co-authors were working on a way to improve Instrumental Variable (IV) estimation using post-stratification. The method they studied involved grouping units based on a covariate that predicts compliance status, then calculating estimates within each group, then averaging the estimates across groups. They used simulations to see whether this method would improve the accuracy of the overall summary effect estimate. In the first simulation, the estimates were full of missing values and odd results because the estimation function failed to account for what happens in groups of observations where the number of compliers was estimated to be zero. After repairing that problem and re-running everything, the simulation results still indicated serious and unexpected bias, which turned out to be due to an error in how the estimation function implemented the step of averaging estimates across groups. After again correcting and re-running, the simulation results showed that the gains in accuracy from this new method were minimal, even when the groups were formed based on a variable that was almost perfectly predictive of compliance status. Eventually, we understood that the groups with very few compliers produced such unstable estimates that they spoiled the overall average estimate. This inspired us to revise our estimation strategy and introduce a method that dropped or down-weighted strata with few compliers, which ultimately helped us to strengthen the contribution of our work.

As this experience highlights, simulations seldom follow a single, well-defined

trajectory. The point of conducting simulations is to help us, as researchers, learn about estimation methods so that we can analyze real data better. Simulations can strengthen our methodological and theoretical understanding, which can then inspire ideas for better approaches which we can then test in subsequent simulations. Of course, at some point one needs to step off this merry-go-round, write up the findings, cook dinner, and clean the bathroom. But, just like many other research endeavors, simulations are often a highly iterative process.

7.4 Handling errors, warnings, and other hiccups

Especially when working with more advanced estimation methods, it is possible that your estimation function will fail, throw an error, or return something uninterpretable for certain input datasets. For instance, maximum likelihood estimation often involves using iterative, numerical optimization algorithms that sometimes fail to converge. This might happen rarely enough that it takes a while to even notice that it is a problem, but even quite rare things can occur when you run simulations with many thousands of repetitions. Less dire but still annoying, your estimation function might occasionally generate warnings, which can pile up if you are running many repetitions. In some cases, such warnings might also signal that the estimator produced a bad result, and it may not be clear whether we should retain this result in the estimator's overall performance assessments. After all, the function tried to warn us that something is off!

Errors and warnings in estimation functions pose two problems, one purely technical and one conceptual. On a technical level, R functions stop running if they hit an error, so we need ways to handle the errors in order to get our simulations up and running. Furthermore, warnings can clutter up the console and slow down code execution, so we may want to capture and suppress them as well. On a conceptual level, we need to decide how to use the information contained in errors and warnings, whether that be by further elaborating the estimators to address different contingencies or by evaluating the performance of the estimators in a way that appropriately accounts for these events. We consider both these problems here, and then revisit the conceptual considerations in Chapter 9, where we discuss assessing estimator performance.

7.4.1 Capturing errors and warnings

Some estimation functions will require complicated or stochastic calculations that can sometimes produce errors. Intermittent errors can really be annoying and time-consuming if not addressed. To protect yourself, it is good practice to

anticipate potential errors, so that you can prevent them from stopping code execution and allow your simulations to keep running. We next demonstrate some techniques for error-handling using tools from the `purrr` package.

For illustrative purposes, consider the following error-prone function that sometimes returns what we want, sometimes returns `NaN` due to taking the square root of a negative number, and sometimes crashes completely because `broken_code()` does not exist:

```
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]] )
    }
}
```

Running it a few times produces a mix of results and warnings:

```
set.seed(156858)
my_complex_function( 7 )
```

```
[1] NaN
```

```
my_complex_function( 7 )
```

```
[1] 2.11999
```

```
my_complex_function( 4 )
```

```
[1] 0.09896136
```

Running it many times produces warnings, then an error:

```
set.seed( 131 )
resu <- replicate(20, my_complex_function( 6 ))
```

```
Error in `broken_code()`:
! could not find function "broken_code"
```

We need to "trap" these warnings and errors so they do not clutter or stop our simulation. Let's do the errors first.

The `purrr` package includes a function called `possibly()` that makes it easy to trap errors:

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

```
[1] NaN
```

```
rs <- replicate(20, my_possible_function( 7 ))
rs
```

```
 [1] 0.8653335       NaN 2.0056463       NaN        NA       NaN
 [7]       NaN 1.0718040 1.5937092 2.0111968       NaN        NA
[13] 1.7401242       NaN 0.9721895       NaN 2.0947465       NaN
[19]       NaN       NaN
```

`possibly()` is an example of a *functional* (or an *adverb* or "wrapper function"), which takes a function and returns a new function that does something slightly different. The new function has the exact same parameters as the function we put into it. The difference is the new version of the function will, if an error happens, return the value specified by the `otherwise` argument (here, `NA`), rather than stopping execution with an error. It does not silence the warnings, however.

To handle warnings, the `quietly()` functional leads to results that are bundled together with any console output, warnings, and messages, rather than printing anything to the console:

```
my_quiet_function <- quietly( my_complex_function )

my_quiet_function( 1 )
```

```
$result
[1] 0.6065094

$output
[1] ""

$warnings
character(0)

$messages
character(0)
```

Wrapping a function with `quietly()` can be especially useful to reduce extraneous printing in a simulation, which can slow down code execution more than you might expect. However, `quietly()` does not trap errors:

```
rs <- replicate(20, my_quiet_function( 7 ))
```

```
Error in `broken_code()`:
! could not find function "broken_code"
```

To handle both errors and warnings, we double-wrap the function, first with `possibly()` and then with `quietly()`:

```
my_safe_quiet_function <- quietly( possibly( my_complex_function, otherwise = NA ) )
my_safe_quiet_function(7)
```

```
$result
[1] 1.538658

$output
[1] ""

$warnings
character(0)

$messages
character(0)
```

To see how this works in practice, we will adapt our `analysis_MLM()` function, which makes use of `lmer()` for fitting a multilevel model. Currently, the estimation function sometimes prints messages to the console:

```
set.seed(101012)  # (hand-picked to show a warning.)
dat <- gen_cluster_RCT( J = 50, n_bar = 100, sigma2_u = 0 )
mod <- analysis_MLM(dat)
```

Wrapping `lmer()` with `quietly()` makes it possible to catch such output and store it along with other results, as in the following:

```
quiet_safe_lmer <- quietly( possibly( lmerTest::lmer, otherwise = NULL ) )
M1 <- quiet_safe_lmer( Yobs ~ 1 + Z + (1|sid), data=dat )
M1
```

```
$result
Linear mixed model fit by REML ['lmerModLmerTest']
Formula: ..1
   Data: ..2
REML criterion at convergence: 14026.44
Random effects:
 Groups   Name        Std.Dev.
 sid      (Intercept) 0.0000
 Residual             0.9828
Number of obs: 5000, groups:  sid, 50
Fixed Effects:
(Intercept)            Z
  -0.013930    -0.008804
optimizer (nloptwrap) convergence code: 0 (OK) ; 0 optimizer warnings; 1 lme4 warnings

$output
[1] ""
```

```
$warnings
character(0)

$messages
[1] "boundary (singular) fit: see help('isSingular')\n"
```

In our estimation function, we replace the original `lmer()` call with our quiet-and-safe version, `quiet_safe_lmer()`. We then pick apart the pieces of its output to return a tidy dataset of results. (Separately, we also add in a generally preferred optimizer, bobyqa, to try and avoid the warnings and errors in the first place.) The resulting function is:

```
analysis_MLM_safe <- function( dat ) {

  M1 <- quiet_safe_lmer( Yobs ~ 1 + Z + (1 | sid),
                         data=dat,
                         control = lme4::lmerControl(
                           optimizer = "bobyqa"
                         ) )

  if ( is.null( M1$result ) ) {
    # we had an error!
    tibble( ATE_hat = NA, SE_hat = NA, p_value = NA,
            message = M1$message,
            warning = M1$warning,
            error = TRUE )
  } else {
    # no error!
    sum <- summary( M1$result )

    tibble(
      ATE_hat = sum$coefficients["Z","Estimate"],
      SE_hat = sum$coefficients["Z","Std. Error"],
      p_value = sum$coefficients["Z", "Pr(>|t|)"],
      message = length( M1$messages ) > 0,
      warning = length( M1$warning ) > 0,
      error = FALSE
      )
  }
}
```

Our improved function runs without extraneous messages and returns the estimation results along with any diagnostic information from messages or warnings:

```
mod <- analysis_MLM_safe(dat)
mod
```

```
# A tibble: 1 x 6
   ATE_hat SE_hat p_value message warning error
     <dbl>  <dbl>   <dbl> <lgl>   <lgl>   <lgl>
1 -0.00880 0.0278   0.751 TRUE    FALSE   FALSE
```

Recall that `analysis_MLM()` was one part of our method that ran all three estimators (MLM, OLS, and Agg) on our data. If we do not want to protect the other methods (perhaps because they do not seem to fail) we can write a "quiet" version of our `analyze_data()` function as so:

```
quiet_analyze_data <- function(dat,
                               CR_se_type = "CR2",
                               agg_se_type = "HC2") {

  # MLM
  a = Sys.time()
  MLM <- analysis_MLM_safe(dat)
  MLM$seconds <- as.numeric(Sys.time() - a, units = "secs")

  # OLS
  a = Sys.time()
  LR <- analysis_OLS(dat, se_type = CR_se_type)
  LR$seconds <- as.numeric(Sys.time() - a, units = "secs")
  LR$message <- 0
  LR$warning <- 0
  LR$error <- 0

  # Aggregated
  a = Sys.time()
  Agg <- analysis_agg(dat, se_type = agg_se_type)
  Agg$seconds <- as.numeric(Sys.time() - a, units = "secs")
  Agg$message <- 0
  Agg$warning <- 0
  Agg$error <- 0

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

For the other methods we manually set the flags for messages and warnings to 0, and then stack to get our final results. We also calculate run time for each method, which can be useful for determining where our simulation might be especially slow. See Section A.3 for more on timing.

Here is what our method now returns:

```
quiet_analyze_data(dat) %>%
  print( digits = 3 )
```

```
# A tibble: 3 x 8
  method  ATE_hat SE_hat p_value message warning error seconds
  <chr>     <dbl>  <dbl>   <dbl>   <dbl>   <dbl> <dbl>   <dbl>
1 MLM    -0.00880 0.0278   0.751       1       0     0  0.0791
2 LR     -0.00880 0.0257   0.733       0       0     0  0.0547
3 Agg    -0.00880 0.0257   0.733       0       0     0  0.0104
```

Storing error and warning information will let us evaluate what proportion of the time there was a concern, run additional analyses on the subset of replications where there was no such warning, or even modify the estimator to take the diagnostics into account. We have solved the technical problem—our code will run and give results—but not the conceptual one: what does it mean when our estimator gives an NA or a convergence warning with a nominal answer? How do we decide how good our estimator is when it does this?

7.4.2 Estimator processes that anticipate errors and warnings

So far, we have seen techniques for handling technical hiccups that occur when data analysis procedures do not always produce results. But how do we account for the absence of results in a simulation? In Chapter 9, we will delve into the conceptual issues with summarizing the performance of methods that do not always provide an answer. However, one of the best solutions to such problems still concerns the formulation of estimation processes, and so we introduce it here. That solution is to *re-define the estimation process* to include contingencies for handling a lack of results.

Consider a data analyst who was planning to apply a fancy statistical model to their data, but then finds that the model does not converge. What would that analyst do in practice (besides cussing and taking a snack break)? Rather than giving up entirely, they would probably think of an alternative analysis and attempt to apply it, perhaps by simplifying the model in some way. To the extent that we can anticipate such possibilities, we can build these error-contingent alternative analyses into our estimation function. In short, we can implement an *estimation procedure* or a *data-analysis procedure* rather than just an *estimator*. In the real world, an overall analysis strategy may involve multiple steps in a branching structure. We can try to emulate such strategies in our code, and then evaluate how well that multiple-step approach works using simulations.

To illustrate, consider an error (a not-particularly-subtle one) that could crop up in the cluster-randomized trial example when clusters are very small:

```
set.seed(65842)
tiny_dat <- gen_cluster_RCT( J = 10, n_bar = 2, alpha = 0.5)
analysis_MLM(tiny_dat)
```

```
Error:
! number of levels of each grouping factor must be < number of observations (problems: sid
```

The method failed! The error occured because all 10 simulated schools happened to include a single student each, making it impossible to estimate a random-intercepts multilevel model:

```
table(tiny_dat$sid)
```

```
 1  2  3  4  5  6  7  8  9 10
 1  1  1  1  1  1  1  1  1  1
```

A natural fall-back analysis here would be to estimate an ordinary least squares regression analysis.

Suppose that our imaginary analyst is not especially into nuance, and so will fall back onto ordinary least squares whenever the multilevel model produces an error. We can express this logic in our estimation function by first catching the error thrown by `lmer()` and then running an OLS regression in the event an error is thrown:

```
analysis_MLM_contingent <- function( dat ) {

  M1 <- quiet_safe_lmer( Yobs ~ 1 + Z + (1 | sid), data=dat )

  if (!is.null(M1$result)) {
    sum <- summary( M1$result )
    tibble(
      ATE_hat = sum$coefficients["Z","Estimate"],
      SE_hat = sum$coefficients["Z","Std. Error"],
      p_value = sum$coefficients["Z", "Pr(>|t|)"] )
  } else {
    # If lmer() errors, fall back on OLS
    M_ols <- summary(lm(Yobs ~ Z, data = dat))
    res <- tibble(
      ATE_hat = M_ols$coefficients["Z","Estimate"],
      SE_hat = M_ols$coefficients["Z", "Std. Error"],
      p_value = M_ols$coefficients["Z","Pr(>|t|)"] )
  }

  # Store original messages, warnings, errors
  res$message <- ifelse( length( M1$message ) > 0, M1$message, NA_character_ )
```

```
  res$warning <- ifelse( length( M1$warning ) > 0, M1$warning, NA_character_ )
  res$error <- is.null( M1$result )

  return(res)
}
```

We still store the messages, warnings, and errors from the initial `lmer()` fit so that we can keep track of how often errors occur. The function now returns a treatment effect estimate even if `lmer()` errors out:

```
analysis_MLM_contingent(tiny_dat)
```

```
# A tibble: 1 x 6
  ATE_hat SE_hat p_value message warning error
    <dbl>  <dbl>   <dbl> <chr>   <chr>   <lgl>
1   0.400  0.603   0.525 <NA>    <NA>    TRUE
```

Of course, we can easily anticipate the conditions under which this particular error will occur: all we need to do is check whether all the clusters are single observations. Because it is easily anticipated, a better strategy for handling this error would be to check *before* fitting the multilevel model and proceeding accordingly in the event that the clusters are all singletons. Exercise 7.5 asks you to implement this approach and further refine this contingent analysis strategy.

Adapting estimation functions to address errors can be an effective—and often very interesting—strategy for studying the performance of estimation procedures. Rather than studying the performance of a data-analysis method that is only sometimes well-defined, we shift to studying a stylized cognitive model for the analyst's decision-making process, one which handles contingencies that might crop up whether analyzing simulated data or real empirical data. Of course, studying such a model is only interesting to the extent that the implemented decision-making process is a plausible representation of what an analyst might actually do in practice.

The adaptive estimation approach does lead to more complex estimation functions, which entail implementing multiple estimation methods and a set of decision rules for applying them. Often, the set of contingencies that need to be handled will not be immediately obvious, so you may find that you need to build and refine the decision rules as you learn more about how they work. Running an estimator over multiple, simulated datasets is an excellent (if aggravating!) way to identify errors and edge cases. We turn to procedures for doing so in Chapter 8.

7.4.3 The `safely()` option

There is one final `purrr` option, `safely()`, which traps the full error message, instead of just replacing the output with a fixed value as `possibly()` does. `safely()` returns a list with two entries: the original result of the original function (or some fixed value if there was an error), and the error message (or NULL if there was no error).

To use it, we feed our function into `safely()` to create a new version:

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

```
$result
[1] 1.277916

$error
NULL
```

Just as with `possibly()`, we include `otherwise = NA` to set a return value if there is an error.

We can use the safe function repeatedly and it will always return a result:

```
resu <- replicate(20, my_safe_function( 7 ), simplify = FALSE)
```

We still get warnings, but any errors are trapped so the function does not crash.

We end up with a 20-entry list, with each element consisting of a pair of the result and error message:

```
length( resu )
```

```
[1] 20
```

```
resu[[3]]
```

```
$result
[1] NaN

$error
NULL
```

We can massage our data into a format that is easier to parse:

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

```
 [1]        NA        NA       NaN       NaN        NA 1.1766144
 [7]        NA 1.6221953        NA       NaN        NA 1.6086350
[13]       NaN 1.5356493        NA 1.1105727        NA       NaN
```

```
[19] 1.1576487 0.8228205
```

The `transpose()` function 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. `transpose()` 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     NA <smplErrr>
2     NA <smplErrr>
3    NaN <NULL>
4    NaN <NULL>
# i 16 more rows
```

We now have our results all organized, and we can see which iterations produced errors.

Unfortunately, to use `safely()` and `quietly()` together, we get a bit of a mess. Extending our cluster RCT example, we have:

```
quiet_safe_lmer <- quietly( safely( lmerTest::lmer, otherwise=NULL ) )
M1 <- quiet_safe_lmer( Yobs ~ 1 + Z + (1 | sid), data = dat )
M1
```

```
$result
$result$result
Linear mixed model fit by REML ['lmerModLmerTest']
Formula: ..1
   Data: ..2
REML criterion at convergence: 14026.44
Random effects:
 Groups   Name        Std.Dev.
 sid      (Intercept) 0.0000
 Residual             0.9828
Number of obs: 5000, groups:  sid, 50
Fixed Effects:
(Intercept)            Z
  -0.013930    -0.008804
optimizer (nloptwrap) convergence code: 0 (OK) ; 0 optimizer warnings; 1 lme4 warnings

$result$error
NULL
```

```
$output
[1] ""

$warnings
character(0)

$messages
[1] "boundary (singular) fit: see help('isSingular')\n"
```

The resulting object has the estimation results buried inside it. Even though the result is a bit of a mess, this structure provides all the pieces that we need, including any errors, warnings, and other output.

We can then carefully unpack what we get back in our estimation function, as so:

```
if ( is.null( M1$result$result ) ) {
  # we had an error!
  error = M1$result$error

} else {
  # We did not.
  error = NULL
  # Now plug in the same code as above
}
# etc.
```

Fortunately, once we have written all this code, we can tuck it inside our estimation function, and then forget about it. Once written, applying the function will produce a tidy table of results that we can easily analyze.

Exercises

Exercise 7.1 (More Heteroskedastic ANOVA)**.** In the classic simulation by Brown and Forsythe (1974), they not only looked at the performance of the homoskedastic ANOVA-F test and Welch's heteroskedastic-F test, they also proposed their own new hypothesis testing procedure.

1. This exercise continues the running example introduces in Section 5.2. Write a function that implements the Brown-Forsythe F* test (the BFF* test!) as described on p. 130 of Brown and Forsythe (1974), using the following code skeleton:

```
BF_F <- function( data ) {

  # fill in the guts here

  return(pval)
}
```

Run the following code to check that your function produces a result:

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

2. Try calling your `BF_F` function on a variety of datasets of different sizes and shapes, to make sure it works. What kinds of datasets should you test out?

3. Add the BFF* test into the output of `ANOVA_Welch_F()` (see Section 5.2) by calling your `BF_F()` function inside the body of `ANOVA_Welch_F()`.[1]

4. The `onewaytests` package implements a variety of different hypothesis testing procedures for one-way ANOVA. Validate your `Welch_ANOVA_F()` function by comparing the results to the output of the relevant functions from `onewaytests`.

Exercise 7.2 (Contingent testing)**.** In the one-way ANOVA problem, one approach that an analyst might think to take is to conduct a preliminary significance test for heterogeneity of variances (such as Levene's test or Bartlett's test), and then report the p-value from the homoskedastic ANOVA F test if variance heterogeneity is not detected but the p-value from the BFF* test if variance heteogeneity is detected. Modify the `Welch_ANOVA_F()` function to return the p-value from this contingent BFF* test in addition to the p-values from the (non-contingent) ANOVA-F, Welch, and BFF* tests. Include an input option that allows the user to control the α level of the preliminary test for heterogeneity of variances, as in the following skeleton.

```
Welch_ANOVA_F <- function( data , pre_alpha = .05) {
  # preliminary test for variance heterogeneity
```

[1] Note that Exercise 5.5 had you do this using an R package; see next exercise below for using that package to validate your code.

```
  # compute non-contingent F tests for group differences
  # compute contingent test
  # compile results
  return(result)
}
```

Exercise 7.3 (Check the cluster-RCT functions)**.** Section 7.2 presented functions implementing several different strategies for estimating an average treatment effect from a cluster-randomized trial. Write some code to validate these functions by comparing their output to the results of other tools for doing the same calculation. Use one or more datasets simulated with `gen_cluster_RCT()`. For each of these tests, you will need to figure out the appropriate syntax by reading the package documentation.

1. For `analysis_MLM()`, check the output by fitting the same model using `lme()` from the `nlme()` package or `glmmTMB()` from the package of the same name.

2. For `analysis_OLS()`, check the output by fitting the linear model using the base R function `lm()`, then computing standard errors using `vcovCL()` from the `sandwich` package. Also compare the output to the results of feeding the fitted model through `coef_test()` from the `clubSandwich` package.

3. For `analysis_agg()`, check the output by aggregating the data to the school-level, fitting the linear model using `lm()`, and computing standard errors using `vcovHC()` from the `sandwich` package.

Exercise 7.4 (Extending the cluster-RCT functions)**.** Exercise 6.10 from Chapter 6 asked you to extend the data-generating function for the cluster-randomized trial to include generating a student-level covariate, X, that is predictive of the outcome. Use your modified function to generate a dataset.

1. Modify the estimation functions from Section 7.2 to use models that include the covariate as a predictor. Your function should return three estimates: an MLM version (`analysis_MLM_cov()`), an aggregated version (`analysis_agg_cov()`), and an OLS version (`analysis_OLS_cov()`. Wrap all three in a final function `analyze_data_cov()` that applies all three methods to the same dataset and returns a tidy dataframe of results. Test that your functions work as expected.

2. Further extend the functions to include an input argument for the set of predictors to be included in the model, as in the following skeleton for the multi-level model estimator:

```
analysis_MLM <- function(dat, predictors = "W") {

}

analysis_MLM( dat )
analysis_MLM( dat, predictors = c("W","X"))
analysis_MLM( dat, predictors = c("W","X", "X:W"))
```

Hint: Check out the `reformulate()` function, which makes it easy to build formulas for different sets of predictors. Alternately, you could instead design the function to take a formula that defines the set of predictors, as in `predictors = ~ W + X + W:X`. With this approach, you will need to wrangle the formula into a useable form.

Exercise 7.5 (Contingent estimator processing)**.** In Section 7.4.2 we developed a version of `analysis_MLM()` that fell back on OLS regression in the event that `lmer()` produced any error. The implementation that we demonstrated is not especially smart. For one, it does not anticipate that the error will occur. For another, if we are using this function as one of several estimation strategies, it will require fitting the OLS regression multiple times. Can you fix these problems?

1. Revise `analysis_MLM_contingent()` so that it checks whether all clusters are single observations *before* fitting the multilevel model. If all clusters are singletons, skip the model-fitting step, return an `NA` values for the point estimator, standard error, and p-value, and return an informative message in the `error` variable. Test that the function works as expected.

2. Revise `analyze_data()` to use your new version of `analysis_MLM_contingent()`. The revised function will now sometimes return `NA` values for the `MLM` results. To implement the strategy of falling-back on OLS regression, add some code in `analyze_data()` that replaces any `NA` values for the MLM results with corresponding results of `analysis_OLS()`. Test that the function is working as expected (in this case, if the MLM fails, the final result would have the same estimates for MLM and OLS as two separate rows). Compared to having `analysis_MLM_contingent()` run OLS internally, writing code this way improves efficiency because we do not fit the OLS regression multiple times on the same data; such an improvement could make a simulation faster to run.

Exercise 7.6 (Estimating 3-parameter item response theory models)**.** Exercise 6.11 asked you to write a data-generating function for the 3-parameter

IRT model described in described in Section 6.7. Use your function to generate a large dataset. Using functions from the {ltm}, {mirt}, or {TAM} packages, *estimate* the parameters of the 3-parameter IRT model based on the simulated dataset.

1. Write a function to *estimate* a 3-parameter IRT model and return a dataset containing estimates of the item characteristics $(\alpha_m, \beta_m, \gamma_m)$.

2. Add an option to the function to allow the user to specify a shared known value `gamma` for the γ_m. Your function would then assume all items had that value if `gamma` is not `NULL`, but would estimate the γ_m if `gamma = NULL`.

3. Create a graphic showing how the item parameter estimates compare to the true item characteristics for a single run of your DGP, with $M = 20$ items and $N = 1000$ respondents.

4. Write a function or set of functions that use the {mirt} package to estimate and return the student-level ability estimates $\theta_1, ..., \theta_N$ along with their standard errors, given a test dataset.

Exercise 7.7 (Meta-regression with selective reporting)**.** Exercise 6.15 asked you to write a data-generating function for the Vevea and Hedges (1995) selection model. The {metafor} package includes a function for fitting this model (as well as a variety of other selection models). Here is an example of the syntax for estimating this model, using a dataset from the {metadat} package:

```
library(metafor)
data("dat.assink2016", package = "metadat")

# rename variables and tidy up
dat <-
  dat.assink2016 %>%
  mutate( si = sqrt(vi) ) %>%
  rename( Ti = yi, s_sq = vi, Xi = year )

# fit a random effects meta-regression model
rma_fit <- rma.uni(yi = Ti, sei = si, mods = ~ Xi, data = dat)
# fit two-step selection model
selmodel(rma_fit, type = "step", steps = c(.025, .500))
```

```
Mixed-Effects Model (k = 100; tau^2 estimator: ML)

tau^2 (estimated amount of residual heterogeneity): 0.2314 (SE = 0.0500)
tau (square root of estimated tau^2 value):         0.4810
```

```
Test for Residual Heterogeneity:
LRT(df = 1) = 349.3718, p-val < .0001

Test of Moderators (coefficient 2):
QM(df = 1) = 42.1433, p-val < .0001

Model Results:

         estimate      se     zval    pval    ci.lb    ci.ub
intrcpt    0.4149  0.1013   4.0942  <.0001   0.2163   0.6135
Xi        -0.0782  0.0120  -6.4918  <.0001  -0.1018  -0.0546

intrcpt  ***
Xi       ***

Test for Selection Model Parameters:
LRT(df = 2) = 1.7814, p-val = 0.4104

Selection Model Results:

                     k  estimate      se     zval    pval
0     < p <= 0.025  59    1.0000     ---      ---     ---
0.025 < p <= 0.5    23    0.6990  0.2307  -1.3049  0.1919
0.5   < p <= 1      18    0.5027  0.2743  -1.8132  0.0698
                     ci.lb   ci.ub
0     < p <= 0.025     ---     ---
0.025 < p <= 0.5    0.2470  1.1511
0.5   < p <= 1      0.0000  1.0403  .

---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The `selmodel()` function can also be used to fit selection models in which one or both of the selection parameters are fixed to user-specified values. For example:

```
# Fixing lambda_1 = 0.5, lambda_2 = 0.2
fix_both <- selmodel(rma_fit, type = "step", steps = c(.025, .500),
                     delta = c(1, 0.5, 0.2))

# Fixing lambda_1 = 0.5, estimating lambda_2
fix_one <- selmodel(rma_fit, type = "step", steps = c(.025, .500),
                    delta = c(1, 0.5, NA))
```

1. Write an estimation function that fits the selection model and

returns estimates, standard errors, and confidence intervals for each of the model parameters $(\beta_0, \beta_1, \tau, \lambda_1, \lambda_2)$.

2. The `selmodel()` fitting function sometimes returns errors, as in the following example:

```
dat_sig <- dat %>% filter(Ti / si > 0)
rma_pos <- rma.uni(yi = Ti, sei = si, mods = ~ Xi, data = dat_sig)
# fit two-step selection model
sel_fit <- selmodel(rma_pos, type = "step", steps = c(.025, .500))
```

Modify your estimation function to catch and return warnings such as these. Write code demonstrating that the function works as expected.

3. The `selmodel()` throws warnings when the dataset contains no observations with p_i in one of the specified intervals. Modify your estimation function to set the selection probability for an interval to $\lambda_1 = 0.1$ if there are no p_i values between .025 and .500 and to set $\lambda_2 = 0.1$ if there are no p_i values larger than .500. Write code demonstrating that the function works as expected.

8

Running the Simulation Process

In the prior two chapters we saw how to write functions that generate data according to a particular model and functions that implement data-analysis procedures on simulated data. The next step in building a simulation involves putting these two pieces together, running the DGP function and the data-analysis function repeatedly to obtain results (in the form of point estimates, standard errors, confidence intervals, p-values, or other quantities) from many replications of the whole process.

As with most things R-related, there are many different techniques that can be used to repeat a set of calculations over and over. In this chapter, we demonstrate two key techniques for doing so. We then explain how to ensure reproducibility of simulation results by setting the seed used by R's random number generator.

8.1 Repeating oneself

Suppose that we want to repeatedly generate bivariate Poisson data and then use the data to estimate the correlation between the variates. We saw a DGP function for the bivariate Poisson in Section 6.3, and an estimation function in Section 7.1. To produce a simulated correlation coefficient, we need to run these two functions in turn:

```
dat <- r_bivariate_Poisson( N = 30, rho = 0.4, mu1 = 8, mu2 = 14 )
r_and_z(dat)
```

```
          r         z      CI_lo     CI_hi
1 0.2548057 0.2605456 -0.1161235 0.5633593
```

To execute a simulation with these components, we need to repeat this set of calculations over and over. R has many different functions for doing exactly this. As one of many alternatives, the `simhelpers` package includes a function called `repeat_and_stack()`, which can be used to evaluate an arbitrary expression many times over. We can use it to generate five replications of our correlation coefficient:

```
library(simhelpers)
repeat_and_stack(
  n = 5,
  {
    dat <- r_bivariate_Poisson( N = 30, rho = 0.4, mu1 = 8, mu2 = 14 )
    r_and_z(dat)
  },
  id = "rep",
  stack = TRUE
)
```

```
  rep          r          z       CI_lo     CI_hi
1   1  0.3921400  0.4143264  0.03711412 0.6592702
2   2  0.6322427  0.7451436  0.35219584 0.8083809
3   3  0.3154004  0.3265311 -0.05062085 0.6067277
4   4  0.5272205  0.5862878  0.20609779 0.7458265
5   5 -0.1091544 -0.1095911 -0.45166205 0.2613941
```

The first argument specifies the number of times to repeat the calculation. The second argument is an R expression that will be evaluated. The expression is wrapped in curly braces (`{}`) because it involves more than a single line of code. Including the option `id = "rep"` returns a dataset that includes a variable called `rep` to identify each replication of the process. Setting the option `stack = TRUE` will stack up the output of each expression into a single tibble, which will facilitate further calculations on the results. Explicitly setting this option is not necessary because it is `TRUE` by default; setting `stack = FALSE` will return the results in a list rather than a tibble (try this for yourself to see!).

There are many other functions that work very much like `repeat_and_stack()`, including the base-R function `replicate()` and the now-deprecated function `rerun()` from `{purrr}`. The functions in the `map()` family from `{purrr}` can also be used to do the same thing as `repeat_and_stack()`. See Appendix Section A.1 for more discussion of these options.

8.2 One run at a time

A slightly different technique for running multiple replications of a process is to first write a function that executes a single run of the simulation, and then repeatedly evaluate that single function. For instance, here is a function that stitches together the two steps in the bivariate-Poisson correlation simulation:

```
one_bivariate_Poisson_r <- function(N, rho, mu1, mu2) {
  dat <- r_bivariate_Poisson( N = N, rho = rho, mu1 = mu1, mu2 = mu2 )
  res <- r_and_z(dat)
  return(res)
}
```

Calling the function produces a nicely formatted set of results:

```
one_bivariate_Poisson_r(N = 30, rho = 0.4, mu1 = 8, mu2 = 14)
```

```
          r         z       CI_lo     CI_hi
1 0.3227466 0.3347101 -0.04245963 0.6118702
```

We can then evaluate the function over and over using `repeat_and_stack()`:

```
repeat_and_stack(
  n = 5,
  one_bivariate_Poisson_r( N = 30, rho = 0.4, mu1 = 8, mu2 = 14 ),
  id = "rep"
)
```

```
  rep         r         z          CI_lo     CI_hi
1   1 0.4738988 0.5150864  0.1370238059 0.7125189
2   2 0.4439057 0.4770846  0.0995584095 0.6932987
3   3 0.4486475 0.4830057  0.1054173092 0.6963612
4   4 0.3610627 0.3781074  0.0009121595 0.6383018
5   5 0.1648357 0.1663534 -0.2077721235 0.4956695
```

This technique of wrapping the data-generating function and estimation function inside of another function might strike you as a bit cumbersome because the wrapper is only two lines of code and writing it requires repeating many of the function argument names when calling the data-generating function (`N = N, rho = rho`, etc.). However, the wrapper technique can be useful for more complicated simulations, such as those that involve comparison of multiple estimation methods.

Consider the cluster-randomized experiment case study presented in Section 6.6 and Section 7.2. In this simulation, we are interested in comparing the performance of three different estimation methods: a multi-level model, a linear regression with clustered standard errors, and a linear regression on the data aggregated to the school level. A single replication of the simulation entails generating a dataset and then applying three different estimation functions to it. Here is a function that takes our simulation parameters and runs a single trial of the full process:

```
one_run <- function(
  n_bar = 30, J = 20, gamma_1 = 0.3, gamma_2 = 0.5,
  sigma2_u = 0.20, sigma2_e = 0.80, alpha = 0.75, p = 0.5
```

```
) {

  dat <- gen_cluster_RCT(
    n_bar = n_bar, J = J, gamma_1 = gamma_1, gamma_2 = gamma_2,
    sigma2_u = sigma2_u, sigma2_e = sigma2_e, alpha = alpha,
    p = p
  )
  MLM <- analysis_MLM( dat )
  LR <- analysis_OLS( dat )
  Agg <- analysis_agg( dat )

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

We have added a bunch of defaults to our function, so that we can run it without having to remember all the various input parameters. When we call the function, we get a nicely structured table of results:

```
one_run( n_bar = 30, J = 20, alpha=0.5 )
```

```
# A tibble: 3 x 4
  method ATE_hat SE_hat p_value
  <chr>    <dbl>  <dbl>   <dbl>
1 MLM      0.308  0.176  0.0969
2 LR       0.316  0.169  0.0790
3 Agg      0.309  0.178  0.0997
```

We organize the output in a tibble to make it easier to do subsequent data processing and analysis. The results for each method are organized in separate lines. For each method, we record the impact estimate, its 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 for keeping track of which results come from which estimation.

Once we have a function to execute a single run, we can produce multiple results using `repeat_and_stack()`:

```
R <- 1000
ATE <- 0.30
runs <- repeat_and_stack( R,
                          one_run( n_bar = 30, J = 20, gamma_1 = ATE ),
                          id = "runID")
```

Setting `id = "runID"` lets us keep track of which iteration number produced which result. Once our simulation is complete, we save our results to a file so that we can avoid having to re-run the full simulation if we want to explore the results in some future work session.

We now have results for each of our estimation methods applied to each of 1000 generated datasets. The next step is to evaluate how well the estimators did. For example, we will want to examine questions about bias, precision, and accuracy of the three point estimators. In Chapter 9, we will look systematically at ways to quantify the performance of estimation methods.

8.3 Reparameterizing

In Section 6.6.4, we discussed how to index the DGP of the cluster-randomized experiment using an intra-class correlation (ICC) instead of using two separate variance components. This type of re-parameterization can be handled as part of writing a wrapper function for executing the DGP and estimation procedures. Here is a revised version of `one_run()`[1], which also renames some of the more obscure model parameters using terms that are easier to interpret:

```
one_run <- function(
  n_bar = 30, J = 20, ATE = 0.3, size_coef = 0.5,
  ICC = 0.2, alpha = 0.75, p = 0.5
) {
  stopifnot( ICC >= 0 && ICC < 1 )

  dat <- gen_cluster_RCT(
    n_bar = n_bar, J=J, gamma_1 = ATE, gamma_2 = size_coef,
    sigma2_u = ICC, sigma2_e = 1-ICC, alpha = alpha, p = p
  )

  quiet_analyze_data(dat)
}
```

For details on `quiet_analyze_data()`, see Section 7.4.1. Also see Section 7.2 for how `quiet_analyze_data()` bundles the three different estimation methods we are evaluating.

Careful use of the foundational elements such as our data generating function can make interpreting our final results much more straightforward. In the revised `one_run()` function, for example, transforming the `ICC` input parameter into the parameters used by `gen_cluster_RCT()` maintains the effect size interpretation of our simulation. We have not modified `gen_cluster_RCT()` at all: instead, we specify the parameters of the DGP function in terms of the parameters we want to directly control in the simulation, and then translate

[1] Note the use of `stopifnot` in the body of `one_run()`. It is wise to ensure our parameter transforms are all reasonable, so we do not get unexplained errors or strange results later on. It is best if your code fails as soon as possible! Otherwise debugging can be quite hard.

to the low-level parametersin the `one_run()` call.

We can then wrap both the reparameterization and replication into a function that runs an entire simulation:

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

  res <-
    simhelpers::repeat_and_stack( reps, {
      one_run( n_bar = n_bar, J = J, ATE = ATE,
               ICC = ICC, p = p,
               size_coef = size_coef,
               alpha = alpha )
    }) %>%
    bind_rows( .id="runID" )

  res
}
```

Using it is then quite simple:

```
runs <- run_CRT_sim( reps = 1000,
             n_bar = 30, J = 20,
             ATE = 0.3, ICC = 0.2,
             size_coef = 0.5, alpha = 0.75 )
saveRDS( runs, file = "results/cluster_RCT_simulation.rds" )
```

Our simulation function is specified with easy-to-interpret parameters (`ATE`, `ICC`) rather than the raw variance components. It runs simulations in effect size units, and provides "knobs" to the simulation that are directly interpretable.

8.4 Bundling simulations with `simhelpers`

The techniques that we have demonstrated for repeating a set of calculations each involve a very specific coding pattern, which will often have the same structure even if the details of the data-generating model or the names of the input parameters are very different from the examples we have presented. The `simhelpers` package provides a function `bundle_sim()` that abstracts this common pattern and allows you to automatically stitch together (or "bundle") a DGP function and an estimation function, so that they can be run once or multiple times. Thus, `bundle_sim()` provides a convenient alternative to

writing your own `one_run()` and `run_CRT_sim()` functions, thereby saving a bit of typing and providing some security against bugs creeping into your code.

The `bundle_sim()` function takes a DGP function and an estimation function as inputs and gives us back a new function that will run a simulation using whatever parameters we give it. The `bundle_sim()` function is a "factory" in that it creates a new function on the fly. Here is a basic example, where we create a function for running our simulation to evaluate how well our Pearson correlation coefficient confidence interval works on a bivariate Poisson distribution:

```
sim_r_Poisson <- bundle_sim( f_generate = r_bivariate_Poisson,
                             f_analyze = r_and_z,
                             id = "rep" )
```

The optional argument `id = "rep"` causes the built function to generate a variable called `rep` with a unique identifier for each replication of the simulation. We can use the newly created function as so:

```
sim_r_Poisson( 4, N = 30, rho = 0.4, mu1 = 8, mu2 = 14 )
```

```
  rep         r         z       CI_lo     CI_hi
1   1 0.3236179 0.3356829 -0.04148850 0.6124785
2   2 0.4264834 0.4555903  0.07823489 0.6819689
3   3 0.5197735 0.5760293  0.19625459 0.7412394
4   4 0.5077860 0.5597421  0.18054580 0.7338119
```

To create this simulation function, `bundle_sim()` examined `r_bivariate_Poisson()`, figured out what its input arguments were, and made sure that the simulation function includes the same input arguments. You can see the full set of arguments for `sim_r_Poisson()` by evaluating it with `args()`:

```
args(sim_r_Poisson)
```

```
function (reps, N, mu1, mu2, rho = 0, seed = NA_integer_)
NULL
```

In addition to the expected arguments from `r_bivariate_Poisson()`, the function has some additional parameters. First is `reps`, which controls the number of times that the simulation process will be evaluated. There is also `seed`, which ensures reproducible code; we discuss seeds in Section 8.5.

The `bundle_sim()` function requires specifying a DGP function and a *single* estimation function, with the data as the first argument. If you have multiple estimators, you would need to wrap them into a single method, just as we did with `quiet_analyze_data()`:

```
sim_cluster_RCT <- bundle_sim(gen_cluster_RCT, quiet_analyze_data, id = "runID")
```

To use the newly created simulation function, we call it by specifying the number of replications desired and any relevant input parameters.

```
runs <- sim_cluster_RCT( reps = 2,
                         J = 20, n_bar = 30, alpha = 0.5,
                         gamma_1 = ATE, gamma_2 = 0.5,
                         sigma2_u = 0.2, sigma2_e = 0.8 )
```

The `bundle_sim()` function is just a convenient way to create a function that pieces together the steps in the simulation process. This is especially useful when the component functions include many input parameters. Think of it as a not overly smart coding assistant who will take some basic directions and give you some code back. The function has several further features, which we will demonstrate in subsequent chapters.

One limitation is `bundle_sim()` produces a function with input names that exactly match the inputs of the DGP function that we give it. It is not possible to re-parameterize or change argument names, as we did with `one_run()` in Section 8.3. See Exercise 8.3 for further discussion of how to work around this constraint.

8.5 Seeds and pseudo-random number generators

In prior chapters, we used built-in functions to generate random numbers and wrote our own data-generating functions that produce artificial data following a specific random process. With either type of function, re-running it with the exact same input parameters will produce different results. For instance, running the `rchisq` function with the same set of inputs will produce two different sequences of χ^2 random variables:

```
c1 <- rchisq(4, df = 3)
c2 <- rchisq(4, df = 3)
rbind(c1, c2)
```

```
        [,1]      [,2]      [,3]      [,4]
c1 0.7716889 0.6487126 0.2029543  2.470464
c2 2.2455424 3.8978796 2.8435750 10.983158
```

If you run the same code as above, you will get different results from these. Likewise, running the bivariate Poisson function from Section 6.3 or the `sim_r_Poisson()` function from Section 8.4 multiple times will produce different datasets:

```
dat_A <- r_bivariate_Poisson(20, rho = 0.5, mu1 = 4, mu2 = 7)
dat_B <- r_bivariate_Poisson(20, rho = 0.5, mu1 = 4, mu2 = 7)
identical(dat_A, dat_B)
```

```
[1] FALSE
```

```
sim_A <- sim_r_Poisson( 10, N = 30, rho = 0.4, mu1 = 8, mu2 = 14)
sim_B <- sim_r_Poisson( 10, N = 30, rho = 0.4, mu1 = 8, mu2 = 14)
identical(sim_A, sim_B)
```

```
[1] FALSE
```

Of course, this is the intended behavior of these functions, but it has an important consequence that needs some care and attention. Using functions like `rchisq()`, `r_bivariate_Poisson()`, or `r_bivariate_Poisson()` in a simulation study means that the results will not be fully reproducible. If we run the simulation code and write up our results, and then someone else tries to run the same simulation using the very same code, they will not get the very same results. This undermines transparency and makes it hard to verify if a set of presented results really came from a given codebase.

When using DGP functions for simulations, it is useful to be able to exactly control the process of generating random numbers. This is much more feasible than it sounds: although Monte Carlo simulations are random in theory, computers are deterministic. When we use R to generate what we have been referring to as "random numbers," the functions produce what are actually *pseudo-random* numbers. Pseudo-random numbers are generated from chains of mathematical equations designed to produce sequences of numbers that appear random, but actually follow a deterministic sequence. Each subsequent random number is a calculated by starting from the previously generated value (i.e., the current state of the random number generator), applying a complicated function, and storing the result (i.e., updating the state). The numbers returned by the generator form a chain that, ideally, cycles through an extremely long list of values in a way that looks stochastic and unpredictable.

The state of the pseudo-random number generator is shared across different functions that produce pseudo-random numbers, so it does not matter if we are generating numbers with `rnorm()` or `rchisq()` or `r_bivariate_Poisson()`. Each time we ask for a random number from the generator, its state is updated. Functions like `rnorm()` and `rchisq()` all call the low-level generator and then transform the result to be of the correct distribution.

Because the generator is actually deterministic, we can control its output by specify a starting value or initial state. In R, the state of the random number generator can be controlled by setting what its known as the seed. The `set.seed()` function allows us to specify a seed value, so that we can exactly reproduce a calculation. For example,

```
set.seed(6)
c1 <- rchisq(4, df = 3)
set.seed(6)
c2 <- rchisq(4, df = 3)
rbind(c1, c2)
```

```
       [,1]    [,2]     [,3]     [,4]
c1 2.575556 0.93847 8.062264 3.685932
c2 2.575556 0.93847 8.062264 3.685932
```

The sequences are the same!

Similarly, we can set the seed and run a series of calculations involving multiple functions that make use of the random number generator:

```
# First time
set.seed(6)
c1 <- rchisq(4, df = 3)
dat_A <- r_bivariate_Poisson(20, rho = 0.5, mu1 = 4, mu2 = 7)

# Exactly reproduce the calculations
set.seed(6)
c2 <- rchisq(4, df = 3)
dat_B <- r_bivariate_Poisson(20, rho = 0.5, mu1 = 4, mu2 = 7)

bind_rows(A = r_and_z(dat_A), B = r_and_z(dat_B), .id = "Rep")
```

```
  Rep         r         z      CI_lo     CI_hi
1   A 0.2990194 0.3084424 -0.1653856 0.6548844
2   B 0.2990194 0.3084424 -0.1653856 0.6548844
```

When writing code, you can pass the seed as a parameter, such as the following:

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

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

A `seed` argument allows the user to set a seed value before running the full simulation. By default, the `seed` argument is `NULL` and so the current seed is not modified, but if you set the seed, you will get the same run each time.

The `bundle_sim()` function, demonstrated in Section 8.4, produces a function

with a `seed` seed argument that works just as in the example above. Specifying an integer-valued seed will make the results exactly reproducible:

```
sim_A <- sim_r_Poisson( 10, N = 30, rho = 0.4, mu1 = 8, mu2 = 14,
                        seed = 25)
sim_B <- sim_r_Poisson( 10, N = 30, rho = 0.4, mu1 = 8, mu2 = 14,
                        seed = 25)
identical(sim_A, sim_B)
```

```
[1] TRUE
```

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. Attending to reproducibility allows us to easily check if we are running the same code that generated a set of results. For instance, try running the previous blocks of code on your machine; if you set the seed to the same value as we did, you should get identical output.

Setting seeds is also very helpful for debugging. Suppose we had an error that showed up in one of a thousand replications, causing the simulation to crash sometimes. If we set a seed and find that the code crashes, we can debug and then rerun the simulation. If it now runs without error, we know we fixed the problem. If we had not set the seed, we would not know if we were just getting (un)lucky, and avoiding the error by chance.

8.6 Conclusions

The act of running a simulation is not itself the goal of a simulation study. Rather, it is only the procedural mechanism by which we generate many realizations of a data analysis procedure under a specified data-generating process. As we have seen in this chapter, doing so requires carefully stitching together the one-two step of data generation and analysis, repeating that combined process many times, and then organizing the resulting output in a form that can be easily and systematically evaluated. Different coding strategies can be used to accomplish this, but they all serve the same purpose: to make repeated evaluation of a procedure reliable, transparent, and easy to extend. Finally, because simulation studies rely on stochastic data generation, it is critical that we pay careful attention to reproducibility. By controlling the pseudo-random number generator through explicit use of seeds, we can ensure that simulation results can be exactly reproduced, checked, and debugged. This reproducibility is essential for both scientific transparency and practical development of simulation code.

With the tools of this chapter in place, we are now positioned to move from

running simulations to evaluating what they tell us about estimator behavior—the focus of Chapter 9.

Exercises

Exercise 8.1 (Welch simulations)**.** In Exercise 7.1 of Chapter 7, you made a new `BF_F` function for the Welch simulation. Update the Welch simulation code to include your `BF_F` function, and then generate simulation results for this additional estimator. See Chapter 5 for the original simulation and overall context.

Exercise 8.2 (Compare sampling distributions of Pearson's correlation coefficients)**.** In Section 8.4, we made a bundled simulation function `sim_r_Poisson()` from a DGP function `r_bivariate_Poisson()` and analysis function `r_and_z()`.

1. Use `sim_r_Poisson()` to generate 5000 replications of the Fisher-z-transformed correlation coefficient under a bivariate Poisson distribution with $\rho = 0.7$ for a sample of $N = 40$ observations. You pick the remaining parameters.

2. Create a bundled simulation function by combining the data-generating function from Exercise 6.5 or Exercise 6.6 with the `r_and_z()` estimation function. Run the function to generate 5000 replications of the Fisher-z-transformed correlation coefficient under a bivariate negative binomial distribution, with the same parameter values as in part 1.

3. Create a plot that compares the sampling distribution of r for both data generating processes. We recommend using a density plot.

Exercise 8.3 (Reparameterization, redux)**.** In Section 8.3, we illustrated how the `one_run()` simulation wrapper function could be tweaked in order to reparameterize the model in terms of a single intra-class correlation rather than two variance parameters. But what if you want to avoid having to write your own simulation wrapper and instead use `bundle_sim()`? Revise `gen_cluster_RCT()` to accomplish the reparameterization, then use `simhelpers::bundle_sim()` to bundle it with `analyze_data()`.

Exercise 8.4 (Fancy clustered RCT simulations)**.** In Exercise 6.10, your task was to write a data-generating function for a cluster-randomized trial that includes a baseline covariate. Then in Exercise 7.4, your task was to create an estimation function that could be applied to data from such a trial.

1. Using your work from these exercises, create a bundled simulation function `one_run_cov()` that combines the data-generating function and the estimation function. Use both `analyze_data()` and `analyze_data_cov()` to get estimates for all three approaches both with and without covariate adjustment.

2. Use the resulting function to simulate results from 1000 replications of a cluster-randomized trial, each analyzed six ways (all three approaches with and without covariate adjustment).

3. Create a plot that shows the six sampling distribution in a way that makes it easy to compare the sampling distributions.

9

Performance Measures

Once we run a simulation, we end up with a pile of results to sort through. For example, Figure 9.1 depicts the distribution of average treatment effect estimates from the cluster-randomized experiment simulation we generated in Section 8.3. There are three different estimators, each with 1000 replications. Each histogram is an approximation of the *sampling distribution* of the estimator, meaning its distribution across repetitions of the data-generating process. Visually, we see the three estimators look about the same—they sometimes estimate too low, and sometimes estimate too high. We also see linear regression is shifted up a bit, and aggregation and MLM might have a few more outlying values. These observations are initial impressions of how well each estimator performs—that is, how close the estimators tend to get to the truth. In this chapter, we look at how to more precisely quantify and compare specific aspects of estimator performance.

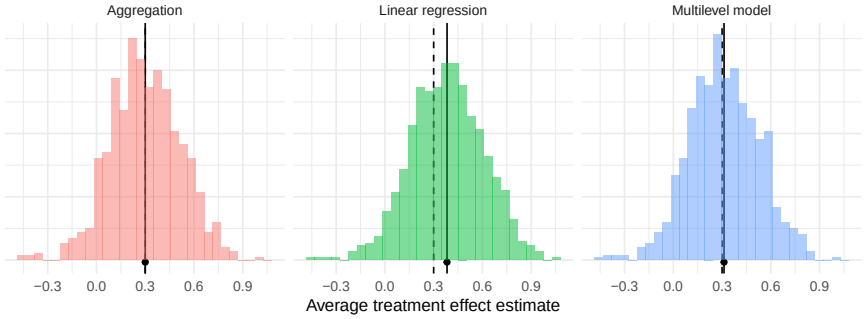


Figure 9.1: Sampling distribution of average treatment effect estimates from a cluster-randomized trial with a true average treatment effect of 0.3 (dashed line).

A *performance measure* summarizes an estimator's sampling distribution to describe how an estimator would behave on average if we were to repeat the data-generating process an infinite number of times. For example, the bias of an estimator measures the central tendency of the sampling distribution, capturing whether we are systematically off, on average, from the true parameter value.

In Figure 9.1, the black dashed lines mark the true average treatment effect of 0.3 and the solid lines with the dot mark the means of the estimators. The horizontal distance between the solid and dashed lines is the bias. This distance is nearly zero for the aggregation estimator and the multilevel model estimator, but larger for the linear regression estimator. Our performance measure of bias would be near zero for aggregation and multilevel modeling, but around 0.08 for linear regression.

Many performance measures exist. For point estimates, conventional performance measures include bias, variance, and root mean squared error. If the point estimates come with corresponding standard errors, then we may also want to evaluate how accurately the standard errors represent the true uncertainty of the point estimators; one conventional performance measures for capturing this is the relative bias of the variance estimator. For procedures that produce confidence intervals or other types of interval estimates, conventional performance measures include the coverage rate and average interval width. Finally, for inferential procedures that involve hypothesis tests (or more generally, classification tasks), conventional performance measures include Type I error rates and power. We describe each of these measures in Sections 9.1 through 9.4.

For any particular measure, the actual performance of an estimator depends on the sampling distribution, which in turn depends on the data generating process under which it is evaluated. We will therefore need to use some statistical notation to define properties of this sampling distribution. In the following, we define many of the most common measures in terms of standard statistical quantities such as the mean, variance, and other moments of the sampling distribution. Notation-wise, for a random variable T (e.g., for some estimator T), we will use the expectation operator $\mathbb{E}(T)$ to denote the mean, $\mathbb{M}(T)$ to denote the median, $\text{Var}(T)$ to denote the variance, and $\Pr()$ to denote probabilities of specific outcomes, all with respect to the sampling distribution of T. We also use $\mathbb{Q}_p(T)$ to denote the p^{th} quantile of a distribution, which is the value x such that $\Pr(T \leq x) = p$. With this notation, the median of a continuous distribution is equivalent to the 0.5 quantile: $\mathbb{M}(T) = \mathbb{Q}_{0.5}(T)$. Using this notation we can write, for example, that the bias of T is $\mathbb{E}(T) - \theta$, where θ is the target parameter.

For some simple combinations of data-generating processes and data analysis procedures, it may be possible to derive exact mathematical formulas for some performance measures (such as exact mathematical expressions for the bias and variance of the linear regression estimator). But for many problems, the math is difficult or intractable—which is partly why we do simulations in the first place. However, simulations do not produce the *exact* sampling distribution or give us *exact* values of performance measures. Instead, simulations generate *samples* (usually large samples) from the the sampling distribution, and we can use these to *estimate* the performance measures of interest. In

Figure 9.1, for example, we estimated the bias of each estimator by taking the mean of 1000 observations from its sampling distribution. If we were to repeat the whole simulation (with a different seed), then our estimates of bias would shift.

It is important to track this *Monte Carlo uncertainty*, i.e., uncertainty arising from using only a finite number of replications of the Monte Carlo simulation. One way to quantify this simulation uncertainty is with the *Monte Carlo Standard Error (MCSE)*, which is the standard error of a performance estimate given a specific number of replications. Just as when we analyze real data, we can apply statistical techniques to estimate the MCSE and to generate confidence intervals for the performance measures themselves.

The magnitude of the MCSE is driven by how many replications we use: if we only use a few, we will have noisy estimates of performance and large MCSEs; if we use millions of replications, the MCSE will usually be tiny. It is important to keep in mind that the MCSE is not measuring anything about how our estimator performs. It only describes how precisely we have *estimated* its performance; MCSE is an artifact of how we conducted the simulation. The MCSE are also under our control: Given a desired MCSE, we can determine how many replications we would need to ensure our performance estimates have a MCSE of that size. Section 9.7 provides details about how to compute MCSEs for conventional performance measures, along with some discussion of general techniques for computing MCSE for less conventional measures.

9.1 Measures for Point Estimators

The most common performance measures used to assess a point estimator are **bias**, **variance**, and **root mean squared error**.

Bias compares the mean of the sampling distribution to the target parameter. Positive bias implies that the estimator tends to systematically over-state the quantity of interest, while negative bias implies that it systematically under-shoots the quantity of interest. If bias is zero (or nearly zero), we say that the estimator is unbiased (or approximately unbiased).

Variance (or its square root, the **true standard error**) describes the spread of the sampling distribution, or the extent to which it varies around its central tendency. All else equal, we would like estimators to have low variance (which means being more precise).

Root mean squared error (RMSE) is a conventional measure of the overall accuracy of an estimator, or its average degree of error with respect to the target parameter. For absolute assessments of performance, an estimator with

low bias, low variance, and thus low RMSE is desired. In making comparisons of several different estimators, one with lower RMSE is usually preferable to one with higher RMSE. If two estimators have comparable RMSE, then the estimator with lower bias would usually be preferable.

To define these quantities more precisely, consider a generic estimator T that is targeting a parameter θ. We call the target parameter the *estimand*. Formally, the bias, variance, and RMSE of T are then defined as

$$\begin{aligned} \text{Bias}(T) &= \mathbb{E}(T) - \theta, \\ \text{Var}(T) &= \mathbb{E}\left[(T - \mathbb{E}(T))^2\right], \\ \text{RMSE}(T) &= \sqrt{\mathbb{E}\left[(T-\theta)^2\right]}. \end{aligned} \tag{9.1}$$

In words, bias is the average estimate minus the truth: is it systematically too high or too low, on average? The variance is the average squared deviation of the estimator from its own mean: how much does it vary around its average estimate? Is it precise? RMSE is the square root of the average squared deviation of the estimator from the truth: how far off are we, on average, from the truth? Does it predict well?

These three measures are inter-connected. In particular, RMSE is the combination of (squared) bias and variance, as in

$$[\text{RMSE}(T)]^2 = [\text{Bias}(T)]^2 + \text{Var}(T). \tag{9.2}$$

When conducting a simulation, we do not compute these performance measures directly but rather must estimate them. In most cases, in running a simulation we generate a (typically large) series of R datasets, with θ being the true target parameter in each of them. To estimate performance, we analyze each dataset, obtaining a sample of estimates $T_1, ..., T_R$ drawn from the sampling distribution and then plug in sample quantities in place of the expectations and variances. Specifically, we estimate bias by taking

$$\widehat{\text{Bias}}(T) = \bar{T} - \theta, \tag{9.3}$$

where $\bar{T}$ is the arithmetic mean of the replicates, $\bar{T} = \frac{1}{R}\sum_{r=1}^{R} T_r$. We estimate variance by taking the sample variance of the replicates, as

$$S_T^2 = \frac{1}{R-1}\sum_{r=1}^{R}\left(T_r - \bar{T}\right)^2. \tag{9.4}$$

S_T (the square root of S_T^2) is an estimate of the *true* standard error, SE, of T, which is the standard deviation of the estimator across an infinite set of replications of the data-generating process.

It is important to note that in a simulation we will be able to see both the *true* standard error, measured across the simulation iterations, and the *estimated*

standard errors, one for each simulation iteration. Generally, when people say "Standard Error" they actually mean *estimated* Standard Error, ($\widehat{SE}$), as we generally calculate in a real data analysis (where we have only a single realization of the data-generating process). It is easy to forget that this standard error is itself an estimate of a parameter—the true SE—and thus has its own uncertainty. We usually prefer to work with the true SE S_T rather than the sampling variance S_T^2 because the former quantity has the same units as the target parameter.

Finally, the RMSE estimate can be calculated as

$$\widehat{\text{RMSE}}(T) = \sqrt{\frac{1}{R}\sum_{r=1}^{R}(T_r - \theta)^2}. \tag{9.5}$$

Often, people talk about the MSE (Mean Squared Error), which is just the square of RMSE. Just as the true SE is usually easier to interpret than the sampling variance, units of RMSE are easier to interpret than the units of MSE.

All of these performance measures depend on the scale of the parameter. For example, if an estimator is targeting a treatment impact in dollars, then the bias, SE, and RMSE of the estimator are all in dollars. (The variance and MSE would be in dollars squared; taking square roots puts these measures back on the more intepretable scale of dollars.)

In many simulations, the scale of the outcome is an arbitrary feature of the data-generating process, so the absolute magnitude of the performance measure is not meaningful. To ease interpretation of performance measures, it is useful to consider their magnitude relative to some reference amount. We generally recommend using the baseline level of variation in the outcome. For example, we might report a bias of 0.2σ, where σ is the standard deviation of the outcome in the population.

One way to ensure performance metrics are on a standardized scale is to generate data so the outcome has unit variance (i.e., we generate outcomes in *standardized units*). This automatically puts the bias, true standard error, and root mean squared error on the scale of standard deviation units, which can facilitate interpretation about what constitutes a meaningfully large bias or a meaningful difference in RMSE.

Finally, a non-linear transformation of a parameter will generally change a performance measure. For instance, suppose that θ measures the proportion of time that something occurs. This parameter could also be given on the log-odds (logit) scale. However, because the log-odds transformation is non-linear,

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

and so on.

9.1.1 Comparing the Performance of the Cluster RCT Estimation Procedures

We now demonstrate the calculation of performance measures for the point estimators of average treatment effects in the cluster-RCT example. We calculate performance measures to address the following questions:

- Is the estimator systematically off? (bias)
- Is it precise? (standard error)
- Does it predict well? (RMSE)

Let us see how the three estimators compare on these measures.

Are the estimators biased?

Bias is defined with respect to a target estimand. Here we assess whether our estimates are systematically different from the γ_1 parameter, which we defined in standardized units by setting the standard deviation of the student-level distribution of the outcome equal to one (see Section@sec-DGP-standardization). In our simulation, we generated data with a school-level ATE parameter of $\gamma_1 = 0.30$ SDs.

```
ATE <- 0.30

ClusterRCT_runs %>%
  group_by( method ) %>%
  summarise(
    mean_ATE_hat = mean( ATE_hat ),
    bias = mean( ATE_hat ) - ATE
  )
```

```
# A tibble: 3 x 3
  method mean_ATE_hat      bias
  <chr>         <dbl>     <dbl>
1 Agg           0.300 -0.000166
2 LR            0.382  0.0824
3 MLM           0.312  0.0122
```

There is no indication of major bias for aggregation or multi-level modeling. Linear regression, with a bias of about 0.12 SDs, appears much more biased than the other estimators. This is because the linear regression is targeting the person-level average treatment effect. Our data-generating process makes larger sites have larger effects, so the person-level average effect is going to be higher because those larger sites count more in the overall average. In contrast, our estimand is the school-level average treatment effect, or the simple average of each school's true impact, which we have set to 0.30. The aggregation and multi-level modeling methods target this school-level average effect. If we had instead decided that the target estimand should be the person-level

average effect, then we would find that linear regression is unbiased whereas aggregation and multi-level modeling are biased. It is crucial to think carefully about the appropriate target parameter and to assess performance with respect to a well-justified and clearly articulated target.

Which method has the smallest standard error?

The standard error measures the degree of variability in a point estimator. It reflects how stable an estimator is across replications of the data-generating process. We estimate the standard error for each estimator by taking the standard deviation of its set of estimates. For purposes of interpretation, it is useful to compare the standard errors to a benchmark estimator. Here, we treat the linear regression estimator as the benchmark and compute the magnitude of the SEs of each method *relative* to the SE of the linear regression estimator:

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

true_SE
```

```
# A tibble: 3 x 3
  method    SE per_SE
  <chr>  <dbl>  <dbl>
1 Agg    0.214  0.956
2 LR     0.224  1
3 MLM    0.213  0.953
```

For this particular DGP, aggregation and multi-level modeling have SEs about 4 to 5% smaller than linear regression, meaning these estimates tend to be a bit less variable across the different datasets. We could therefore say aggregation and multi-level modeling are preferable to linear regression, for this context, because they are more precise. In a real data analysis, these standard errors are what we would be trying to approximate with a standard error estimator.

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 estimators. We can assess overall accuracy by combining these two quantities with the RMSE:

```
ClusterRCT_runs %>%
  group_by( method ) %>%
  summarise(
    bias = mean( ATE_hat - ATE ),
```

```
    SE = sd( ATE_hat ),
    RMSE = sqrt( mean( (ATE_hat - ATE)^2 ) )
  ) %>%
  mutate(
    per_RMSE = RMSE / RMSE[method=="LR"]
  )
```

```
# A tibble: 3 x 5
  method      bias    SE  RMSE per_RMSE
  <chr>      <dbl> <dbl> <dbl>    <dbl>
1 Agg    -0.000166 0.214 0.214    0.897
2 LR      0.0824   0.224 0.238    1
3 MLM     0.0122   0.213 0.213    0.896
```

We also included SE and bias for ease of reference.

RMSE takes both bias and variance into account. For aggregation and multi-level modeling, the RMSE is the same as the standard error, which makes sense because these estimators are not biased. For linear regression, the combination of bias plus increased variability yields a higher RMSE, with the standard error dominating the bias term (note how RMSE and SE are more similar than RMSE and bias). Overall, aggregation and multi-level modeling have RMSEs around 10% smaller than linear regression, meaning they tend to produce estimates that are closer to the truth than linear regression.

9.1.2 Robust Performance Measures

We have introduced bias, variance and RMSE as three core measures of performance for point estimators. However, all of these measures are sensitive to outliers in the sampling distribution. Consider an estimator that generally does well, except for an occasional large mistake. Because conventional measures are based on arithmetic averages and squares of errors, they could indicate that such an estimator performs very poorly overall, when in fact it performs very well most of the time and terribly only rarely.

If we are more curious about typical performance, then we might prefer to use other performance measures, such as the **median bias** and the **median absolute deviation** of T. These are called "robust" measures because they are not strongly affect by a few extreme values.

One way to make a robust measure is to simply chop off some percent of the most extreme values, and then look at the central tendency or spread of the remaining values. For example, one robust measure of central tendency uses the $p \times 100\%$-trimmed mean, which ignores the estimates in the lowest and highest p-quantiles of the sampling distribution. Formally, the trimmed-mean

bias is

$$\text{Trimmed-Bias}(T;p) = \mathbb{E}\left[T \,|\, \mathbb{Q}_p(T) < T < \mathbb{Q}_{1-p}(T)\right] - \theta. \tag{9.6}$$

To estimate the trimmed bias, we take the mean of the middle $1-2p$ fraction of the distribution:

$$\widehat{\text{Trimmed-Bias}}(T;p) = \tilde{T}_{\{p\}} - \theta. \tag{9.7}$$

where (assuming pR is a whole number):

$$\tilde{T}_{\{p\}} = \frac{1}{(1-2p)R} \sum_{r=pR+1}^{(1-p)R} T_{(r)}$$

For a symmetric sampling distribution, the trimmed-mean bias will be the same as the conventional (mean) bias, but its estimator $\tilde{T}_{\{p\}}$ will be less affected by outlying values (i.e., values of T very far from the center of the distribution) compared to $\bar{T}$. However, if a sampling distribution is not symmetric, trimmed-mean bias will be a distinct performance measure, different from mean bias, that puts less emphasis on large errors compared to the conventional bias measure.

Median bias, is a special case of trimmed mean bias, with $p = 0.5$. For median bias, we "chop off" essentially all the values, focusing on the middle one alone. Positive median bias implies that more than 50% of the sampling distribution exceeds the quantity of interest, while negative median bias implies that more than 50% of the sampling distribution fall below the quantity of interest. Formally,

$$\text{Median-Bias}(T) = \mathbb{M}(T) - \theta. \tag{9.8}$$

where $\mathbb{M}(T)$ is the median of T.

We estimate median bias with the sample median, as

$$\widehat{\text{Median-Bias}}(T) = M_T - \theta, \tag{9.9}$$

where, if we define $T_{(r)}$ as the r^{th} order statistic for $r = 1, ..., R$, we have $M_T = T_{((R+1)/2)}$ if R is odd or $M_T = \frac{1}{2}\left(T_{(R/2)} + T_{(R/2+1)}\right)$ if R is even.

A further robust measure of central tendency is based on winsorizing the sampling distribution by truncating all errors larger than a certain maximum size. A winsorized distribution amounts to arguing that you do not care about errors beyond a certain magnitude, so anything beyond some set threshold will be treated the same as if it were exactly on the threshold. One useful threshold for truncation is to extend some multiplier of the inter-quartile range from the first and third quartiles of the sampling distribution (similar to determining outliers for a boxplot). For example, we might use:

$$L_w = \mathbb{Q}_{0.25}(T) - w \times (\mathbb{Q}_{0.75}(T) - \mathbb{Q}_{0.25}(T))$$
$$U_w = \mathbb{Q}_{0.75}(T) + w \times (\mathbb{Q}_{0.75}(T) - \mathbb{Q}_{0.25}(T)),$$

where $\mathbb{Q}_{0.25}(T)$ and $\mathbb{Q}_{0.75}(T)$ are the first and third quartiles of the distribution of T, respectively, and w is the number of inter-quartile ranges below which an observation will be treated as an outlier.[1] Once we have calculated the thresholds, we compute a truncated estimate as $X = \min\{\max\{T, L_w\}, U_w\}$. The winsorized bias, variance, and RMSE are then defined using winsorized values in place of the raw values of T, as

$$\begin{aligned} \text{Bias}(X) &= \mathbb{E}(X) - \theta, \\ \text{Var}(X) &= \mathbb{E}\left[(X - \mathbb{E}(X))^2\right], \\ \text{RMSE}(X) &= \sqrt{\mathbb{E}\left[(X-\theta)^2\right]}. \end{aligned} \tag{9.10}$$

To compute estimates of the winsorized performance criteria, we substitute sample quantiles $T_{(R/4)}$ and $T_{(3R/4)}$ in place of $\mathbb{Q}_{0.25}(T)$ and $\mathbb{Q}_{0.25}(T)$, respectively, to get estimated thresholds, $\hat{L}_w$ and $\hat{U}_w$, find $\hat{X}_r = \min\{\max\{T_r, \hat{L}_w\}, \hat{U}_w\}$, and compute the sample performance measures using Equations Equation 9.3, Equation 9.4, and Equation 9.5, but with $\hat{X}_r$ in place of T_r.

Alternative measures of the overall accuracy of an estimator can also be defined using quantiles. For instance, an alternative to RMSE is to use the median absolute error (MAE), defined as

$$\text{MAE} = \mathbb{M}(|T - \theta|). \tag{9.11}$$

Letting $E_r = |T_r - \theta|$, the MAE can be estimated by taking the sample median of $E_1, ..., E_R$. Many other robust measures of the spread of the sampling distribution are also available, including the Rosseeuw-Croux scale estimator Q_n (Rousseeuw and Croux, 1993) and the biweight midvariance (Wilcox, 2022). Maronna et al. (2006) provide a useful introduction to these measures and robust statistics more broadly. The `robustbase` package (Maechler et al., 2024) provides R functions for calculating many of these robust statistics.

9.2 Measures for Variance Estimators

Statistics is concerned not only with how to estimate things (e.g., point estimates), but also with understanding how well we have estimated them (e.g., standard errors). Monte Carlo simulation studies can help us evaluate both these concerns. The prior section focused on assessing how well our estimators

[1] For a normally distributed sampling distribution, the interquartile range is 1.35 SD; with $w = 2$, the lower and upper thresholds would then fall at ± 3.37 SD, or the 0.04^{th} and 99.96^{th} percentiles. Still assuming a normal sampling distribution, taking $w = 2.5$ will mean that the thresholds fall at the 0.003^{th} and 99.997^{th} percentiles.

work. In this section we focus on assessing how well our uncertainty assessments work, looking in particular on estimation of the standard error.

A first order question is whether our estimated standard errors are **calibrated**, meaning whether they are the right size, on average. For this question we will examine the bias of the squared estimated standard error. A second order question is whether our estimated standard errors are **stable**, meaning that they do not vary too much from one dataset to another. For this question we evaluate estimated standard errors in terms of effective degrees of freedom or the standard error (or RMSE) of the standard error.

9.2.1 Calibration

Calibration is a relative measure, where we look at the ratio of the expected (average) value of an uncertainty estimator to the true degree of uncertainty. The relative measure removes scale, which is important because the absolute magnitude of uncertainty will depend on many features of the data-generating process, such as sample size. In particular, across different simulation scenarios (even those with the same target parameter θ), the true standard error of an estimator T may change considerably. In other words, the ground truth changes in magnitude from one scenario to another. With a relative measure, we can make statements such as "Our standard errors are 10% too large, on average," which are easy to interpret. The relative measures lets us sidestep some of the complexity of dealing with wide variation in the magnitude of the ground truth.

Typically, we assess performance for *variance* estimators ($\widehat{SE^2}$) rather than standard error estimators. There are a few reasons it is more common to work with variance rather than standard error. First, in practice, so-called unbiased standard errors are not actually unbiased, whereas variance estimators might be.[2] For linear regression, for example, the classic standard error estimator is an unbiased *variance* estimator, but the standard error estimator is not exactly unbiased because

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

Variance is also the measure that gives us the bias-variance decomposition of Equation 9.2. Thus, if we are trying to determine whether an overall MSE is due to instability or systematic bias, operating in this squared space may be preferable.

To make the calibration measure concrete, consider a generic standard error estimator $\widehat{SE}$ to go along with our generic estimator T of target parameter θ, and let $V = \widehat{SE^2}$ be the corresponding variance estimator.

[2] See the delightfully titled section 11.5, "The Joke Is on Us: The Standard Deviation Estimator is Biased after All," in Westfall and Henning (2013) for further discussion.

In our simulation we obtain a large sample of standard errors, $\widehat{SE}_1, ..., \widehat{SE}_R$ and corresponding variance estimators $V_r = \widehat{SE}_r^2$ for $r = 1, ..., R$. Formally, the **relative bias** or **calibration** of V is then defined as

$$\text{Relative Bias}(V) = \frac{\mathbb{E}(V)}{\text{Var}(T)} \tag{9.12}$$

We say our variance estimator V is *calibrated* if the relative bias is close to one. When calibrated, V reflects how much uncertainty there actually is, on average. If relative bias is less than one, V is *anti-conservative*, meaning it tends to under-estimate the true uncertainty of T, which would usually be considered quite bad. If relative bias is more than one, V is *conservative*, meaning it tends to over-estimate the true uncertainty of T. This would be considered less than ideal, though perhaps not as severe an issue as being anti-conservative.

Calibration also has implications for the performance of other uncertainty assessments such as hypothesis tests and confidence intervals. A relative bias of less than 1 implies that V tends to under-state the amount of uncertainty, which will lead to confidence intervals that are overly narrow and do not cover the true parameter value at the desired rate. It also will lead to rejecting the null more often than desired, which elevates Type I error. Relative bias greater than 1 implies that V tends to over-state the amount of uncertainty in the point estimator, making confidence intervals too large and null hypotheses harder than necessary to reject.

To estimate relative bias of V, we simply substitute sample quantities in place of the $\mathbb{E}(V)$ and $\text{Var}(T)$. Unlike with performance measures for T, where we know the target quantity θ, here we do not know $\text{Var}(T)$ exactly. Instead, we must estimate it using S_T^2, the sample variance of T across replications. Denoting the arithmetic mean of the variance estimates across simulation trials as

$$\bar{V} = \frac{1}{R}\sum_{r=1}^{R} V_r$$

we estimate the relative bias (calibration) of V using

$$\widehat{\text{Relative Bias}}(V) = \frac{\bar{V}}{S_T^2}. \tag{9.13}$$

If we then take the square root of our estimate, we can obtain the estimated relative bias of the standard error.

9.2.2 Stability

Beyond calibration, we also care about the stability of our variance estimator. Are the estimates of uncertainty V close to the true uncertainty $Var(T)$, from

dataset to dataset?

We offer two ways of assessing this. The first is to look at the relative SE or RMSE of the variance estimator V. The second is to assess the effective degrees of freedom of V, which takes into account how one would account for the uncertainty of a standard error estimator when generating confidence intervals or conducting hypothesis tests.[3]

The relative Variance and MSE of V as an estimate of $Var(T)$ are defined as

$$\begin{aligned}\text{Relative Var}(V) &= \frac{\text{Var}(V)}{\text{Var}(T)},\\ \text{Relative MSE}(V) &= \frac{\mathbb{E}\left[\left(V-\text{Var}(T)\right)^2\right]}{\text{Var}(T)}.\end{aligned}$$

As usual, we would generally work with the the square root of these values (to get relative SE and relative RMSE). Relative standard errors describe the variability of V in comparison to the true degree of uncertainty in T—the lower the better. A relative standard error of 0.5 would mean that the variance estimator has average error of 50% of the true uncertainty, implying that V could often be off by a factor of 2—this would be very bad. Ideally, a variance estimator will have small relative bias (i.e., it is calibrated), small relative standard error (i.e., it is stable), and thus a small relative RMSE.

We can estimate the relative variance and MSE simply by plugging in estimates of the various quantities:

$$\begin{aligned}\widehat{\text{Relative}}\ \text{Var}(V) &= \frac{S_V^2}{S_T^2}\\ \widehat{\text{Relative}}\ \text{MSE}(V) &= \frac{\frac{1}{R}\sum_{r=1}^{R}\left(V_r - S_T^2\right)^2}{S_T^2},\end{aligned} \tag{9.14}$$

where the sample variance of V is

$$S_V^2 = \frac{1}{R-1}\sum_{r=1}^{R}\left(V_r-\bar{V}\right)^2.$$

Another more abstract measure of the stability of a variance estimator is its effective degrees of freedom. For some simple statistical models such as classical analysis of variance and linear regression with homoskedastic errors, the variance estimator is a sum of squares of normally distributed errors. In such cases, the sampling distribution of the variance estimator is a multiple

[3] A degrees of freedom correction used in a confidence interval or hypothesis test functions in exactly this way. For instance, when V is unstable, as represented by a low degree of freedom, we hedge against it being randomly too small by making our confidence intervals systematically larger.

of a χ^2 distribution, with degrees of freedom corresponding to the number of independent observations used to compute the sum of squares. This is what leads to t statistics and so forth.

In the context of analysis of variance problems, Satterthwaite (1946) described a method of extending this idea by approximating the variability of more complex statistics (those that can be represented as a linear combination of a sum of squares), by using a chi-squared distribution with a certain degrees of freedom. When applied to an arbitrary variance estimator V, these degrees of freedom can be interpreted as the number of independent, normally distributed errors going into a sum of squares that would lead to a variance estimator that is as equally precise as V. More succinctly, these effective degrees of freedom correspond to the amount of independent observations used to estimate V.

Following Satterthwaite (1946), we define the effective degrees of freedom of V as

$$df = \frac{2\left[\mathbb{E}(V)\right]^2}{\text{Var}(V)}. \tag{9.15}$$

We can estimate the degrees of freedom by taking

$$\widehat{df} = \frac{2\bar{V}^2}{S_V^2}. \tag{9.16}$$

For simple statistical methods in which V is based on a sum-of-squares of normally distributed errors, the effective degrees of freedom will be constant and correspond exactly to the number of independent observations in the sum of squares. Even with more complex methods, the degrees of freedom are interpretable: higher degrees of freedom imply that V is based on more observations, and thus will be a more precise estimate of the actual degree of uncertainty in T.

9.2.3 Assessing SEs for the Cluster RCT Simulation

Returning to the cluster RCT example, we will assess whether our estimated SEs are calibrated by comparing the average *estimated* (squared) standard error to the true sampling variance. Our standard errors are *inflated* if they are systematically larger than they should be, across the simulation runs. We will also look at how stable our variance estimates are by comparing their standard deviation to the true sampling variance and by computing the Satterthwaite degrees of freedom.

```
SE_performance <-
  ClusterRCT_runs %>%
  mutate( V = SE_hat^2 ) %>%
  group_by( method ) %>%
  summarise(
    SE_sq = var( ATE_hat ),
```

```
    V_bar = mean( V ),
    calib = V_bar / SE_sq,
    S_V = sd( V ),
    rel_SE_V = S_V / SE_sq,
    df = 2 * mean( V )^2 / var( V )
  )

SE_performance %>%
  knitr::kable( digits = c(0,3,3,2,3,2,1),
                caption = "Performance of the variance estimators for cluster RCT simulati
```

Table 9.1: Performance of the variance estimators for cluster RCT simulation

method	SE_sq	V_bar	calib	S_V	rel_SE_V	df
Agg	0.046	0.051	1.13	0.018	0.38	17.2
LR	0.050	0.055	1.11	0.023	0.46	11.5
MLM	0.045	0.051	1.12	0.017	0.38	17.3

The variance estimators all appear to be a bit conservative on average, with relative bias (calibration) of around 1.12, or about 12% higher than the true sampling variance. The column labelled `rel_SE_V` reports how variable the variance estimators are relative to the true sampling variances of the estimators. The column labelled `df` reports the Satterthwaite degrees of freedom of each variance estimator. Both of these measures indicate that the linear regression variance estimator is less stable than the other methods, with around 6 fewer degrees of freedom. The linear regression method uses a cluster-robust variance estimator, which is known to be unstable with few clusters (Cameron and Miller, 2015). Overall, it is a bad day for linear regression.

9.3 Measures for Confidence Intervals

Some estimation procedures provide confidence intervals (or confidence sets) which are ranges of values that should include the true parameter value with a specified confidence level. For a 95% confidence level, the interval should include the true parameter in 95% replications of the data-generating process. However, with the exception of some simple methods and models, methods for constructing confidence intervals usually involve approximations and simplifying assumptions, so their actual coverage rate might deviate from the intended confidence level.

We typically measure confidence interval performance along two dimensions:

coverage rate and **expected width**. Suppose that the confidence interval is for the target parameter θ and has intended coverage level β for $0 < \beta < 1$. Denote the lower and upper end-points of the β-level confidence interval as A and B. A and B are random quantities—they will differ each time we compute the interval on a different dataset. The coverage rate of a β-level interval estimator is the probability that it covers the true parameter, formally defined as

$$\text{Coverage}(A, B) = \Pr(A \leq \theta \leq B). \tag{9.17}$$

For a well-performing interval estimator, Coverage will be at least β and ideally will not exceed β by too much. Some analysts prefer to look at lower and upper coverage separately, where lower coverage is $\Pr(A \leq \theta)$ and upper coverage is $\Pr(\theta \leq B)$, and where it is desirable for each of these rates to be near $\beta/2$.

The expected width of a β-level interval estimator is the average difference between the upper and lower endpoints, formally defined as

$$\text{Width}(A, B) = \mathbb{E}[B - A]. \tag{9.18}$$

Smaller expected width means that the interval tends to be narrower, on average, and thus more informative about the value of the target parameter. Expected width is related to the average estimated standard error, but also includes any increase of width due to any degree-of-freedom correction.

In practice, we approximate the coverage and width of a confidence interval by summarizing across replications of the data-generating process. Let A_r and B_r denote the lower and upper end-points of the confidence interval from simulation replication r, and let $W_r = B_r - A_r$. The coverage rate and expected length measures can be estimated as

$$\begin{aligned} \widehat{\text{Coverage}}(A, B) &= \frac{1}{R}\sum_{r=1}^{R} I(A_r \leq \theta \leq B_r) \\ \widehat{\text{Width}}(A, B) &= \frac{1}{R}\sum_{r=1}^{R} W_r = \frac{1}{R}\sum_{r=1}^{R} (B_r - A_r)\,. \end{aligned} \tag{9.19}$$

Following a strict statistical interpretation, a confidence interval performs acceptably if it has actual coverage rate greater than or equal to β. If multiple methods satisfy this criterion, then the method with the lowest expected width would be preferable.

In many instances, confidence intervals are constructed using point estimators and uncertainty estimators. For example, a conventional Wald-type confidence interval is centered on a point estimator, with end-points taken to be a multiple of an estimated standard error below and above the point estimator:

$$A = T - c \times \widehat{SE}, \quad B = T + c \times \widehat{SE}$$

for some critical value c (e.g., for a normal critical value with a $\beta = 0.95$ confidence level, $c = 1.96$). Because of these connections, confidence interval coverage will often be closely related to the performance of the point estimator and variance estimator. Biased point estimators will tend to have confidence intervals with coverage below the desired level because they are not centered in the right place. Likewise, variance estimators that have relative bias below 1 will tend to produce confidence intervals that are too short, leading to coverage below the desired level. Thus, confidence interval coverage captures multiple aspects of the performance of an estimation procedure.

9.3.1 Confidence Intervals in the Cluster RCT Simulation

Returning to the CRT simulation, we will examine the coverage and expected width of normal Wald-type confidence intervals for each of the estimators under consideration. To do this, we first have to calculate the confidence intervals because we did not do so in the estimation function. We compute a normal critical value for a $\beta = 0.95$ confidence level using `qnorm(0.975)`, then compute the lower and upper end-points using the point estimators and estimated standard errors:

```
runs_CIs <-
  ClusterRCT_runs %>%
  mutate(
    A = ATE_hat - qnorm(0.975) * SE_hat,
    B = ATE_hat + qnorm(0.975) * SE_hat
  )
```

Now we can estimate the coverage rate and expected width of these confidence intervals:

```
runs_CIs %>%
  group_by( method ) %>%
  summarise(
    coverage = mean( A <= ATE & ATE <= B ),
    width = mean( B - A )
  )
```

```
# A tibble: 3 x 3
  method coverage width
  <chr>     <dbl> <dbl>
1 Agg       0.948 0.877
2 LR        0.92  0.903
3 MLM       0.945 0.872
```

The coverage rate is close to the desired level of 0.95 for the multilevel model and aggregation estimators, but it is a bit too low for linear regression. The lower-than-nominal coverage level occurs because of the bias of the linear

regression point estimator. The linear regression confidence intervals are a bit wider than the other methods due to the larger standard errors.

The normal Wald-type confidence intervals we have examined here are based on fairly rough approximations. In practice, we might want to examine more carefully constructed intervals such as ones that use critical values based on t distributions or ones constructed by profile likelihood. Especially in scenarios with a small or moderate number of clusters, such methods might provide better intervals, with coverage closer to the desired confidence level. In our earlier assessment of the variance estimator (see Section 9.2.3), we also found Linear Regression had lower degrees of freedom, meaning it might have even wider intervals if they were calculated more carefully. See Exercise 9.4.

9.4 Measures for Inferential Procedures (Hypothesis Tests)

Hypothesis testing entails first specifying a null hypothesis, such as that there is no difference in average outcomes between two experimental groups. One then collects data and evaluates whether the observed data is compatible with the null hypothesis. Hypothesis test results are often described in terms of a p-value, which measures how extreme or surprising a feature of the observed data (a test statistic) is relative to what one would expect if the null hypothesis is true. A small p-value (such as $p < .05$ or $p < .01$) indicates that the observed data would be unlikely to occur if the null is true, leading the researcher to reject the null hypothesis. Alternately, testing procedures might be formulated by comparing a test statistic to a specified critical value; a test statistic exceeding the critical value would lead the researcher to reject the null.

Hypothesis testing procedures aim to control the level of false positives, corresponding to the probability that the null hypothesis is rejected when it holds in truth. The level of a testing procedure is often denoted as α, and it has become conventional in many fields to conduct tests with a level of $\alpha = .05$.[4] Just as in the case of confidence intervals, hypothesis testing procedures can sometimes be developed that will have false positive rates exactly equal to the intended level α. However, in many other problems, hypothesis testing procedures involve approximations or simplifying assumptions, so that the actual rate of false positives might deviate from the intended α-level. Knowing the risk of this is important. When we evaluate a hypothesis testing procedure, we

[4]The convention of using $\alpha = .05$ does not have a strong theoretical rationale. Many scholars have criticized the rote application of this convention and argued for using other α levels. See Benjamin et al. (2017) and Lakens et al. (2018) for spirited arguments about choosing α levels for hypothesis testing.

are concerned with two primary measures of performance: *validity* and *power*.

9.4.1 Validity

Validity pertains to whether we erroneously reject a true null more often than we should. An α-level testing procedure is valid if it has no more than an α chance of rejecting the null, when the null is true. If we were using the conventional $\alpha = .05$ level, then a valid testing procedure will reject the null in only about 50 of 1000 replications of a data-generating process where the null hypothesis actually holds.

To assess validity, we will need to specify a data generating process where the null hypothesis holds (e.g., where there is no difference in average outcomes between experimental groups). We then generate a large series of data sets with a true null, conduct the testing procedure on each dataset and record the p-value or critical value, then score whether we reject the null hypothesis. In practice, we may be interested in evaluating a testing procedure by exploring data generation processes where the null is true but other aspects of the data (such as outliers, skewed outcome distributions, or small sample size) make estimation difficult, or where auxiliary assumptions of the testing procedure are violated. Examining such data-generating processes allows us to understand if our methods are robust to patterns that might be encountered in a real data analysis. The key to evaluating the validity of a procedure is that, for whatever data-generating process we examine, the null hypothesis must be true.

9.4.2 Power

Power is concerned with the chance that we notice when an effect exists—that is, the probability of rejecting the null hypothesis when it does not actually hold. Compared to validity, power is a more nuanced concept because larger effects will clearly be easier to notice than smaller ones, and more blatant violations of a null hypothesis will be easier to identify than subtle ones. Furthermore, the rate at which we can detect violations of a null will depend on the α level of the testing procedure. A lower α level will make for a less sensitive test, requiring stronger evidence to rule out a null hypothesis. Conversely, a higher α level will reject more readily, leading to higher power but at a cost of increased false positives.

In order to evaluate the power of a testing procedure by simulation, we will need to generate data where there is something to detect. In other words, we will need to ensure that the null hypothesis is violated and that some specific alternative hypothesis of interest holds. The process of evaluating the power of a testing procedure is otherwise identical to that for evaluating its validity: generate many datasets, carry out the testing procedure, and track the rate at which the null hypothesis is rejected. The only difference is the *conditions*

under which the data are generated.

We often find it useful to think of power as a *function* rather than as a single quantity because its absolute magnitude will generally depend on the sample size of a dataset and the magnitude of the effect of interest. Because of this, power evaluations will typically involve examining a *sequence* of data-generating scenarios with varying sample size or varying effect size. If we are actually planning a future analysis, we would then look to determine what sample size or effect size would give us 80% power (a traditional target power level used for planning in many cases). See Chapter 19 for more on power analysis for study planning.

Separately, if our goal is to compare testing procedures, then absolute power at substantively large effects is often less informative than the relative power of different testing procedures at substantively small effects, near to but not exactly at the null. In this setting, attention naturally shifts to infinitesimal power—that is, how quickly power increases as we move from a null effect to very small alternatives. Examining infinitesimal power helps identify which tests are most sensitive to small departures from the null in a more standardized way.

9.4.3 Rejection Rates

When evaluating either validity or power, the main performance measure is the **rejection rate** of the hypothesis test. Let P be the p-value from a procedure for testing the null hypothesis that a parameter $\theta = 0$. The rejection rate is then

$$\rho_\alpha(\theta) = \Pr(P < \alpha) \tag{9.20}$$

When data are simulated from a process in which the null hypothesis is true (i.e., θ really does equal 0), then the rejection rate is equivalent to the Type-I error rate of the test. When the data are simulated from a process in which the null hypothesis is violated ($\theta \neq 0$), then the rejection rate is equivalent to the power of the test.

To estimate the rejection rate, calculate the proportion of replications where the test rejects the null hypothesis. Letting $P_1, ..., P_R$ be the p-values simulated from R replications of a data-generating process with true parameter θ, estimate the rejection rate by calculating

$$r_\alpha(\theta) = \frac{1}{R}\sum_{r=1}^{R} I(P_r < \alpha). \tag{9.21}$$

It may be of interest to evaluate the performance of the test at several different α levels. For instance, Brown and Forsythe (1974) evaluated the Type-I error rates and power of their tests using $\alpha = .01$, .05, and .10. Recording the p-value of the test makes it easy to estimate rejection rates for multiple α levels,

since we simply apply Equation 9.21 for each value of α. When simulating from a data-generating process where the null hypothesis holds, one can also plot the empirical cumulative distribution function of the p-values; for an exactly valid test, the p-values should follow a standard uniform distribution with a cumulative distribution falling along the 45° line.

Methodologists hold a variety of 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, for any specific conditions of a data-generating process. Among a collection of level-α testing procedures, we would prefer the one with highest power. If looking only at null rejection rates, then the test with Type-I error closest to α would usually be preferred.

9.4.4 Inference in the Cluster RCT Simulation

Returning to the cluster RCT simulation, we evaluate the validity and power of hypothesis tests for the average treatment effect based on each of the three estimation methods. The data used in previous sections of the chapter was simulated under a process with a non-null treatment effect parameter (equal to 0.3 SDs), so the null hypothesis of zero average treatment effect does not hold. Thus, the rejection rates for this scenario correspond to estimates of power. We compute the rejection rate for tests with an α level of .05:

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

```
# A tibble: 3 x 2
  method power
  <chr>  <dbl>
1 Agg    0.241
2 LR     0.32
3 MLM    0.263
```

For this particular scenario, none of the tests have especially high power, and the linear regression estimator apparently has higher power than the aggregation method and the multi-level model.

To make sense of this power pattern, we should consider the validity of the testing procedures. We can do so by running a new simulation using the function we wrote in Section 8.3, but with an average treatment effect of zero (i.e., we make the null hypothesis true). We also increase `size_coef` and `alpha` parameter to induce additional treatment effect heterogeneity, and increase the number of clusters to get more stable estimation:

```
set.seed( 4404044 )
runs_val <- run_CRT_sim( reps = 1000,
                         n_bar = 30, J = 30,
                         ATE = 0, ICC = 0.2,
                         size_coef = 0.75, alpha = 0.75 )
```

Assessing validity involves repeating the exact same rejection rate calculations as we did for power. We also estimate the bias of the point estimator, since bias can affect the validity of a testing procedure:

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

```
# A tibble: 3 x 3
  method     bias power
  <chr>     <dbl> <dbl>
1 Agg    -0.00147 0.06
2 LR      0.129   0.104
3 MLM     0.0170  0.065
```

The aggregation and multi-level modeling approaches both seem to reject a bit below 0.05, at 0.04, while LR is here at 0.10. The test for the linear regression estimator has an elevated Type-I error, probably driven by the upward bias of the point estimator used in constructing the test (see the bias column). The elevated rejection rate due to bias might then be part of the reason that the linear regression had higher power than the other procedures in our power example, above. If so, it is not entirely fair to compare the power of the three testing procedures, because one of them has Type-I error in excess of the desired level.

As discussed above, linear regression targets the person-level average treatment effect. In the scenario we just simulated for evaluating validity, the person-level average effect is not zero even though our cluster-average effect is zero, because we have specified a non-zero impact heterogeneity parameter (`size_coef=0.75`) along with positive school size variation (`alpha = 0.75`), so that the school-specific treatment effects vary around 0. To see if this is why the linear regression test has an inflated Type-I error rate, we could re-run the simulation using settings where both the school-level and person-level average effects are truly zero by setting `size_coef = 0`.

9.5 Relative or Absolute Measures?

A **relative measure** is when you compare a performance measure to the target quantity by taking a ratio. For example, **relative bias** is the expected value of an estimator divided by the true parameter value. As we saw when evaluating variance, above, relative measures can be very useful, giving statements such as "this estimator's standard errors are 10% too big, on average." In the best case, they are easy to interpret because they are unitless. However, they also suffer from some notable drawbacks.

Relative measures can be difficult to interpret under several circumstances. For one, ratios can be deceiving when the denominator, i.e. reference value, is near zero. This can be a problem when, for example, examining bias relative to a benchmark estimator if the benchmark has no or little bias. Relative measures can also be problematic when the denominator can take on both negative or positive values. Finally, relative measures can again be misleading if the reference value changes unexpectedly with changes in the simulation context.

We now discuss these issues in more detail and offer guidance on how to decide whether to to use relative or absolute performance measures.

9.5.1 Performance relative to the target parameter

In considering performance measures for point estimators, we have defined the measures in terms of differences (bias, median bias) and average deviations (variance and RMSE), all of which are on the scale of the target parameter. In contrast, for evaluating estimated standard errors we have defined measures in relative terms, calculated as *ratios* of the average estimate to the target quantity, rather than as differences. In the latter case, relative measures are justified because the target quantity (the true degree of uncertainty) is always positive and is usually strongly affected by design parameters of the data-generating process. Is it ever reasonable to use relative measures for point estimators? If so, how should we decide whether to use relative or absolute measures?

Many published simulation studies do use relative performance measures for evaluating point estimators. For instance, studies might use relative bias or relative RMSE, defined as

$$\begin{aligned} \text{Relative Bias}(T) &= \frac{\mathbb{E}(T)}{\theta}, \\ \text{Relative RMSE}(T) &= \frac{\sqrt{\mathbb{E}\left[(T-\theta)^2\right]}}{\theta}. \end{aligned} \tag{9.22}$$

and estimated as

$$\begin{aligned} \widehat{\text{Relative Bias}}(T) &= \frac{\bar{T}}{\theta}, \\ \widehat{\text{Relative RMSE}}(T) &= \frac{\widehat{RMSE}(T)}{\theta}. \end{aligned} \tag{9.23}$$

As justification for evaluating bias in relative terms, authors often appeal to Hoogland and Boomsa (1998), who suggested that relative bias of under 5% (i.e., relative bias falling between 0.95 and 1.05) could be considered acceptable for an estimation procedure. However, Hoogland and Boomsa (1998) were writing about a very specific context—robustness studies of structural equation modeling techniques—that have parameters of a particular form. In our view, their proposed rule-of-thumb is often generalized far beyond the circumstances where it is defensible, including to problems where it is clearly arbitrary and inappropriate.

One big problem with relative bias, in particular, arises when the target parameter is equal to or near zero. Suppose you have a bias of 0.1 that is consistent across values of the true parameter. For a true parameter value of 0.2, the relative bias would be 0.5. But if the true value were 0.01, then the relative bias would be 10! As the true value gets smaller and smaller, the relative bias will explode.

Another problem with relative bias occurs when the target parameter is on an arbitrary scale, so that the absolute magnitude is not meaningful. For example, in our Cluster RCT simulation, the relative bias of the ATE estimators would change if we moved the ATE from 0.2 to 0.5, or from 0.2 to 0 (where relative bias would now be entirely undefined). In contrast, the absolute bias would stay the same regardless of the ATE.

A more principled approach to choosing between whether to use absolute and relative measures is to first consider whether the magnitude of the measure changes across different performances of the target parameter θ. If the estimand of interest is a location parameter, then the bias, variance or RMSE of an estimator will likely not change as θ changes (see Figure 9.2, left). Focusing on relative measures in this scenario would lead to a much more complicated story because different values of θ will produce drastically different values, ranging from nearly unbiased to nearly infinite bias (for θ very close to zero). Better to use absolute bias, which is nearly constant and simpler to interpret and explain. On the other hand, if changing θ would lead to proportionate changes in the magnitude of bias, variance, or RMSE, then relative bias may be useful because it could lead to a simple story such as "the estimator is usually estimating around 12% too high," or similar. This is what is shown at right of Figure 9.2. We usually expect this type of pattern to occur for scale parameters, such as standard deviations or standard errors (as we considered in Section 9.2).

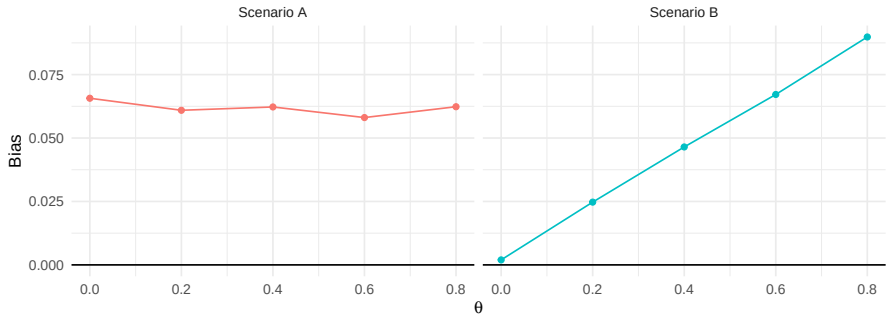


Figure 9.2: Hypothetical relationships between bias and a target parameter θ. In Scenario A, bias is unrelated to θ and absolute bias is a more appropriate measure. In Scenario B, bias is proportional to θ and relative bias is a more appropriate measure.

How do we know which of these scenarios is a better match for a particular problem? For some estimators and data-generating processes, it may be possible to use statistical theory to examine the how bias or variance would be expected to change as a function of θ. However, many problems are too complex to be tractable. A generally more feasible route is to evaluate performance for *multiple* values of the target parameter and then generate a graph such as shown in Figure 9.2 to understand how performance changes as a function of the target parameter. If the absolute bias is roughly the same for all values of θ (as in Scenario A), then use absolute bias. On the other hand, if the bias grows roughly in proportion to θ (as in Scenario B), then relative bias might be a better summary criterion. Generally, unless you believe bias must be zero if the parameter θ is zero, absolute bias is usually the safer choice.

9.5.2 Performance relative to a benchmark estimator

Another way to define performance measures in relative terms is to take the ratio of the performance measure for one estimator over the performance measure for a benchmark estimator (or some other benchmark quantity). We have already demonstrated this approach in comparing standard errors and RMSEs for the cluster RCT example (Section 9.1.1), where we used the linear regression estimator as the benchmark. This approach is natural in simulations that involve comparing the performance of multiple estimators and where one of the estimators could be considered the current standard or conventional method.

Comparing the performance of one estimator relative to another can be especially useful when examining measures whose magnitude varies drastically across design parameters. For example, we typically expect precision and ac-

curacy to improve (variance and RMSE to decrease) as sample size increases. Comparing estimators in terms of *relative* precision or *relative* accuracy may then make it easier to identify consistent patterns in the simulation results. A relative measure might allow us to summarize findings by saying that "the aggregation estimator has standard errors that are consistently 6-10% smaller than the standard errors of the linear regression estimator." This is much easier to interpret than saying that "aggregation has standard errors that are around 0.01 smaller than linear regression, on average." In the latter case, it is very difficult to determine whether a difference of 0.01 is large or small. Furthermore, focusing on an average difference conceals relevant variation across scenarios involving different sample sizes.

Comparing performance relative to a benchmark method can be an effective tool, but it also has potential drawbacks. Because these relative performance measures are inherently comparative, higher or lower ratios could either be due to the behavior of the method of interest (the numerator) or due to the behavior of the benchmark method (the denominator). Consider if the benchmark completely falls apart under some circumstance, but the comparison only worsens a slight bit. The relative measure, focused on the comparison, would appear to be a marked improvement, which could be deceptive.

Ratio comparisons are also less effective for performance measures, such as power, that are on a constrained scale. If a baseline method has power of 0.05, and we improve it to 0.10, we have doubled our power, but if it is 0.50 and we increase to 0.55, we have only increased by 10%. Whether this is misleading or not will depend on context.

9.6 Estimands Not Represented By a Parameter

In the Cluster RCT example, we focused on the estimand of the school-level ATE, represented by the model parameter γ_1. What if we were instead interested in the person-level average effect? This estimand does not correspond to any input parameter in our data generating process. Instead, it is defined *implicitly* by a combination of other parameters. In order to compute performance characteristics such as bias and RMSE, we would need to first calculate the target estimand based on the inputs to the data-generating processes. There are at least three possible ways to accomplish this: mathematical theory, generating a massive dataset, or tracking the estimand during simulation.

Mathematical theory. One approach is to use mathematical distribution theory to compute the implied parameter. Our target parameter will be some function of the parameters and random variables in the data-generating process, and it may be possible to evaluate that function algebraically or numer-

ically (i.e., using numerical integration functions such as `integrate()`). This can be a very worthwhile exercise if it provides insights into the relationship between the target parameter and the inputs of the data-generating process. However, this approach requires knowledge of distribution theory, and it can get quite complicated and technical.[5]

Massive dataset. Another approach is to generate a massive dataset—one so large that it can stand in for the entire data-generating model—and then simply calculate the target parameter of interest in this massive dataset. In the cluster-RCT example, we can apply this strategy by generating data from a very large number of clusters and then simply calculating the true person-average effect across all generated clusters. If the dataset is big enough, then the uncertainty in this estimate will be negligible compared to the uncertainty in our simulation.

We implement this approach as follows, generating a dataset with 100,000 clusters:

```
dat <- gen_cluster_RCT(
  n_bar = 30, J = 100000,
  gamma_1 = 0.3, gamma_2 = 0.5,
  sigma2_u = 0.20, sigma2_e = 0.80,
  alpha = 0.75
)

ATE_person <- mean( dat$Yobs[dat$Z==1] ) - mean( dat$Yobs[dat$Z==0] )
```

Our extremely precise estimate of the person-average effect is 0.4, which is consistent with what we would expect given the bias we saw earlier for the linear model.

If we recalculate performance measures for all of our estimators with respect to the `ATE_person` estimand, the bias and RMSE of our estimators will shift, although the standard errors will stay the same.

```
performance_person_ATE <-
  ClusterRCT_runs %>%
  group_by( method ) %>%
  summarise(
    bias = mean( ATE_hat ) - ATE_person,
    SE = sd( ATE_hat ),
    RMSE = sqrt( mean( (ATE_hat - ATE_person)^2 ) )
  ) %>%
  mutate( per_RMSE = RMSE / RMSE[method=="LR"] )

performance_person_ATE
```

[5] In the cluster-RCT example, the distribution theory is tractable. See Exercise 9.7.

```
# A tibble: 3 x 5
  method    bias    SE  RMSE per_RMSE
  <chr>    <dbl> <dbl> <dbl>    <dbl>
1 Agg    -0.0955 0.214 0.234     1.05
2 LR     -0.0130 0.224 0.224     1
3 MLM    -0.0832 0.213 0.229     1.02
```

For the person-weighted estimand, the aggregation estimator and multilevel model are biased but the linear regression estimator is mostly unbiased. However, the aggregation estimator and multilevel model estimator still have smaller standard errors than the linear regression estimator. RMSE now captures the trade-off between increased bias and reduced variance. The bias price is too high in this case: aggregation and multilevel modeling have an RMSE 5 and 2 percent larger than linear regression.

Track during the simulation. A third approach for calculating an implied estimand is to record a truly unbiased estimate of it with each simulation trial, and then average across all trials to get the true estimand.

For example, in the cluster RCT simulation, we could record the true person average effect of each dataset for each simulation iteration, and then average the sample-specific parameters at the end. The overall average of the dataset-specific `ATE_person` parameters corresponds to the population person-level ATE. This approach is equivalent to generating a single massive dataset—we just generate it piece by piece.

To implement this approach, we would need to modify the data-generating function `gen_cluster_RCT()` to track the additional information. For instance, we might calculate

```
tx_effect <- gamma_1 + gamma_2 * ( nj - n_bar ) / n_bar
beta_0j <- gamma_0 + Zj * tx_effect + u0j
```

and then include `tx_effect` along with `Yobs` and `Z` as a column in our dataset. This approach is quite similar to directly calculating *potential outcomes*, as discussed in Chapter 20.

After modifying the data-generating function, we will also need to modify the analysis function(s) to record the sample-specific treatment effect parameter. We might have, for example:

```
analyze_data = function( dat ) {
  MLM <- analysis_MLM( dat )
  LR <- analysis_OLS( dat )
  Agg <- analysis_agg( dat )
  res <- bind_rows(
    MLM = MLM, LR = LR, Agg = Agg,
    .id = "method"
  )
```

```
  res$ATE_person <- mean( dat$tx_effect )
  return( res )
}
```

When we run our simulation with this code, we will end up with a column of true person-average treatment effects. The average of these value across replications would give an estimated true person average treatment effect for the population. We would then use this as the target parameter for performance calculations, as we did above.

An estimand not represented by any single input parameter is more difficult to work with than one that corresponds directly to an input parameter. That said, it might be a more important estimand than one represented directly with an input parameter. As the above shows, it is quite feasible to examine such estimands with a bit of forethought and careful programming. The key is to be clear about what you are trying to estimate because the performance of an estimator depends on the estimand against which it is compared.

9.7 Uncertainty in Performance Estimates

The performance measures we have described are all defined in terms of an infinite number of replications of the data-generating process. Of course, simulations only involve a finite set of replications, based on which we *estimate* the measures. If we re-ran the simulation (with a different seed), our estimates would change. We use the term *Monte Carlo error* to describe the variability of performance measure estimates that is due to using a random, finite set of replications. To understand how good our estimates of performance are, we need to assess how large the Monte Carlo error is likely to be.

To account for Monte Carlo error, we can think of simulation results as a sample from a population. Each replication is an independent draw from the population of the sampling distribution. Once we frame the problem in these terms, we can apply standard statistical techniques to calculate standard errors for our performance measures, which we will call *Monte Carlo Standard Errors*, or MCSEs. For most performance measures, closed-form expressions for MCSEs are available. If no closed-form expressions are available, we can still estimate MCSEs using techniques such as the jackknife or bootstrap.

9.7.1 Conventional measures for point estimators

For the performances measures that we have described for evaluating point estimators, Monte Carlo standard errors can be calculated using conventional

formulas.[6] When we run a simulation, we end up with R replicates of a point estimator T of a target parameter θ. A performance measure of T (e.g., bias, RMSE, or SE) is a summary across these R replicates.

For example, the bias of T is $\mathbb{E}(T-\theta)$, which is the expected value of T minus the true parameter value, and we estimate the bias as $\bar{T}-\theta$. Each replicate of $T-\theta$ is an *estimate* of the expected value of $T-\theta$, and, in this case, we have R independent draws of $T-\theta$ and can use simple sampling theory to get the standard error for the population mean of them:

$$MCSE\left(\widehat{\text{Bias}}(T)\right) = \sqrt{\frac{S_T^2}{R}}, \tag{9.24}$$

where S_T is the sample standard deviation of T across the R replications of the simulation process (noting that $Var(T-\theta) = Var(T)$). Equation 9.24 is simply the standard error for the mean of R independent samples from a population.

This approach is readily extended to estimators of other performance measures. If we have a performance measure M for each replicate in a simulation, we can estimate average performance as the average of the sample of R performances, $\bar{M}$. The standard error for average performance then follows along the same lines as Equation 9.24. For example, the Mean Square Error (the square of RMSE) is simply the average of $M = (T-\theta)^2$ and the MCSE of the MSE is then

$$MCSE(\widehat{MSE}) = \sqrt{\frac{\text{Var}(M)}{R}} = \sqrt{\frac{\text{Var}\left((T-\theta)^2\right)}{R}}. \tag{9.25}$$

For quantifying the accuracy of an estimator, we have recommended using RMSE, or the square root of the MSE, as shown in Equation 9.5. To get the MCSE for RMSE, we can use a delta method approximation.[7] In the case of

[6] To be precise, the formulas that we give are *estimators* for the Monte Carlo standard errors of the performance measure estimators. Our presentation does not emphasize this point because the performance measures will usually be estimated using a large number of replications from an independent and identically distributed process, so the distinction between true and estimated Monte Carlo standard errors will not be consequential. One could write $MCSE$ and $\widehat{MCSE}$ to distinguish between the two, but we avoid this notation for simplicity.

[7] Oehlert (1992) provides a friendly introduction to the delta method, or see Pustejovsky (2018). The delta method approximation says (with some conditions), that if we assume $X \sim N\left(\phi, \sigma_X^2\right)$, then we can approximate the distribution of $g(X)$ for some continuous function $g(\cdot)$ as

$$g(X) \sim N\left(g(\phi),\ \ g'(\phi)^2 \times \sigma_X^2\right),$$

where $g'(\phi)$ is the derivative of $g(\cdot)$ evaluated at ϕ. In practice, ϕ is an unknown parameter, but we can estimate the variance of $g(X)$ by substituting X in place of ϕ. Thus, an approximate estimator of the variance of $g(X)$ is $g'(X)^2 \times \hat{\sigma}_X^2$, and an approximate SE for $g(X)$ is

$$SE(g(X)) \approx |g'(\mu_X)| \times SE(X).$$

To find the MCSE of $RMSE$, we apply the delta method approximation to $X = MSE$ with $g(x) = \sqrt{x}$ and $g'(x) = \frac{1}{2\sqrt{x}}$.

the RMSE,

$$MCSE(\widehat{RMSE}) = \frac{1}{2 \times RMSE}\sqrt{\frac{Var((T-\theta)^2)}{R}}. \tag{9.26}$$

To actually approximate the MCSE, we plug in the estimated $RMSE$ and the variance of the R values of $(T-\theta)^2$. The delta method approach can be applied to many other quantities of interest as well.

The MCSE for the true standard error S_T of T does not directly follow the above pattern because the sampling variance S_T^2 of T is estimated across all the replicates, rather than simply being an average of R independent estimates. Statistical theory gives the standard error of a variance estimator as

$$MCSE\left(\widehat{\text{Var}}(T)\right) = MCSE\left(S_T^2\right) = S_T^2\sqrt{\frac{k_T - 1}{R}}, \tag{9.27}$$

where k_T is the standardized kurtosis of T:

$$k_T = \frac{1}{S_T^4}\frac{1}{R}\sum_{r=1}^{R}\left(T_r - \bar{T}\right)^4. \tag{9.28}$$

With the MCSE of the variance, we can use the same delta method approximation as above to get the MCSE of the standard error:

$$MCSE\left(S_T\right) \approx \frac{1}{2S_T}MCSE(S_T^2) = \frac{S_T}{2}\sqrt{\frac{k_T - 1}{R}}. \tag{9.29}$$

Section 9.5 discussed circumstances where we might prefer to calculate performance measures in relative rather than absolute terms. For measures that are calculated by dividing a raw measure by the known target parameter, the MCSE for the relative measure is simply the MCSE for the raw measure divided by the target parameter. For instance, the MCSE of relative bias $\bar{T}/\theta$ is

$$MCSE\left(\frac{\bar{T}}{\theta}\right) = \frac{1}{\theta}MCSE(\bar{T}) = \frac{S_T}{\theta\sqrt{R}}. \tag{9.30}$$

MCSEs for relative variance and relative RMSE follow similarly.

9.7.2 MCSEs for robust measures for point estimators

In Section 9.1.2 we described several alternative performance measures for evaluating point estimators. MCSEs for these less conventional measures can be obtained using results from the theory of robust statistics (Hettmansperger and McKean, 2010; Maronna et al., 2006).

McKean and Schrader (1984) proposed a standard error estimator for the sample median from a continuous but not necessarily normal distribution,

derived from a non-parametric confidence interval for the sample median. We use their approach to compute a MCSE for M_T, the sample median of T. Let $c = \left\lceil (R+1)/2 - 1.96 \times \sqrt{R/4} \right\rfloor$, where the inner expression is rounded to the nearest integer. Then

$$MCSE\left(M_T\right) = \frac{T_{(R+1-c)} - T_{(c)}}{2 \times 1.96}. \tag{9.31}$$

A Monte Carlo standard error for the median absolute deviation can be computed following the same approach, but substituting the order statistics of $E_r = |T_r - \theta|$ in place of those for T_r.

For the trimmed mean bias with trimming proportion p, a MCSE is

$$MCSE\left(\tilde{T}_{\{p\}}\right) = \sqrt{\frac{U_p}{R}}, \tag{9.32}$$

where

$$U_p = \frac{1}{(1-2p)R}\left(pR\left(T_{(pR)} - \tilde{T}_{\{p\}}\right)^2 + pR\left(T_{((1-p)R+1)} - \tilde{T}_{\{p\}}\right)^2 + \sum_{r=pR+1}^{(1-p)R}\left(T_{(r)} - \tilde{T}_{\{p\}}\right)^2\right).$$

(Maronna et al., 2006, Eq. 2.85). The additional uncertainty due to the thresholds being estimated is ignorable.

Performance measures based on winsorization include winsorized bias, winsorized standard error, and winsorized RMSE. MCSEs for these measures can be computed using the same formuals as for the conventional measures of bias, standard error, and RMSE, but using sample moments of $\hat{X}_r$ in place of the sample moments of T_r. The additional uncertainty from the thresholds being estimated is ignorable.

9.7.3 MCSE for Confidence Intervals and Hypothesis Tests

Performance measures for confidence intervals and hypothesis tests are simple compared to those we have described for point and variance estimators, and come directly from the theory of estimating the proportion in a population. First, the MCSE for the estimated rejection rate is

$$MCSE(r_\alpha) = \sqrt{\frac{r_\alpha(1-r_\alpha)}{R}}. \tag{9.33}$$

This MCSE uses the estimated rejection rate to approximate its Monte Carlo error. When evaluating the validity of a test, we might expect the rejection rate to be fairly close to the nominal α level, in which case we could compute a MCSE using α in place of r_α, taking $\sqrt{\alpha(1-\alpha)/R}$. When evaluating power, we will not usually know the neighborhood of the rejection rate in advance

of the simulation. However, a conservative upper bound on the MCSE can be derived by observing that MCSE is maximized when $r_\alpha = \frac{1}{2}$, giving

$$MCSE(r_\alpha) \leq \frac{1}{2\sqrt{R}}.$$

When evaluating confidence interval performance, we focus on coverage rates and expected widths. MCSEs for the estimated coverage rate work similarly to those for rejection rates. If the coverage rate is expected to be in the neighborhood of the intended coverage level β, then we can approximate the MCSE as

$$MCSE(\widehat{\text{Coverage}}(A, B)) = \sqrt{\frac{\beta(1-\beta)}{R}}. \tag{9.34}$$

Alternately, Equation 9.34 could be computed using the estimated coverage rate $\widehat{\text{Coverage}}(A, B)$ in place of β.

Finally, the expected confidence interval width is estimated as $\bar{W}$, with MCSE

$$MCSE(\bar{W}) = \sqrt{\frac{S_W^2}{R}}, \tag{9.35}$$

where S_W^2 is the sample variance of $W_1, ..., W_R$, the widths of the confidence interval from each replication.

9.7.4 Using the Jackknife to obtain MCSEs (for relative variance estimators)

Estimating the MCSE of relative performance measures for variance estimators is complicated by the appearance of an estimated quantity in the denominator of the ratio. For instance, the relative bias of V is estimated as the ratio $\bar{V}/S_T^2$, and both the numerator and denominator are estimated quantities that will include some Monte Carlo error. To properly account for the Monte Carlo uncertainty of the ratio, one possibility is to use formulas for the standard errors of ratio estimators. Alternately, we can use general uncertainty approximation techniques such as the jackknife (Boos and Osborne, 2015) or bootstrap (Efron and Tibshirani, 1994). The jackknife involves calculating a statistic of interest repeatedly, each time excluding one observation. The variance of this set of one-left-out statistics usually serves as a reasonable approximation to the actual sampling variance of the statistic calculated from the full sample.

To apply the jackknife to assess MCSEs of relative bias or relative RMSE of a variance estimator, we will need to compute several statistics repeatedly. Let $\bar{V}_{(j)}$ and $S^2_{T(j)}$ be the average variance estimate and the 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 of these R versions yields a jackknife MCSE:

$$MCSE\left(\frac{\bar{V}}{S_T^2}\right) = \sqrt{\frac{1}{R}\sum_{j=1}^{R}\left(\frac{\bar{V}_{(j)}}{S_{T(j)}^2} - \frac{\bar{V}}{S_T^2}\right)^2}. \tag{9.36}$$

Similarly, a MCSE for the relative standard error of V is

$$MCSE\left(\frac{S_V}{S_T^2}\right) = \sqrt{\frac{1}{R}\sum_{j=1}^{R}\left(\frac{S_{V(j)}}{S_{T(j)}^2} - \frac{S_V}{S_T^2}\right)^2}, \tag{9.37}$$

where $S_{V(j)}$ is the sample variance of $V_1, ..., V_R$, omitting replicate j.

For the relative RMSE of V, let

$$RRMSE_V = \frac{1}{S_T^2}\sqrt{(\bar{V} - S_T^2)^2 + \frac{R-1}{R}S_V^2}$$

and

$$RRMSE_{V(j)} = \frac{1}{S_{T(j)}^2}\sqrt{(\bar{V}_{(j)} - S_{T(j)}^2)^2 + \frac{R-1}{R}S_{V(j)}^2}.$$

Then the jackknife MCSE for the estimated relative RMSE of V is

$$MCSE\left(RRMSE_V\right) = \sqrt{\frac{1}{R}\sum_{j=1}^{R}\left(RRMSE_{V(j)} - RRMSE_V\right)^2}. \tag{9.38}$$

Jackknife calculation would be cumbersome if we did it by brute force. However, a few algebra tricks provide a much quicker way. The tricks come from observing that

$$\begin{aligned}
\bar{V}_{(j)} &= \frac{1}{R-1}\left(R\bar{V} - V_j\right) \\
S_{V(j)}^2 &= \frac{1}{R-2}\left[(R-1)S_V^2 - \frac{R}{R-1}\left(V_j - \bar{V}\right)^2\right] \\
S_{T(j)}^2 &= \frac{1}{R-2}\left[(R-1)S_T^2 - \frac{R}{R-1}\left(T_j - \bar{T}\right)^2\right]
\end{aligned}$$

These formulas can be used to avoid re-computing the mean and sample variance from every subsample. Instead, all we need to do is calculate the overall mean and overall variance, and then do a small adjustment with each jackknife iteration.

Jackknife methods are useful for approximating MCSEs of other performance measures beyond just those for variance estimators. For instance, the jackknife is a convenient alternative for computing the MCSE of the standard error or

(raw) RMSE of a point estimator, which avoids the need to compute skewness or kurtosis. However, Boos and Osborne (2015) notes that the jackknife does not work for performance measures involving percentiles, including medians; for such statistics, bootstrapping techniques (discussed next) usually remains valid.

9.7.5 Bootstrapping MCSEs for relative performance measures

An alternative to the jackknife for estimating MCSEs is the bootstrap (Efron and Tibshirani, 1994). The bootstrap involves repeatedly resampling the set of simulation replications with replacement, calculating the performance measure of interest for each resample, and then calculating the standard deviation of the performance measures across the resamples.

To illustrate, we calculate a bootstrap MCSE for two quantities from our running cluster RCT example: the relative improvement in variance for Aggregation vs. Linear Regression, and the relative bias of Aggregation's variance estimator (i.e., how well calibrated it is).

We first calculate these performance measures for quick reference and for making confidence intervals after we bootstrap:

```
ests <- ClusterRCT_runs %>%
  group_by( method ) %>%
  summarise(
    SE2 = var( ATE_hat ),
    Vbar = mean( SE_hat^2 )
  ) %>%
  mutate( calib = Vbar / SE2,
          imp_var = SE2 / SE2[method=="LR"]  )
ests
```

```
# A tibble: 3 x 5
  method    SE2   Vbar calib imp_var
  <chr>   <dbl>  <dbl> <dbl>   <dbl>
1 Agg    0.0457 0.0515  1.13   0.914
2 LR     0.0500 0.0553  1.11   1
3 MLM    0.0454 0.0510  1.12   0.908
```

As before, we see that Aggregation has variance around 9% smaller than linear regression and that its variance estimator is inflated by around 13%. Let's make bootstrap confidence intervals on these two quantities to account for Monte Carlo error in our simulation.

We want to resample entire *simulation iterations* to take within-dataset correlations of the model estimates into account. To do this, we first pivot our raw simulation results to wide format:

```
runW <- ClusterRCT_runs %>%
  dplyr::select( runID, method, SE_hat, ATE_hat ) %>%
  rename( S=SE_hat, A=ATE_hat ) %>%
  tidyr::pivot_wider( names_from = method,
                      values_from = c( S, A ) )
print( runW, n = 4 )
```

```
# A tibble: 1,000 x 7
  runID S_MLM  S_LR S_Agg A_MLM  A_LR A_Agg
  <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1     1 0.196 0.189 0.198 0.128 0.164 0.106
2     2 0.185 0.199 0.181 0.181 0.295 0.147
3     3 0.253 0.210 0.261 0.132 0.204 0.122
4     4 0.209 0.190 0.211 0.556 0.769 0.501
# i 996 more rows
```

We now can simply bootstrap the rows, as so:

```
rps <- simhelpers::repeat_and_stack( 1000, {
  run_star = sample_n( runW, size = nrow(runW), replace = TRUE )
  tibble(
    cal_agg = mean( run_star$S_Agg^2 ) / var( run_star$A_Agg ),
    rel_imp_var = var( run_star$A_Agg ) / var( run_star$A_LR )
  )
})
```

For simplicity, we are only looking at the aggregated estimator—we could extend our code to bootstrap for everything. Regardless, once we have 1000 bootstrap iterations, we can evaluate how much our performance measures change across iterations:

```
rps %>%
  pivot_longer( everything(),
                names_to = "measure", values_to = "value" ) %>%
  group_by( measure ) %>%
  summarise(
    avg = mean( value ),
    MCSE = sd( value )
  ) %>%
  mutate(
    theta = c( ests$calib[[1]], ests$imp_var[[1]] ),
    CI_l = theta - 2 * MCSE,
    CI_u = theta + 2 * MCSE
  ) %>%
  relocate( measure, theta ) %>%
  knitr::kable( digits = 3 )
```

measure	theta	avg	MCSE	CI_l	CI_u
cal_agg	1.127	1.134	0.054	1.020	1.235
rel_imp_var	0.914	0.914	0.027	0.861	0.967

Interpreting the confidence interval in the second row, Aggregation has a lower variance than linear regression, with the upper end of the confidence interval being 97%. Second, we have evidence that the Aggregation variance estimator is systematically too high, with the lower end of the confidence interval being above 1 (but only just). If we had run fewer bootstrap iterations, we may well not have been able to make these claims with certainty.

The bootstrap is a very general tool. It can be applied to assess MCSE for nearly any performance measure you might encounter.

9.8 Performance Calculations with the `simhelpers` Package

The `simhelpers` package provides several functions for calculating most of the performance measures that we have reviewed, along with MCSEs for each. The functions are easy to use. Consider this set of simulation runs on the Welch dataset:

```
head(welch)
```

```
# A tibble: 6 x 9
     n1    n2 mean_diff method      est    var p_val lower_bound
  <dbl> <dbl>     <dbl> <chr>     <dbl>  <dbl> <dbl>       <dbl>
1    50    50         0 t-test  0.0258  0.0954 0.934      -0.587
2    50    50         0 Welch   0.0258  0.0954 0.934      -0.590
3    50    50         0 t-test  0.00516 0.0848 0.986      -0.573
4    50    50         0 Welch   0.00516 0.0848 0.986      -0.574
5    50    50         0 t-test -0.0798  0.0818 0.781      -0.647
6    50    50         0 Welch  -0.0798  0.0818 0.781      -0.649
# i 1 more variable: upper_bound <dbl>
```

We first assess the rejection rate for the traditional t-test based on the subset of simulation results with sample sizes of $n_1 = n_2 = 50$ and a mean difference of 0, using α levels of .01 and .05:

```
welch_sub <- filter(welch, method == "t-test",
                    n1 == 50, n2 == 50, mean_diff == 0 )

calc_rejection(welch_sub, p_values = p_val, alpha = c(.01, .05))
```

```
# A tibble: 1 x 5
  K_rejection rej_rate_01 rej_rate_05 rej_rate_mcse_01
        <int>       <dbl>       <dbl>            <dbl>
1        1000       0.009       0.048          0.00299
# i 1 more variable: rej_rate_mcse_05 <dbl>
```

The column labeled `K_rejection` reports the number of replications used to calculate the performance measures. The MCSE columns are the Monte Carlo uncertainty for the measures.

Here is the coverage rate calculated for the same condition:

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

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

The performance functions are designed to be used within a `tidyverse`-style workflow, including on grouped datasets. For instance, we can calculate the performance measures for all the methods for our given scenario:

```
welch_sub <- filter(welch, n1 == 50, n2 == 50,
                    mean_diff == 0 )
welch_sub %>%
  group_by( method ) %>%
  summarise(
    calc_rejection( p_values = p_val, alpha = .05 )
  )
```

```
# A tibble: 2 x 4
  method K_rejection rej_rate rej_rate_mcse
  <chr>        <int>    <dbl>         <dbl>
1 Welch         1000    0.047       0.00669
2 t-test        1000    0.048       0.00676
```

Later, when we analyze multifactor simulations, this grouping becomes even more powerful: we will group by the simulation factors and evaluate performance for all our scenarios at once.

9.8.1 Quick confidence interval coverage

In Section 6.1.2, we created the function `r_bivariate_Poisson()` for simulating data from a bivariate Poisson distribution. In Section 7.1, we developed

a function called `r_and_z()` for computing confidence intervals for Pearson's correlation using Fisher's z transformation. In Section 8.4, we created a simulation driver function, `sim_r_Poisson()` from these components. We can use this function to generate 1000 simulated confidence intervals:

```
bvp_CIs <- sim_r_Poisson( 1000, N = 30, rho = 0.4, mu1 = 8, mu2 = 14 )
head(bvp_CIs, n = 3)
```

```
  rep         r         z       CI_lo     CI_hi
1   1 0.4270785 0.4563179  0.07895797 0.6823579
2   2 0.4983680 0.5471325  0.16832012 0.7279383
3   3 0.3299007 0.3427168 -0.03446475 0.6168548
```

It will be useful to have a function that evaluates the coverage and width of these confidence intervals based on a simulated dataset. We can create such a function using `calc_coverage()`:

```
evaluate_CIs <- function(data, rho = 0) {
  simhelpers::calc_coverage(
    data = data,
    lower_bound = CI_lo, upper_bound = CI_hi,
    true_param = rho
  )
}
```

This function makes it quite easy to evaluate the output of `sim_r_Poisson()`:

```
evaluate_CIs(bvp_CIs, rho = 0.4)
```

```
# A tibble: 1 x 5
  K_coverage coverage coverage_mcse width width_mcse
       <int>    <dbl>         <dbl> <dbl>      <dbl>
1       1000    0.941       0.00745 0.602    0.00254
```

9.8.2 Quick Performance Calculations for our Cluster RCT Example

In Section 9.1.1, we computed performance measures for three point estimators of the school-level average treatment effect in a cluster RCT. We next carry out the same calculations using the `calc_absolute()` function from `simhelpers`, which also provides MCSEs. Examining the MCSEs is useful to ensure that 1000 replications of the simulation is suffiicent to provide reasonably precise estimates of the performance measures. We have:

```
library( simhelpers )

perf <- ClusterRCT_runs %>%
  group_by(method) %>%
```

```
  summarise(
    calc_absolute(
      estimates = ATE_hat, true_param = ATE,
      criteria = c("bias","stddev", "rmse")
    )
  )
perf %>%
  dplyr::select( -K_absolute ) %>%
  knitr::kable( digits = 3 )
```

method	bias	bias_mcse	stddev	stddev_mcse	rmse	rmse_mcse
Agg	0.000	0.007	0.214	0.005	0.214	0.006
LR	0.082	0.007	0.224	0.005	0.238	0.006
MLM	0.012	0.007	0.213	0.005	0.213	0.006

The MCSEs are small relative to the linear regression bias term and all of the SEs (`stddev`) and RMSEs. Results based on 1000 replications seems adequate to support our conclusions about the gross trends identified. We cannot rule out the possibility that the aggregation estimator and multilevel modeling estimator are slightly biased: they could have true bias of as much as 0.014 (two MCSEs).

For our variance estimators, we can use the `calc_relative_var()` function to compute performance measures and MCSEs using the jackknife:

```
perf <- ClusterRCT_runs %>%
  group_by(method) %>%
  summarise(
    calc_relative_var(
      estimates = ATE_hat,
      var_estimates = SE_hat^2,
      criteria = c("relative bias", "relative rmse")
    )
  )
perf %>%
  dplyr::select( -K_relvar ) %>%
  knitr::kable( digits = 2 )
```

method	rel_bias_var	rel_bias_var_mcse	rel_rmse_var	rel_rmse_var_mcse
Agg	1.13	0.05	0.40	0.03
LR	1.11	0.05	0.47	0.03
MLM	1.12	0.05	0.40	0.03

We can use these calculations to create confidence intervals for the performance measures:

```
perf %>%
  dplyr:: select( method, rel_bias_var, rel_bias_var_mcse ) %>%
  mutate( CI_l = rel_bias_var - 2 * rel_bias_var_mcse,
          CI_u = rel_bias_var + 2 * rel_bias_var_mcse ) %>%
  knitr::kable( digits = 2 )
```

method	rel_bias_var	rel_bias_var_mcse	CI_l	CI_u
Agg	1.13	0.05	1.02	1.23
LR	1.11	0.05	1.00	1.21
MLM	1.12	0.05	1.02	1.23

We get essentially the same results as what we found in Section 9.7.5 using bootstrapping.

9.9 Concluding thoughts

The performance of an estimator is not well captured by any single number. As we have seen, estimators have multiple properties, with bias and precision being only two of many. Table 9.2 lists the main measures we have introduced for point estimators; each captures a different aspect of performance, and their importance depends on how the estimator will be used.

Beyond point estimates, most data analysis procedures also produce quantities such as standard errors, confidence intervals, and p-values. These outputs are closely related. For example, a confidence interval is typically constructed from a point estimate and its standard error, so its performance depends on both the bias of the estimator and the accuracy of the standard error. Likewise, the performance of a hypothesis testing procedure will often strongly depend on the properties of the point estimator and standard error used to compute the test. As a result, simulation studies typically evaluate performance using a collection of measures to capture these interrelated aspects and to compare procedures more fully.

In many simulation studies, the goal is also to understand how performance varies across different data-generating processes (DGPs). In the next chapters, we expand from a single scenario to multiple scenarios. By looking at different performance measures change across scenario, we can begin to understand when and why different methods perform well or poorly.

Table 9.2: Performance measures for evaluating point estimators

Criterion	Definition	Estimator	Monte Carlo Standard Error
Bias	$\mathbb{E}(T) - \theta$	$\bar{T} - \theta$	Equation 9.24
Median bias	$\mathbb{M}(T) - \theta$	$m_T - \theta$	Equation 9.31
Variance	$\mathbb{E}\left[(T - \mathrm{E}(T))^2\right]$	S_T^2	Equation 9.27
Standard error	$\sqrt{\mathbb{E}\left[(T - \mathrm{E}(T))^2\right]}$	S_T	Equation 9.29
Mean squared error	$\mathbb{E}\left[(T - \theta)^2\right]$	$\left(\bar{T} - \theta\right)^2 + \frac{R-1}{R} S_T^2$	Equation 9.25
Root mean squared error	$\sqrt{\mathbb{E}\left[(T - \theta)^2\right]}$	$\sqrt{\left(\bar{T} - \theta\right)^2 + \frac{R-1}{R} S_T^2}$	Equation 9.26
Median absolute error	$\mathbb{M}\left[\lvert T - \theta \rvert\right]$	$\left[\lvert T - \theta \rvert\right]_{R/2}$	Equation 9.31
Relative bias	$\mathbb{E}(T)/\theta$	$\bar{T}/\theta$	Equation 9.30
Relative median bias	$\mathbb{M}(T)/\theta$	m_T/θ	Equation 9.30
Relative RMSE	$\sqrt{\mathbb{E}\left[(T - \theta)^2\right]}/\theta$	$\frac{\sqrt{\left(\bar{T} - \theta\right)^2 + \frac{R-1}{R} S_T^2}}{\theta}$	Equation 9.30

Exercises

Exercise 9.1 (Brown and Forsythe (1974) results)**.**

1. Use the `generate_ANOVA_data` data-generating function for one-way heteroskedastic ANOVA (Section 5.1) and the data-analysis function you wrote for Exercise 7.1 to create a simulation driver function for the Brown and Forsythe simulations.

2. Use your simulation driver to evaluate the Type-I error rate of the ANOVA F-test, Welch's test, and the BFF* test for a scenario with four groups, sample sizes $n_1 = 11$, $n_2 = 16$, $n_3 = 16$, $n_4 = 21$, equal group means $\mu_1 = \mu_2 = \mu_3 = \mu_4 = 0$, and group standard deviations $\sigma_1 = 3$, $\sigma_2 = 2$, $\sigma_3 = 2$, $\sigma_4 = 1$. Are all of the tests level-α?

3. Use your simulation driver to evaluate the power of each test

for a scenario with group means of $\mu_1 = 0$, $\mu_2 = 0.2$, $\mu_3 = 0.4$, $\mu_4 = 0.6$, with sample sizes and group standard deviations as listed above. Which test has the highest power?

Exercise 9.2 (Size-adjusted power)**.** When different hypothesis testing procedures have different rejection rates under the null hypothesis, it becomes difficult to interpret differences in the non-null power of the tests. One approach for conducting a fair comparison of testing procedures in this situation is to compute the *size-adjusted* power of the tests. Size-adjusted power involves computing the rejection rate of a test using a different threshold α', selected so that the Type-I error rate of the test is equal to the desired α level. Specifically, size adjusted power is

$$\rho_\alpha^{adjusted}(\theta) = \Pr(P < \rho_\alpha(0)).$$

To estimate size-adjusted power using simulation, we first need to estimate the Type-I error rate, $r_\alpha(0)$. We can then evaluate the rejection rate of the testing procedure under scenarios with other values of θ by computing

$$r_\alpha^{adjusted}(\theta) = \frac{1}{R}\sum_{r=1}^{R} I(P_r < r_\alpha(0)).$$

Compute the size-adjusted power of the ANOVA F-test, Welch's test, and the BFF* test for the scenario in part (3) of Exercise 9.1.

Exercise 9.3 (Three correlation estimators)**.** Consider the bivariate negative binomial distribution model described in Exercise 6.6. Suppose that we want to estimate the correlation parameter ρ of the latent bivariate normal distribution. Without studying the statistical theory for this problem, we might think to use simulation to evaluate whether any common correlation measures work well for estimating this parameter. Potential candidate estimators include the usual sample Pearson's correlation, Spearman's rank correlation, or Kendall's τ coefficient. The latter two estimators might seem promising because they are based on the ranked data, so could be more appropriate that Pearson's correlation for frequency count variates.

The following estimation function computes all three correlations, along with corresponding p-values for the null hypothesis of no association and confidence intervals computed using Fisher's z transformation. The confidence interval calculations are developed for Pearson's correlation under bivariate normality, so they might not be appropriate for this data-generating process or for Spearman's or Kendall's correlations. Still, we can use simulation to see how well or how poorly they perform.

```r
three_corrs <- function(
  data,
  method = c("pearson","kendall","spearman"),
  level = 0.95
) {

  r_est <- lapply(method, \(m) cor.test(data$C1, data$C2,
                                        method = m, exact = FALSE))
  est <- sapply(r_est, \(x) as.numeric(x$estimate))
  pval <- sapply(r_est, \(x) x$p.value)
  z_est <- atanh(est)
  se_z <- 1 / sqrt(nrow(data) - 3)
  crit <- qnorm(1 - (1 - level) / 2)
  ci_lo <- tanh(z_est - crit * se_z)
  ci_hi <- tanh(z_est + crit * se_z)

  data.frame(
    stat = method,
    r = est,
    z = z_est,
    se_z = se_z,
    pval = pval,
    ci_lo = ci_lo,
    ci_hi = ci_hi
  )

}
```

1. Combine your data-generating function and `three_corrs()` into a simulation driver.

2. Use your simulation driver to generate 500 replications of the simulation for a scenario with $N = 20$, $\mu_1 = \mu_2 = 5$, $p_1 = p_2 = 0.5$, and $\rho = 0.7$.

3. Compute the bias, standard error, and RMSE of each correlation estimator, along with corresponding MCSEs. Which correlation estimator is most accurate?

4. Compute the coverage rate and expected width of the confidence intervals based on each correlation estimator. Do any of the estimators have reasonable coverage? If so, which has the best expected width?

5. Use your simulation driver to estimate the Type-I error rate of the hypothesis tests for each correlation coefficient for a scenario

with $N = 20$, $\mu_1 = \mu_2 = 5$, $p_1 = p_2 = 0.5$. Are any of the tests level-α?

Exercise 9.4 (Confidence interval comparison)**.** Consider the estimation functions for the cluster RCT example, as given in Section 7.2. Modify the functions to return **both** normal Wald-type 95% confidence intervals (as computed in Section 9.3.1) and cluster-robust confidence intervals based on t distributions with Satterthwaite degrees of freedom. For the latter, use `conf_int()` from the `clubSandwich` package, as in the following example code:

```
library(clubSandwich)

M1 <- lme4::lmer(
  Yobs ~ 1 + Z + (1 | sid),
  data = dat
)
conf_int(M1, vcov = "CR2")

M2 <- estimatr::lm_robust(
  Yobs ~ 1 + Z, data = dat,
  clusters = sid,  se_type = se_type
)
conf_int(M2, cluster = dat$sid, vcov = "CR2")

M3 <- estimatr::lm_robust(
  Ybar ~ 1 + Z, data = datagg,
  se_type = se_type
)
conf_int(M3, cluster = dat$sid, vcov = "CR2")
```

Pick some simulation parameters and estimate the coverage and interval width of both types of confidence intervals. How do the normal Wald-type intervals compare to the cluster-robust intervals?

Exercise 9.5 (Jackknife calculation of MCSEs for RMSE)**.** The following code generates 100 replications of a simulation of three average treatment effect estimators in a cluster RCT, using a simulation driver function we developed in Section 8.3 using components described in Section 6.6 and Section 7.2.

```
set.seed( 20251029 )
runs_val <- run_RCT_sim(
  reps = 100,
  J = 16, n_bar = 20, alpha = 0.5,
  ATE = 0.3, ICC = 0.2 )
```

Compute the RMSE of each estimator, and use the jackknife technique described in Section 9.7.4 to compute a MCSE for the RMSE. Check your results

against the results from `calc_absolute()` in the `simhelpers` package.

Exercise 9.6 (Jackknife calculation of MCSEs for RMSE ratios)**.** Continuing from Exercise 9.5, compute the ratio of the RMSE of each estimator to the RMSE of the linear regression estimator. Use the jackknife technique to compute a MCSE for these RMSE ratios.

Exercise 9.7 (Distribution theory for person-level average treatment effects)**.** Section 6.6 described a data-generating process for a cluster-randomized experiment in which the school-specific treatment effects varied according to the size of the school. The auxiliary model for the size of school j was

$$n_j \sim \text{Unif}\left[(1-\alpha)\bar{n}, (1+\alpha)\bar{n}\right],$$

where $\bar{n}$ was the average school size and $0 \leq \alpha < 1$ determines the degree of variation in school sizes. The data-generating process for the outcome data was

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

where Y_{ij} is the outcome for student i in school j, Z_j is an indicator for whether school j is assigned to treatment ($Z_j = 1$) or control ($Z_j = 0$), and $S_j = \frac{n_j - \bar{n}}{\bar{n}}$. The error terms u_j and e_{ij} are both assumed to be normally distributed with zero means.

Under this model, the average treatment effect for school j is $\tau_j = \gamma_1 + \gamma_2 S_j$. Because $\mathbb{E}(S_j) = 0$ by construction, the average of the school-specific treatment effects is γ_1. This is the school-level population average treatment effect estimate. But what is the student-level population average treatment effect estimate? Use the properties of the uniform distribution to find the student-level population average treatment effect $\mathbb{E}\left(\frac{n_j}{\bar{n}} \times \tau_j\right)$. Check your derivation by simulating a large sample of school sizes and school-specific treatment effects.

Part III

Systematic Simulations

10

Simulating across multiple scenarios

In Chapter 4, we described the general structure of basic simulations as following four steps: generate, analyze, repeat, and summarize, with each step represented by its own function or set of code. We also saw how to bundle these steps into a simulation driver that runs our entire simulation. With these tools, it is easy to run a simulation for a single combination of parameter values. In practice, however, we want to examine a range of different values, to see how things change as we vary the levels of our focal parameter values, auxiliary parameters, or sample size (or other design parameters). We might want to do this to gain assurance that our findings are general, and we might want to understand what aspects of the context affect the performance of the estimator or estimators we are studying. 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 hope to understand trade-offs, general rules, and limits.

This brings us to the remaining piece of the simulation puzzle: the study's experimental design. In this section of the book, we walk through three further steps for a systematic simulation: *design* a set of scenarios to examine, *execute* across multiple scenarios, and *synthesize* the performance results across the scenarios explored. The principles of tidy simulation apply to these steps as well.

In this chapter, we examine the "execute" step. We demonstrate how to create a dataset representing the experimental design of the simulation as a set of scenarios to run, how to execute a simulation driver across those identified scenarios, and how to organize the results for synthesis. Our focus here is on the programming techniques and computational structure. In the following chapter, we go deeper into "design," and discuss some of the deeper theoretical challenges of designing multifactor simulations, and provide some worked case studies to bring these ideas to life. In subsequent chapters, we turn to the "synthesize" step and examine tools for making sense of the full set of results from a multifactor simulation.

For all of these steps we focus on **full factorial** designed experiments, a common design used in Monte Carlo simulation. 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. Before we dig into that, we start with a simpler case of a single-factor simulation, where we vary only one factor across multiple levels. Even this simple, single-factor design is enough to illustrate the basic principles of systematic simulation and to demonstrate core elements of the code architecture.

10.1 Simulating across levels of a single factor

Even if we are only using simulation in an ad hoc, exploratory way, we will often be interested in examining how something changes as we change some feature of interest. For example, in Chapter 3, we looked at how the coverage rate of a confidence interval for the mean of an exponential distribution changed as a function of the sample size. To explore this, we ran a simulation across a set of sample sizes ranging from 10 to 300, and we found that the coverage rate improved towards the desired level as sample size increased.

Simple forms of systematic exploration such as this are useful in many situations. For instance, when using Monte Carlo simulation for study planning, we might examine simulated power over a range of the target parameter to identify the smallest parameter that provides power above a desired level. If we are using simulation simply to study an unfamiliar model, we might vary a key parameter over a wide range to see how the performance of an estimator changes. These forms of exploration are single-factor simulations.

To demonstrate, we revisit the case study on heteroskedastic analysis of variance, as studied by Brown and Forsythe (1974) and developed in Chapter 5. Our goal is to understand how the power of Welch's test varies as a function of the maximum distance between group means.

We do this in four steps:

1) Make a function to run a single scenario
2) Make a table of all the scenarios we want to run
3) Run our single-scenario function for each scenario on the table.
4) Analyze the results (with a plot)

10.1.1 Making a function to run a single scenario

Back in Chapter 5, before we knew much, we developed a function, `generate_ANOVA_data()`, to generate data for a heteroskedastic ANOVA scenario, and a function, `ANOVA_Welch_F()`, to analyze the resulting data. We

then built our simulation and analysis by hand. Before we move on to multiple scenarios, let us clean this work up by writing a simulation driver that runs a simulation and give results as a single call.

`generate_ANOVA_data()` takes an arbitrary vector of means per group. Our goal is, however, to have a single "knob" that we can turn to increase the spread of groups in the generated data. To do this, we will rewrite our function to take a single value, maximum difference (`max_diff`). Inside our new function we will generate group means that are equally spaced between zero and this value. We will also re-parameterize the function in terms of the total sample size (`N`) and the fraction of observations allocated to each group (`allocation`) to make calling it simpler.

The revised function is

```
generate_ANOVA_new <- function(
  G, max_diff, sigma_sq = 1, N = 20, allocation = "equal"
) {

  mu <- seq(0, max_diff, length.out = G)
  if (identical(allocation, "equal")) {
    allocation <- rep(1 / G, times = G)
  } else {
    allocation <- rep(allocation, length.out = G)
    allocation <- allocation / sum(allocation)
  }

  N_g <- round(N * allocation)
  sigma <- rep(sqrt(sigma_sq), length.out = G)

  group <- factor(rep(1:G, times = N_g))
  mu_long <- rep(mu, times = N_g)
  sigma_long <- rep(sigma, times = N_g)

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

  return(sim_data)
}
```

We next create an initial simulation driver by combining `generate_ANOVA_new()` with the data-analysis function we created in Section 5.2:

```
sim_ANOVA <- bundle_sim(f_generate = generate_ANOVA_new,
                        f_analyze = ANOVA_Welch_F)
```

Our simulation driver does not give us final performance, however, just the

raw results:

```
sims <- sim_ANOVA( 100, G = 4, max_diff = 1,
           sigma_sq = c(1, 2, 2, 3), N = 40 )
sims
```

```
# A tibble: 100 x 2
     ANOVA   Welch
     <dbl>   <dbl>
 1 0.444   0.600
 2 0.0341  0.0205
 3 0.129   0.100
 4 0.494   0.374
 5 0.540   0.594
 6 0.0611  0.115
 7 0.529   0.685
 8 0.914   0.922
 9 0.00566 0.00842
10 0.528   0.589
# i 90 more rows
```

Using what we learned in Chapter 9, we can calculate power as so:

```
calc_rejection(sims, p_values = Welch, alpha = c(.01, .05))
```

```
# A tibble: 1 x 5
  K_rejection rej_rate_01 rej_rate_05 rej_rate_mcse_01
        <int>       <dbl>       <dbl>            <dbl>
1         100        0.15        0.29           0.0357
# i 1 more variable: rej_rate_mcse_05 <dbl>
```

We want power for both the Welch test and the conventional ANOVA F test, so we can compare them, so we write a small function to generate a table of power levels for both tests across the alpha levels we are interested in. In order to easily incorporate it into our pipeline, our function takes a set of simulation results as input and provide a dataset of performance measures as output:

```
summarize_power <- function(data, alpha = c(.01,.05)) {
  ANOVA <- calc_rejection(data, p_values = ANOVA, alpha = alpha,
                          format = "long")
  Welch <- calc_rejection(data, p_values = Welch, alpha = alpha,
                          format = "long")
  bind_rows(
    ANOVA = ANOVA,
    Welch = Welch,
    .id = "test"
  )
}
```

We get back a nice tibble of results when we use it:

```
summarize_power( sims )
```

```
# A tibble: 4 x 5
  test  K_rejection alpha rej_rate rej_rate_mcse
  <chr>       <int> <dbl>    <dbl>         <dbl>
1 ANOVA         100  0.01     0.13        0.0336
2 ANOVA         100  0.05     0.33        0.0470
3 Welch         100  0.01     0.15        0.0357
4 Welch         100  0.05     0.29        0.0454
```

Our next step is to wrap all of the above into a single call. The `bundle_sim()` function from `simhelpers` will actually create such a function for us, combining a performance calculation function along with the data-generating and data-analysis functions. In particular, the `bundle_sim()` function will take any summarizing function that takes in a set of simulation results and gives back a table of performance measures, which is just what `summarize_power()` does. Witness!

```
sim_ANOVA <- bundle_sim(
  f_generate = generate_ANOVA_new,
  f_analyze = ANOVA_Welch_F,
  f_summarize = summarize_power
)
sim_ANOVA(  100, G = 4, max_diff = 1,
  sigma_sq = c(1, 2, 2, 3), N = 40,
  alpha = c(.01, .05) )
```

```
# A tibble: 4 x 5
  test  K_rejection alpha rej_rate rej_rate_mcse
  <chr>       <int> <dbl>    <dbl>         <dbl>
1 ANOVA         100  0.01     0.08        0.0271
2 ANOVA         100  0.05     0.25        0.0433
3 Welch         100  0.01     0.11        0.0313
4 Welch         100  0.05     0.26        0.0439
```

Now we can apply our simulation driver to several different scenarios with different values of `max_diff`.

10.1.2 Making a table of scenarios to run

Following the principles of tidy simulation, it is useful to represent the design of a systematic simulation as a dataset with a row for each scenario to be considered. For a single-factor simulation, the experimental design consists of a dataset with just a single variable:

```
Welch_design <- tibble(max_diff = seq(0, 2, 0.2))
str(Welch_design)
```

```
tibble [11 x 1] (S3: tbl_df/tbl/data.frame)
 $ max_diff: num [1:11] 0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 ...
```

For a single factor simulation, making our schedule of simulations to run is easy. This step will get more involved with multi-factor simulations, as described in Section 10.2.

10.1.3 Running the simulation across scenarios

To compute simulation results for each scenario on our list, we can use the `map()` function from the `purrr` package. `map()` takes a list of values as the input, then calls a function on each value. Our `sim_ANOVA()` function has several further arguments that need to be specified. Because these will be the same for every value of `max_diff`, we can include them as additional arguments in `map()`, and they will be used every time `sim_ANOVA()` is called. Here is one way to code this:

```
Welch_results <-
  Welch_design %>%
  mutate(
    pvals = map(max_diff, sim_ANOVA, reps = 100,
                G = 4, sigma_sq = c(1, 2, 2, 3),
                N = 40)
  )
```

Another way to accomplish the same thing is to specify an anonymous function (also called a lambda) in the `map()` call. This syntax makes it clearer that the additional arguments are getting called in every evaluation of `sim_ANOVA()`:

```
Welch_results <-
  Welch_design %>%
  mutate(
    pvals = map(max_diff, ~ sim_ANOVA(100, max_diff = .x,
                                      G = 4, sigma_sq = c(1, 2, 2, 3),
                                      N = 40))
  )
```

In the resulting dataset, the `pvals` variable is a list, with each entry consisting of a tibble of our results:

```
Welch_results
```

```
# A tibble: 11 x 2
   max_diff pvals
      <dbl> <list>
```

```
 1      0   <tibble [4 x 5]>
 2      0.2 <tibble [4 x 5]>
 3      0.4 <tibble [4 x 5]>
 4      0.6 <tibble [4 x 5]>
 5      0.8 <tibble [4 x 5]>
 6      1   <tibble [4 x 5]>
 7      1.2 <tibble [4 x 5]>
 8      1.4 <tibble [4 x 5]>
 9      1.6 <tibble [4 x 5]>
10      1.8 <tibble [4 x 5]>
11      2   <tibble [4 x 5]>
```

The above code and output may look a bit peculiar: we are storing a set of dataframes (our result) in our original dataframe as a new "variable." This is actually ok in R: our results will be in what is called a **list-column**, where each element in our list column is the set of simulation results for the given scenario. For instance, if we want to examine the results from the third scenario, we can pull it out as follows:

```
Welch_results$pvals[[3]]
```

```
# A tibble: 4 x 5
  test  K_rejection alpha rej_rate rej_rate_mcse
  <chr>       <int> <dbl>    <dbl>         <dbl>
1 ANOVA         100  0.01     0.03        0.0171
2 ANOVA         100  0.05     0.07        0.0255
3 Welch         100  0.01     0.03        0.0171
4 Welch         100  0.05     0.05        0.0218
```

10.1.4 Analyzing the results

List columns are neat, but hard to work with. To turn the list-column into normal data, we can use `unnest()` to expand the `pvals` variable, replicating the values of the main variables once for each row in the nested dataset:

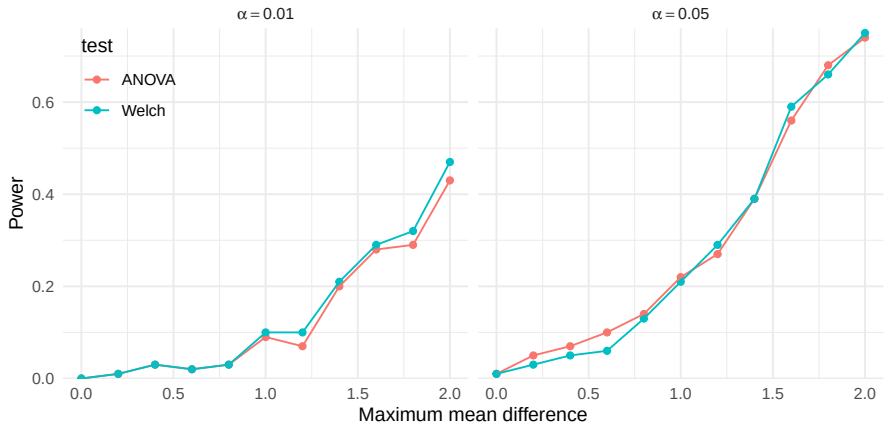
```
Welch_results_long <- unnest(Welch_results, cols = pvals)
Welch_results_long
```

```
# A tibble: 44 x 6
   max_diff test  K_rejection alpha rej_rate rej_rate_mcse
      <dbl> <chr>       <int> <dbl>    <dbl>         <dbl>
 1      0   ANOVA         100  0.01     0           0
 2      0   ANOVA         100  0.05     0.01        0.00995
 3      0   Welch         100  0.01     0           0
 4      0   Welch         100  0.05     0.01        0.00995
 5      0.2 ANOVA         100  0.01     0.01        0.00995
 6      0.2 ANOVA         100  0.05     0.05        0.0218
```

```
 7      0.2 Welch         100  0.01     0.01       0.00995
 8      0.2 Welch         100  0.05     0.03       0.0171
 9      0.4 ANOVA         100  0.01     0.03       0.0171
10      0.4 ANOVA         100  0.05     0.07       0.0255
# i 34 more rows
```

The final dataset has 44 rows, consisting of 4 different result rows for each of 11 scenarios. The results are also organized in a way that facilitates visualization of the power levels:

```
ggplot(Welch_results_long) +
  aes(max_diff, rej_rate, color = test) +
  geom_point() + geom_line() +
  scale_y_continuous(limits = c(0, NA), expand = expansion(0,c(0,0.01))) +
  facet_wrap(~ alpha, labeller = label_bquote(alpha == .(alpha))) +
  labs(x = "Maximum mean difference", y = "Power") +
  theme_minimal() +
  theme(legend.position  ="inside", legend.position.inside = c(0.08,0.85))
```



Under the conditions examined here, both tests appear to have similar power. Although the tests appear to work similarly here, these results are based on a very specific set of conditions, including equally sized groups and a specific configuration of within-group variances. A natural further question is whether this pattern holds under other configurations of sample allocations, total sample size, or within-group variances. These questions can be examined by expanding the simulation design to further scenarios.

10.2 Simulating across multiple factors

We just wrote a simulation to systematically explore a range of values for a single factor. We next consider systematically exploring multiple factors together.

For this example, recall our simulation study examining the performance of confidence intervals for Pearson's correlation coefficient under a bivariate Poisson distribution. We first examined this data-generating model in Section 6.1.2, implementing it in the function `r_bivariate_Poisson()`. The model has three parameters (the means of each variate, μ_1, μ_2 and the correlation ρ) and there is one design parameter (sample size, N). Thus, we could in principle examine up to four factors: μ_1, μ_2, ρ, and N.

As a first pass, we might consider the following values:

- the sample size, with values of $N = 10, 20$, or 30
- the mean of the first variate, with values of $\mu_1 = 4, 8$, or 12
- the mean of the second variate, with values of $\mu_2 = 4, 8$, or 12
- the true correlation, with values ranging from $\rho = 0.0$ to 0.7 in steps of 0.1

All combinations of our set of factors gives a $3 \times 3 \times 3 \times 8$ factorial design, where each element is the number of levels for that factor. We call this design a four-factor experiment, because we have four different things we are varying.

Using these parameters directly as factors in the simulation design will lead to considerable redundancy because of the symmetry of the model: generating data with $\mu_1 = 10$ and $\mu_2 = 5$ would lead to identical correlations as using $\mu_1 = 5$ and $\mu_2 = 10$. It is useful to re-parameterize to reduce redundancy and simplify things. We will therefore restrict our simulation conditions and always have μ_1 as the larger variate. We can then reparameterize our factors, and specify the ratio of the smaller to the larger mean as $\lambda = \mu_2/\mu_1$. We might then examine the following factors:

- the sample size, with values of $N = 10, 20$, or 30
- the mean of the larger variate, with values of $\mu_1 = 4, 8$, or 12
- the ratio of means, with values of $\lambda = 0.5$ or 1.0.
- the true correlation, with values ranging from $\rho = 0.0$ to 0.7 in steps of 0.1

We now have a $3 \times 3 \times 2 \times 8$ factorial design.

To implement this design in code, we first save the simulation parameters as a list with one entry per factor, where each entry consists of the levels that we would like to explore. We will run a simulation for every possible combination of these values. Here is code that generates all of the scenarios given the above design, storing these combinations in a data frame, `params`, that represents the full experimental design:

```
design_factors <- list(
  N = c(10, 20, 30),
  mu1 = c(4, 8, 12),
  lambda = c(0.5, 1.0),
  rho = seq(0.0, 0.7, 0.1)
)

lengths(design_factors)
```

```
     N    mu1 lambda    rho
     3      3      2      8
```

```
params <- expand_grid( !!!design_factors )
glimpse(params)
```

```
Rows: 144
Columns: 4
$ N      <dbl> 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,~
$ mu1    <dbl> 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,~
$ lambda <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.~
$ rho    <dbl> 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.0, 0.~
```

We use `expand_grid()` from the `tidyr` package to create all possible combinations of the four factors.[1] We have a total of $3 \times 3 \times 2 \times 8 = 144$ rows, each row corresponding to a simulation scenario to explore. With multifactor experiments, it is easy to end up running a lot of experiments!

10.3 Using pmap to run multifactor simulations

Once we have selected factors and levels for simulation, we now need to run the simulation code across all of our factor combinations. Conceptually, each

[1] `expand_grid()` is set up to take one argument per factor of the design. A clearer example of its natural syntax is:

```
params <- expand_grid(
  N = c(10, 20, 30),
  mu1 = c(4, 8, 12),
  lambda = c(0.5, 1.0),
  rho = seq(0.0, 0.7, 0.1)
)
```

However, we generally find it useful to create a list of design factors before creating the full grid of parameter values, so we prefer to make `design_factors` first. To use `expand_grid()` on a list, we need to use `!!!`, the splice operator from the `rlang` package, which treats `design_factors` as a set of arguments to be passed to `expand_grid`. The syntax does look a bit wacky, but it is succinct and useful.

row of our `params` dataset represents a single simulation scenario, and we want to run our simulation code for each of these scenarios. We would thus call our simulation function, using all the values in that row as parameters to pass to the function.

One way to call a function on each row of a dataset in this manner is by using `pmap()` from the `purrr` package. `pmap()`, an extension of the `map()` function from above, 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.[2] Because R's `data.frame` objects are also sets of lists (where each variable is a vector, which is a simple form of list), `pmap()` also works seamlessly on `data.frame` or `tibble` objects.

Here is a small illustration of `pmap()` in action:

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

args_data <- tibble( a = 1:3, b = 5:7, theta = c(0.2, 0.3, 0.7) )
purrr::pmap_dbl( args_data, .f = some_function, scale = 10 )
```

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

One important constraint of `pmap()` is that the variable names over which to iterate over must correspond *exactly* to arguments of the function to be evaluated. In the above example, `args_data` must have column names that correspond to the arguments of `some_function`. For functions with additional arguments that are not manipulated, extra parameters can be passed after the function name, just as with `map()` (as in the `scale` argument in this example). Extra arguments will be passed to each function call, and be the same for all calls.

10.3.1 Making a function to run a single scenario

We use the core idea of `pmap()`—call a function for each row of a dataset—to run our simulation for each row of our schedule of scenarios. Analogous to the single-factor approach, we first make a driver to run a single scenario, and then use `pmap()` to run this driver for each row of `params`. We again use `bundle_sim()` as we did for the single-factor simulation.

In Section 7.1, we developed a function called `r_and_z()` for computing

[2] Just like `map()` or `map2()`, `pmap()` has variants such as `_dbl` or `_dfr`. These variants automatically stack or convert the list of things returned into a tidier collection (for `_dbl` it will convert to a vector of numbers, for `_dfr` it will stack the results to make a large tibble, assuming each returned item is a little tibble).

confidence intervals for Pearson's correlation using Fisher's z transformation, and in Section 9.8.1, we wrote a function called `evaluate_CIs()` for evaluating confidence interval coverage and average width. We bundle `r_bivariate_Poisson()`, `r_and_z()`, and `evaluate_CIs()` into a simulation driver as follows:

```
Pearson_sim <- bundle_sim(
  f_generate = r_bivariate_Poisson, f_analyze = r_and_z, f_summarize = evaluate_CIs
)
args(Pearson_sim)
```

```
function (reps, N, mu1, mu2, rho = 0, seed = NA_integer_, summarize = TRUE)
NULL
```

`Pearson_sim()` will run a simulation for any given scenario:

```
Pearson_sim(1000, N = 10, mu1 = 5, mu2 = 5, rho = 0.3)
```

```
# A tibble: 1 x 5
  K_coverage coverage coverage_mcse width width_mcse
       <int>    <dbl>         <dbl> <dbl>      <dbl>
1       1000    0.947       0.00708  1.10    0.00557
```

In order to call `Pearson_sim()` with `pmap()`, we need to ensure that each column of `params` exactly corresponds to an argument in `Pearson_sim()`. In particular, because we re-parameterized the model in terms of λ, we will need to compute the parameter value for μ_2 and remove the `lambda` variable as it is not an argument of `Pearson_sim()`:

```
params_mod <- params %>%
  mutate(mu2 = mu1 * lambda) %>%
  dplyr::select(-lambda)
```

Now we can use `pmap()` to run the simulation for all 144 parameter settings:

```
sim_results <- params
sim_results$res <- pmap(params_mod, Pearson_sim, reps = 1000 )
unnest( sim_results, cols ="res" ) %>%
  dplyr::select( -K_coverage, -coverage_mcse, -width_mcse )
```

We can combine the above into a single pipeline as so:

```
sim_results <-
  params %>%
  mutate(
    mu2 = mu1 * lambda,
    reps = 1000
  ) %>%
  mutate(
```

```
    res = pmap(dplyr::select(., -lambda), .f = Pearson_sim)
  ) %>%
  unnest(cols = res)
```

As an alternative approach, the `evaluate_by_row()` function from the `simhelpers` package provides the same functionality but with a more direct syntax:

```
sim_results <-
  params %>%
  mutate( mu2 = mu1 * lambda ) %>%
  evaluate_by_row( Pearson_sim, reps = 1000 )
```

An advantage of `evaluate_by_row()` is that the input dataset can include extra variables (such as `lambda`). Another advantage is that it is easy to run the calculations in parallel; see Chapter 18.

As a final step, we save our results using tidyverse's `write_rds()` (for background on this function, see R for Data Science, Section 7.5). We first ensure we have a directory by making one via `dir.create()` (see Section 16.4 for more on files):

```
dir.create( "results", showWarnings = FALSE )
write_rds( sim_results, file = "results/Pearson_Poisson_results.rds" )
```

We now have a complete set of simulation results for all of the scenarios we specified.

10.4 When to calculate performance metrics

For a single-scenario simulation, we repeatedly generate and analyze data, and then assess the performance across the repetitions. When we extend this process to multifactor simulations, we have a choice: do we compute performance measures for each simulation scenario as we go (inside) or do we compute all of them after we get all of our individual results (outside)? There are pros and cons to each approach.

10.4.1 Aggregate as you simulate (inside)

The *inside* approach runs a stand-alone simulation for each scenario of interest. For each combination of factors, we simulate data, apply our estimators, assess performance, and return a table with summary performance measures. We can then stack these tables to get a dataset with all of the results, ready for analysis.

This is the approach we illustrated above. It is straightforward and streamlined: we already have a method to run simulations for a single scenario, and we just repeat it across multiple scenarios and combine the outputs. After calling `pmap()` (or `evaluate_by_row()`) and stacking the results, we end up with a dataset containing all the simulation conditions, one simulation context per row (or maybe we have sets of several rows for each simulation context, with one row for each method), with the columns consisting of the simulation factors and calculated performance measures. This table of performance measures is exactly what we need to conduct further analysis and draw conclusions about how the estimators work.

The primary advantages of the inside strategy are that it is easy to modularize the simulation code and that it produces a compact dataset of results, minimizing the number and size of files that need to be stored. On the con side, calculating summary performance measures inside of the simulation driver limits our ability to add new performance measures on the fly or to examine the distribution of individual estimates. For example, suppose we wanted to check if the distribution of Fisher-z estimates in a particular scenario was right-skewed, perhaps because we are worried that the estimator sometimes breaks down. We might want to make a histogram of the point estimates, or calculate the skew of the estimates as a performance measure. Because the individual estimates are not saved, we would have no way of investigating these questions without rerunning the simulation for that condition. In short, the inside strategy minimizes disk space but constrains our ability to explore or revise performance calculations.

10.4.2 Keep all simulation runs (outside)

The *outside* approach involves retaining the entire set of estimates from every replication, with each row corresponding to an estimate for a given simulated dataset. The benefit of the outside approach is that it allows us to add or change how we calculate performance measures without re-running the entire simulation. This is especially important if the simulation is time-intensive, such as when the estimators being evaluated are computationally expensive. The primary disadvantage the outside approach is that it produces large amounts of data that need to be stored and further manipulated. Thus, the outside strategy maximizes flexibility, at the cost of increased dataset size.

In our Pearson correlation simulation, we initially followed the inside strategy, but the `bundle_sim()` method provides a way to easily move to the outside strategy. Consider the arguments for `Pearson_sim()`:

```
args(Pearson_sim)
```

```
function (reps, N, mu1, mu2, rho = 0, seed = NA_integer_, summarize = TRUE)
NULL
```

The added `summarize` argument allows the user to specify whether to calculate performance for each scenario as the simulation runs, or to return the raw results for each replication. To move outside, simply set `summarize` to `FALSE`:

```
Pearson_sim(reps = 4, N = 15, mu1 = 5, mu2 = 5, rho = 0.5,
            summarize = FALSE)
```

```
          r         z       CI_lo     CI_hi
1 0.4512626 0.4862847 -0.07934109 0.7826127
2 0.4203364 0.4481005 -0.11715191 0.7673676
3 0.6901028 0.8481519  0.27508702 0.8883289
4 0.2223996 0.2261791 -0.32713251 0.6595247
```

By not summarizing as we go, we can save the entire set of estimates, rather than just the performance summaries. This result file will have R times as many rows as the older file. In practice, these result files can grow extremely large, but disk space is cheap.

The following code runs the same experiment as in Section 10.3, but stores the individual replications:

```
sim_results_full <-
  params %>%
  mutate( mu2 = mu1 * lambda ) %>%
  evaluate_by_row( Pearson_sim, reps = 1000, summarize = FALSE )

write_rds( sim_results_full,
           file = "results/Pearson_Poisson_results_full.rds" )
```

The outside method will result in many more rows of data and a much larger file. One small tweak to this workflow is to reduce the file size by keeping the results from each replication in a list-column rather than unnesting them. Here we set `nest_results = TRUE` in the call to `evaluate_by_row()`:

```
sim_results_nested <-
  params %>%
  mutate( mu2 = mu1 * lambda ) %>%
  evaluate_by_row( Pearson_sim, reps = 1000,
                   summarize = FALSE,
                   nest_results = TRUE )

write_rds( sim_results_nested, file = "results/Pearson_Poisson_results_nested.rds" )
```

Here is a comparison of the three approaches:

approach	n_rows	Kb
inside	144	43
outside	144000	9000

approach	n_rows	Kb
nested	144	4528

Storing just the results takes only a few kilobytes, but the raw results take several megabytes, a big difference. If we follow the outside strategy, keeping the results nested reduces the file size by around 50%.

10.4.3 Getting the "outside" results ready for analysis

If we generate raw results using the "outside" approach, we will then need to do the performance calculations across replications within each simulation context. One way to do this is to use `group_by()` and `summarize()` to carry out the performance calculations all at once across the full unnested simulation results:

```
sim_results_full %>%
  group_by( N, mu1, mu2, rho ) %>%
  summarise(
    calc_coverage(lower_bound = CI_lo, upper_bound = CI_hi, true_param = rho)
  )
```

```
# A tibble: 144 x 9
# Groups:   N, mu1, mu2 [18]
       N   mu1   mu2   rho K_coverage coverage coverage_mcse
   <dbl> <dbl> <dbl> <dbl>      <int>    <dbl>         <dbl>
 1    10     4     2   0          1000    0.945       0.00721
 2    10     4     2   0.1        1000    0.936       0.00774
 3    10     4     2   0.2        1000    0.956       0.00649
 4    10     4     2   0.3        1000    0.944       0.00727
 5    10     4     2   0.4        1000    0.946       0.00715
 6    10     4     2   0.5        1000    0.935       0.00780
 7    10     4     2   0.6        1000    0.95        0.00689
 8    10     4     2   0.7        1000    0.945       0.00721
 9    10     4     4   0          1000    0.946       0.00715
10    10     4     4   0.1        1000    0.944       0.00727
# i 134 more rows
# i 2 more variables: width <dbl>, width_mcse <dbl>
```

If we want to use our full performance measure function `evaluate_CIs()` to get additional metrics such as MCSEs, we would *nest* our data into a series of mini-datasets (one for each simulation), and then process each element. As we saw above, nesting collapses a larger dataset into one where one of the variables consists of a list of datasets:

```
results <-
  sim_results_full |>
  group_by( N, mu1, mu2, rho ) %>%
  nest( .key = "res" )
results
```

```
# A tibble: 144 x 5
# Groups:   N, mu1, mu2, rho [144]
       N   mu1   rho   mu2 res
   <dbl> <dbl> <dbl> <dbl> <list>
 1    10     4   0       2 <tibble [1,000 x 4]>
 2    10     4   0.1     2 <tibble [1,000 x 4]>
 3    10     4   0.2     2 <tibble [1,000 x 4]>
 4    10     4   0.3     2 <tibble [1,000 x 4]>
 5    10     4   0.4     2 <tibble [1,000 x 4]>
 6    10     4   0.5     2 <tibble [1,000 x 4]>
 7    10     4   0.6     2 <tibble [1,000 x 4]>
 8    10     4   0.7     2 <tibble [1,000 x 4]>
 9    10     4   0       4 <tibble [1,000 x 4]>
10    10     4   0.1     4 <tibble [1,000 x 4]>
# i 134 more rows
```

Note how each row of our nested data has a little tibble containing the results for that context, with 1000 rows each.[3] Once nested, we can then use `map2()` to apply a function to each element of `res`:

```
results_summary <-
  results %>%
  mutate( performance = map2( res, rho, evaluate_CIs ) ) %>%
  dplyr::select( -res ) %>%
  unnest( cols="performance" )
results_summary
```

```
# A tibble: 144 x 9
# Groups:   N, mu1, mu2, rho [144]
       N   mu1   rho   mu2 K_coverage coverage coverage_mcse
   <dbl> <dbl> <dbl> <dbl>      <int>    <dbl>         <dbl>
 1    10     4   0       2       1000    0.945       0.00721
 2    10     4   0.1     2       1000    0.936       0.00774
 3    10     4   0.2     2       1000    0.956       0.00649
 4    10     4   0.3     2       1000    0.944       0.00727
 5    10     4   0.4     2       1000    0.946       0.00715
 6    10     4   0.5     2       1000    0.935       0.00780
 7    10     4   0.6     2       1000    0.95        0.00689
```

[3] Alternately, we could store the results in nested form (as in `sim_results_nested`), so that the `group_by()` and `nest()` steps are unnecessary.

```
 8    10     4   0.7     2       1000    0.945       0.00721
 9    10     4   0       4       1000    0.946       0.00715
10    10     4   0.1     4       1000    0.944       0.00727
# i 134 more rows
# i 2 more variables: width <dbl>, width_mcse <dbl>
```

We have built our final performance table *after* running the entire simulation, rather than running it on each simulation scenario in turn.

With the "outside" approach, it is easy to add a performance metric without having to recompute the entire simulation: simply change `evaluate_CIs` and recalculate. Summarizing during the simulation vs. after, as we just did, leads to the same set of results.[4] Allowing yourself the flexibility to re-calculate performance measures can be very advantageous, and we tend to follow this outside strategy for any simulations involving more complex estimation procedures.

10.5 Summary

Multifactor simulations are simply a series of individual scenario simulations, where the set of scenarios are structured by systematically manipulating some of the parameters of the data-generating process. The overall workflow for implementing a multifactor simulation begins with identifying which parameters and which specific values of those parameters to explore. These parameters correspond to the factors of the simulation's design; the specific values correspond to the levels (or settings) of each factor. Following the principles of tidy simulation, we represent these decisions as a dataset consisting of all the combinations of the factors that we wish to explore. Think of this as a menu, or checklist, of simulation scenarios to run. The next step in the workflow is then to walk down the list, running a simulation of each scenario in turn.

After executing a multifactor simulation, we will have results from every simulation scenario. These might be the raw results (estimates of quantities of interest from every individual iteration of the simulation) or summary results (performance measures calculated across iterations of the simulation for a given scenario). In either form, the results will be connected to the parameter values (the factor levels) use to generate them. Stacking all the results up will produce a single dataset, suitable for further analysis.

With the workflow that we have demonstrated, it is easy to specify multifactor simulations that involve hundreds or even thousands of distinct scenarios. The amount of data generated by such simulations can quickly grow overwhelming,

[4] In fact, if we use the same seed, we should obtain *exactly* the same results.

and making sense of the results will require further, careful analysis. In the next several chapters, we will examine several strategies for exploring and presenting results from multifactor simulations.

Exercises

Exercise 10.1 (Extending Brown and Forsythe)**.** Brown and Forsythe (1974) evaluated the power of the ANOVA F and Welch test under twenty different conditions, varying in the number of groups, sample sizes, and degree of heteroskedasticity (see Table 5.1).

1. Extend their work by building a multifactor simulation design to compare the power of the tests when $G = 4$, for three or more different sample sizes and for settings with either unbalanced group allocations (e.g., `allocation = c(0.1, 0.2, 0.3, 0.4)`) or equal group sizes.

2. Execute the multifactor simulations using `pmap()` or `evaluate_by_row()`.

3. Create a graph or graphs that depict the power levels of each test as a function of `mu_max`, sample size, and group allocations. How do the power levels of the tests compare to each other overall?

Exercise 10.2 (Comparing the trimmed mean, median and mean)**.** In this extended exercise, you will develop a multifactor simulation to compare several different estimators of a common parameter under a range of scenarios. The specific tasks in this process illustrate how we would approach programming a methodological simulation to compare different estimation strategies, as you might see in the "simulation" section of an article in a statistics journal. In this example, though, both the data-generating process and the estimation strategies are very simple and quick to calculate, so that it is feasible to quickly execute a multifactor simulation. Following tidy simulation principles, the steps described below will walk you through the steps of building and testing functions for each component of the simulation, assembling them into a simulation driver, specifying a simulation design, and executing a multifactor simulation.

The aim of this simulation is to investigate the performance of the mean, trimmed mean, and median as estimators of the center of a symmetric distribution (such that the mean and median parameters are identical).
As the data-generation function, use a scaled t-distribution so that the standard deviation will always be 1 but will have different fatness of tails (high chance of outliers):

```
gen_scaled_t <- function( n, mu, df0 ) {
    mu + rt( n, df=df0 ) / sqrt( df0 / (df0-2) )
}
```

The variance of a t distribution is $df/(df-2)$, so when we divide our observations by the square root of this, we standardize them so they have unit variance. The estimand of interest here is `mu`, the center of the distribution. The estimation methods of interest are the conventional (arithemetic) mean, a 10% trimmed mean, and the median of a sample of n observations. For performance measures, focus on bias, true standard error, and root mean squared error.

1. Verify that `gen_scaled_t()` produces data with mean `mu` and standard deviation 1 for various `df0` values.

2. Write a function to calculate the mean, trimmed mean, and median of a vector of data. The trimmed mean should trim 10% of the data from each end. The method should return a data frame with the three estimates, one row per estimator.

3. Verify your estimation method works by analyzing a dataset generated with `gen_scaled_t()`. For example, you can generate a dataset of size 100 with `gen_scaled_t(100, 0, 3)` and then analyze it.

4. Use `bundle_sim()` to create a simulation function that generates data and then analyzes it. The function should take `n` and `df0` as arguments, and return the estimates from your analysis method. Use `id` to give each simulation run an ID.

5. Run your simulation function for 1000 datasets of size 10, with `mu=0` and `df0=5`. Store the results in a variable called `raw_exps`.

6. Write a function to calculate the RMSE, bias, and standard error for your three estimators, given the results.

7. Make a single function that takes `df0` and `n`, and runs a simulation and returns the performances of your three methods.

8. Now make a grid of $n = 10, 50, 250, 1250$ and $df_0 = 3, 5, 15, 30$, and generate results for your multifactor simulation.

9. Make a plot showing how SE changes as a function of sample size for each estimator. Do the three estimator seem to follow the same pattern? Or do they work differently?

Exercise 10.3 (Estimating latent correlations)**.** Exercise 6.6 introduced a bivariate negative binomial model and asked you to write a data-generating

function that implements the model. Exercise 9.3 provided an estimation function (called `three_corrs`) that calculates three different types of correlation coefficients, and asked you to write a function for calculating the bias and RMSE of these measures.

1. Combine your data-generating function, `three_corrs()`, and your performance calculation into a simulation driver.

2. Propose a multifactor simulation design to examine the bias and RMSE of these three correlations. Write code to create a parameter grid for your proposed simulations.

3. Execute the simulations for your proposed design.

4. Create a graph or graphs that depict the bias and RMSE of each correlation as a function of ρ and any other key parameters.

Exercise 10.4 (Examine a multifactor simulation design)**.** Find a published article that reports a multifactor simulation study examining a methodological question.[5] Write code to create a parameter grid for the scenarios examined in the study. Write a few sentences explaining the overall design of the simulation study. Summarize any justification that the authors provided for the choice of parameter values examined.

[5]Some journals that regularly publish methodological simulation studies include Psychological Methods, Psychometrika, Journal of Educational and Behavioral Statistics, Multivariate Behavioral Research, Behavior Research Methods, Research Synthesis Methods, and Statistics in Medicine.

11

Designing multifactor simulations

Multi-factor simulations have the potential to provide powerful evidence for how methods perform across a range of conditions. We can use them to deeply understand when a method will shine, and when it will fail. The multi-factor aspect of a simulation can take us from an overly narrow exploration to one that has broader significance. As Little (2013) puts it:

> Good simulation studies are not given the respect they deserve. Often the design is perfunctory and simplistic, neglecting to attempt a factorial experimental design to cover the relevant sample space, and results are over-generalized. Well designed simulation studies with realistic sample sizes are an antidote to a fixation on asymptotics and a useful tool for assessing calibration.

In Section 10.2, we demonstrated how to represent a full factorial design as a list of the factors and levels to be explored. We built out a multifactor simulation study to examine the performance of confidence intervals for Pearson's correlation coefficient under a bivariate Poisson distribution. We arrived at a $3 \times 3 \times 2 \times 8$ factorial design, varying the sample size across three levels, the mean of one variable across three levels, the ratio of means across two levels, and the true correlation between variables across eight levels.

But how do we go about choosing which 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 is difficult to extrapolate findings from a simulation study beyond the set of simulation conditions that were examined. On the other hand, it is often difficult or impossible to examine the full space of all possible parameter values, except for very simple problems. Even in the relatively straightforward Pearson correlation simulation, we have four factors and the last three could each take on an infinite number of possible levels. How do we come up with a defensible set of levels to examine?

11.1 Choosing factors and parameter combinations

We have no generic algorithm for choosing which parameters to vary in a simulation design, nor for choosing what levels to use for each parameter. The particular answers to these questions always depend on the problem being studied, the focus of the research questions, and the availability of any relevant theoretical results. So instead of specific, acontextual rules, we offer broad principles and practical advice informed by our experience conducting and reading simulation studies.

Our primary principle is to vary parameters that we believe matter or that we think other people will believe matter. The first is so we can learn for ourselves. The second is to build our case. Breaking this down a bit, we find the following six precepts useful:

1. **Include the obvious or known**. Sometimes theory, general knowledge, or prior research will tell you how well a method will perform under some circumstance. For example, it might be mathematically tractable to derive the performance of an estimator if all the groups are the same size and everything is normally distributed. As a starting point, include such cases as part of your exploration to reinforce and validate intuition, use as a point of reference for the other results, and demonstrate that your implementation is working as expected.
2. **Adhere to your research questions**. The parameters you select should be those that are most relevant to the research questions you are trying to answer. For example, if you are interested in how well a method performs when there are outliers, then you need to think carefully about how you are going to vary the presence of outliers in your data. By keeping the research questions in mind, it is easier to navigate chocies and trade-offs between things you might vary.
3. **Relate to existing research**. Especially for research simulations, it is important to be able to relate your findings to previous research. This suggests that you should select parameter levels that make this possible, such as by selecting sample sizes similar to those examined in previous studies. That said, previous simulation studies are not always perfect (in truth, there are a lot of quite bad ones out there!), and so prior work should not be your sole guide or justification.
4. **Be comprehensive**. Generally, 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 later on.

5. **Push boundaries**. One of the most useful potential contributions of research simulations is to explore the breakdown points of a method (e.g., what sample size is too small for an estimator to be trusted?) 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.

6. **Explore the incidental**. Even if you do not expect a particular parameter to matter, it can be useful to include it in your simulation design as a sanity check. In particular, in addition to any primary parameters of interest, include at least a few nuisance parameters to test the sensitivity of your results to these other aspects.

11.1.1 Deciding on the factors

Write down all the factors you can think of, and then sort through them to identify which are the most important. In particular, look at the core research questions, and try to identify which factors would give evidence for those questions. For many of the factors, you may eventually select a single level, if you do not think the factor matters for the questions you are considering.

Often, your DGP will be a parametric model. In principle, each parameter could then be taken as a factor, but this can create some difficulties. In particular, the factors represented by the parameters can feel quite interdependent in terms of the effects they have on the resulting data. For example, if you had a linear model for your DGP, with $Y = a + bX + \epsilon$, with $\epsilon \sim N(0, \sigma^2)$, then if you identify b as a factor, increasing it would both increase the R^2 of X and also increase the variation of Y. Instead we might want to separately manipulate the R^2 and the variance of Y by using a different parameterization of the DGP, such as by using the R^2 and the variance of Y as initial parameters that we then translate to values of b and σ^2. See Section 6.6.4.

More straightforward as factors are the design parameters, e.g., sample sizes. You should still list these, and think about why you would want to vary them. For example, sample size is often varied because the simulation is partly for identifying when an asymptotic behavior kicks in, and for understanding what happens before that point.

If possible, write the data generation code with all the factors as arguments, so that you can easily vary them in your simulation design if you choose to later on. That said, it is also helpful to specify default values for factors, so you can ignore them if you decide they are not important enough to vary.

You will end up with too many factors to vary, so you will then need to select a subset of them to include in your simulation design. The entire simulation

process is iterative: pick the factors you are most confident about, and know as you accumulate simulation evidence and thus learn more about what is going on, you will likely be adjusting your set of factors or running other simulations to verify your developing understanding.

11.1.2 Deciding on the range and levels of a factor

Once you have identified your factors, you then have to decide on the range and the specific levels of the factor you will include in the simulation. Determine these details by first determining the range, and then determining the number of levels (and specific values) within the range.

For the range, there are at least three strategies you might take:

1. Vary a factor over as much of its range as you can.
2. Choose factor levels to represent a realistic practical range. These ranges would ideally be empirically justified based on systematic reviews of prior applications. Lacking such empirical evidence, you may need to rely on informal impressions of what is realistic in practice.
3. Tune the factors to emulate an important application or context.

Of these choices, option (1) is the most general but also the most computationally intensive. In the ideal, option (2) focuses attention on what is most relevant to a practitioner. Option (3) is usually coupled with a subsequent applied data analysis, and in this case the simulation is often used to enrich that analysis. In particular, 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.

For determining the number of levels and specific values to use, there are also a few strategies you might take. Begin with a default of at least three levels of each factor going from the minimum to maximum of your range. These three levels encode something notably small, something about what we would expect to see in the real world, and something notably high. From these levels you can get a good sense of what happens both when things are "normal" and also when you push the boundaries.

Three levels is also the smallest number that gives some sense of how much variation there is when looking at raw results. For example, say you had three SE estimates, that, as a factor increases, were 3.1, 5.2, and 2.6. The up and down nature would either be something very surprising, or an indication that the Monte Carlo uncertainty was large enough to create notable "wiggles" in your results. An advantage of more levels of a factor is that it makes Monte Carlo uncertainty more salient.

An additional reason for using at least three levels is that doing so can give a sense of whether trends in a factor exhibit curvature. This can be useful for determining, for example, if the relationship between a factor and performance is linear, quadratic, or leveling off. A two-level factor will always give an impression of increase or decrease, without any sense of shape or even whether the increase is due to Monte Carlo uncertainty.

By contrast, having only a few levels can often make analyzing simulations easier. In particular, experiments with just "high" and "low" conditions are often easier to analyze and explain. Even more straightforward is when a factor is "on" or "off"—this allows straightforward statements such as "when there is a correlation, the estimator is biased," and so forth.

Many factors fit neatly into an "on" and "off" framing. For example, the error distribution is normal or it is skewed. There are outliers or there are not. For these factors, a binary "on/off" design is often sufficient, and more levels may not be necessary. Be sure the "on" level, in this case, is *notably* on: you want to have a large change so you can clearly see the impact of the factor on your performance criteria. That said, we would also want to know if a risk is a realistic risk in practice. Some estimators may be prone to bias, but if the bias is not practically large, then we might not mind. For example, using regression in randomized trials can cause bias (Freedman, 2008), but in practice that bias is small (Lin, 2013). If this is a concern, then considering three levels might be reasonable: "off," "realistic," and "high."

For truly continuous factors of great importance, having many values along the range can make it possible to fit a curve to fully describe how performance changes as the factor does. A dense sequence of levels is frequently used for power analyses, for example: slowly raise the size of the effect and see how power changes to trace out a full power curve.

11.2 Case Study: A multifactor evaluation of cluster RCT estimators

To bring the design of multifactor simulations to life, we return to the case study of comparing three ways to analyze a cluster randomized trial presented in Section 6.6. In our original setup, we wrote code to generate cluster randomized trial data where a cluster's size could be correlated its average treatment effect. In Section 9.1.1, we looked across a variety of performance criteria so we could see how the estimators compared for any given scenario we wanted.

So far, we have only examined a single scenario at a time. But how do our findings generalize? Under what conditions do the various estimation methods perform better or worse? To answer these questions, we need to extend to

a multifactor simulation to *systematically* explore how our three estimators behave across a range of contexts. Happily, the modular functions that we have designed make it relatively straightforward to explore a range of scenarios by calling our simulation function over and over, using the tools of this chapter. We now illustrate how to extend this simulation to a full multifactor design. Then in Chapter 13, we will continue this running example by examining how to analyze the results of the multifactor design.

11.2.1 Choosing parameters for the Clustered RCT

We begin by identifying some potential research questions suggested by our preliminary exploration:

- Regarding bias, we noticed in our initial simulation that Linear Regression targets a person-weighted average effect, so it would be considered biased for the cluster-average treatment effect. But how large is this bias in practice, and how much does bias change as we change the association between cluster-size and impact?
- Considering precision, we saw that Linear Regression has a higher standard error than the other estimators. Is this a general pattern? If not, are there contexts where linear regression will have a lower standard error than the others?
- We originally thought that aggregation would lose information because smaller clusters would have the same weight as larger clusters, but be more imprecisely estimated. This did not seem to happen. Were we wrong? Or perhaps we would see it if cluster size were even more variable?
- The estimated SEs for all three methods all appeared to be good, although they were rather variable, relative to the true SE. Is this a general pattern? Are the standard errors always the right size, on average? Or will the estimated SEs fall apart (i.e., be far too large or far too small) in some contexts? If so, which ones? Do cluster-robust SEs work with fewer clusters?

To answer all of these questions we need to systematically explore the space of data-generating conditions. Our data-generating function has many knobs to turn. In particular, we can manipulate all of the following features:

- the number of clusters (J)
- the proportion of clusters that receive treatment (p)
- the average cluster size ($\bar{n}$)
- the degree of variation in cluster sizes (α)
- the average outcome in the control group (γ_0)
- the average treatment effect (γ_1)
- how strongly average impact is connected to cluster size (γ_2)
- how much the cluster intercepts vary (the degree of cross-cluster variation) (σ_u^2)

- the degree of residual variation (σ^2_ϵ)

Manipulating all of these factors would lead to a huge and unwieldy number of simulation conditions to evaluate. To cut down our list, we turn to our six principles laid out at the start of the chapter.

First, we want to include the obvious and known. Here, a natural obvious baseline would be clusters of equal size with no impact variation, so we should definely generate data with $\alpha = 0$ and $\gamma_2 = 0$. We probably want to also have some scenarios with $\sigma^2_u = 0$ as well.

Next, we turn to our research questions and then speculate as to what factor would help us answer each one:

- Bias: This is about variation in cluster size, and how strongly it is connected to cluster-level impact. We know, then, that we want to manipulate both α and γ_2. Sticking with our first principle, we want to also verify that there is not bias if there is no correlation. We might here notice that we cannot have bias if there is no variation in cluster size; more on what to do about this later.
- Precision: When would linear regression have lower standard error? We might not know, but guess it could happen if clusters were very different from one another. This might motivate manipulating σ^2_u and, again, α.
- Aggregation: These questions motivate ensuring that α goes from 0 to very large.
- SE: We know SEs are usually asymptotic in nature, so we should explore when J and $\bar{n}$ are each small, and both small.

When selecting factors to manipulate, it is important to ensure the each factor is isolated, so that changing one of them should not change other aspects of the data-generating process that might impact performance. In this case, for example, if we simply added more cross-cluster variation by directly increasing the random effects for the clusters, the total variation in the outcome will also increase. If we then see that the performance of an estimator deteriorates as variation increases, we have a confound: is the cross-cluster variation causing the problem, or is it the total variation? To avoid this confound, we should vary cluster variation while holding the total variation fixed; this is why we use the ICC parameterization, as discussed in Section 6.6, rather than directly manipulating σ^2_ϵ and σ^2_u.

We notice none of our questions involve the average outcome (γ_0) or the ATE (γ_1). Based on our theoretical understanding of the problem, we are fairly confident the intercept and ATE will not have any effect on performance, as they are simply constant shifts. We thus decide to set $\gamma_0 = 0$ (because we are working with a standardized outcome) and $\gamma_1 = 0.20$, a typical value seen in education. We could later expand these if we wanted to.

For proportion treated, again our research questions do not seem to relate

to it. We thus select $p = 0.50$. This is a common choice, but is not actually a good choice. When in doubt, it is better to create imbalance in the DGP, as it is easy for things to cancel out when everything is symmetrical. For example, $p = 1/3$ might have been more likely to uncover surprising findings. Nevertheless, we cannot explore everything.

For ranges of our other factors, we look at existing research and existing empirical studies in education. For sample sizes, we know clusters can be near 20 students to hundreds in size, and we know experiments can be very small (e.g., 5) to moderately large (e.g., 80). For the ICC, we explore none to extremely high levels. We could have focused on more realistic range (e.g., 0 to 0.4), but we decided to see what happens to the estimators in the extremes to help understand the dynamics at play. For cluster variation, we go with no variation, moderate variation, and high variation. We can only take α to at most 1; at 1 the smallest cluster would be size 0. At 0.8, the smallest cluster would be 20% the size of the average cluster; we could do further exploration of the literature to understand variation better to calibrate this term.

Our reasoning above gives our final set of values:

```
crt_design_factors <- list(
  n_bar = c( 20, 80, 320 ),
  p = c( 0.5 ),
  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 )
)
```

We have $3 \times 3 \times 2 \times 5 \times 3 = 270$ total scenarios.

Overall, we have selected factors and levels that satisfy most of the principles we have articulated. We have included the obvious (the no variation cases), we have focused on factors relevant to our questions, we have selected values similar to those seen in previous studies, and we tried for a wide range of values for each factor— even including some extreme values (e.g., ICC up to 0.8 or α of 0.8). We have deeply not explored the incidental (here p and γ_1, in particular), but we could do this later as a sanity check on our results. This is also our initial pass. We may find that our estimators only start to breakdown at, say, alpha = 0.8. We might then go back and expand the range to 0.9 or 0.99 to explore that edge further. Or we may find that we should try $\bar{n} = 4$ or even smaller, to explore small-sample properties more thoroughly.

11.2.2 Redundant factor combinations

There is some redundancy in the parameter combinations that we have selected. In particular, if `size_coef` is nonzero, but `alpha` is 0, then the cluster size will not impact the cluster average ATE because there is no variation in cluster size. We could drop one of the redundant conditions to save some simulation runs—in essence we are running the same simulation for `alpha=0` two times, once for each `size_coef` value, for each level of `ICC`, `n_bar` and `J`. However, doing so would mean that our simulation is no longer a full factorial (because the factors are not fully crossed), making it harder to analyze. With an imbalanced experiment, if we average performance across some factors to see the effect of others, we might end up with surprising or even misleading results.

Because computation is cheap, we will leave the redundant conditions in as a sanity check for our results. We know we should get the same results; if we do not, then we either have uncontrolled uncertainty in our simulation or an error in the code.

11.2.3 Running the simulations

We will run our cluster RCT simulation using the same code pattern as we used with the Pearson correlation simulations. Because we are not exactly sure which performance metrics we will want to use, we will save the individual replications and then calculate performance metrics after generating the simulation results. That is, we will use the "outside" strategy.

We first make a table of scenarios with unique seeds for each:

```
params <-
  expand_grid( !!!crt_design_factors ) %>%
  mutate(
    seed = 20200320 + 17 * 1:n()
  )
```

The different seeds for each scenario avoid anything confusing about shared randomness across scenarios (see Section 8.5 for further discussion). We then run the simulation 1000 times per scenario, then unnest to get our final data:

```
params$res <- pmap(params, .f = run_CRT_sim, reps = 1000 )
res <- unnest(params, cols=data)
saveRDS( res, file = "results/simulation_CRT.rds" )
```

Normally, we would speed up these calculations using parallel processing; we discuss how to do so in Chapter 18. Even under parallel processing, this simulation took several hours to run. Our final results look like this:

```
Rows: 810,000
Columns: 16
```

```
$ n_bar     <dbl> 320, 320, 320, 320, 320, 320, 320, 320, 320,~
$ J         <dbl> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,~
$ ATE       <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2,~
$ size_coef <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2,~
$ ICC       <dbl> 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8, 0.8,~
$ alpha     <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5,~
$ seed      <dbl> 103553, 103553, 103553, 103553, 103553, 1035~
$ runID     <int> 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5,~
$ method    <chr> "MLM", "LR", "Agg", "MLM", "LR", "Agg", "MLM~
$ ATE_hat   <dbl> -0.11701641, -0.27662276, -0.11673502, -0.63~
$ SE_hat    <dbl> 0.5485042, 0.5980588, 0.6178521, 0.4906032, ~
$ p_value   <dbl> 0.84473987, 0.68825974, 0.86220164, 0.286537~
$ message   <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
$ warning   <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
$ error     <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
$ seconds   <dbl> 0.176337957, 0.086425066, 0.141067028, 0.025~
```

We have a lot of rows of data!

Our next step is to group the results by the simulation factors and calculate the performance metrics across the replications of each simulation condition. Here we calculate our primary performance measures by hand:

```
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"
  )
```

If we want MCSEs (as we usually do), then we could do that by hand or use the `simhelpers` package as so:

```
library( simhelpers )

sres <-
  res %>%
  group_by( n_bar, J, ATE, size_coef, ICC, alpha, method ) %>%
```

```
  summarise(
    calc_absolute(
      estimates = ATE_hat, true_param = ATE,
      criteria = c("bias","stddev","rmse")
    ),
    calc_relative_var(
      estimates = ATE_hat, var_estimates = SE_hat^2,
      criteria = "relative bias"
    )
  ) %>%
  rename( SE = stddev, SE_mcse = stddev_mcse ) %>%
  dplyr::select( -K_absolute, -K_relvar ) %>%
  ungroup()

glimpse( sres )
```

```
Rows: 810
Columns: 15
$ n_bar             <dbl> 20, 20, 20, 20, 20, 20, 20, 20, 20, ~
$ J                 <dbl> 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, ~
$ ATE               <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0~
$ size_coef         <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ ICC               <dbl> 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0~
$ alpha             <dbl> 0.0, 0.0, 0.0, 0.5, 0.5, 0.5, 0.8, 0~
$ method            <chr> "Agg", "LR", "MLM", "Agg", "LR", "ML~
$ bias              <dbl> 0.0078528924, 0.0078528924, 0.007852~
$ bias_mcse         <dbl> 0.006512388, 0.006512388, 0.00651238~
$ SE                <dbl> 0.2059398, 0.2059398, 0.2059398, 0.2~
$ SE_mcse           <dbl> 0.004380492, 0.004380492, 0.00438049~
$ rmse              <dbl> 0.2059865, 0.2059865, 0.2059865, 0.2~
$ rmse_mcse         <dbl> 0.005468829, 0.005468829, 0.00546882~
$ rel_bias_var      <dbl> 0.9834635, 0.9834635, 1.2690179, 1.0~
$ rel_bias_var_mcse <dbl> 0.05290317, 0.05290317, 0.05794197, ~
```

After we calculate our performance measures, we have 810 rows of results across 270 different scenarios. We can simply aggregate across everything to get overall comparisons:

```
sres %>%
  group_by( method ) %>%
  summarise( mean_bias = mean( bias ),
             mean_SE = mean( SE ),
             mean_rmse = mean( rmse ) )
```

```
# A tibble: 3 x 4
  method mean_bias mean_SE mean_rmse
```

```
  <chr>       <dbl>   <dbl>     <dbl>
1 Agg     0.000783   0.311     0.311
2 LR      0.00863    0.320     0.321
3 MLM     0.00227    0.310     0.310
```

Even with this very crude summary of the results, we see tantalizing hints of differences between the methods. Linear Regression does seem biased, on average, across the scenarios considered. But so does multilevel modeling—we did not see bias before, but apparently for some scenarios it is biased as well. Overall SE and RMSE seems roughly the same across scenarios, and it is clear that uncertainty trumps bias, at least on average. However, this pattern could be driven by the scenarios with few clusters that are small in size. In order to understand the full story of what is going on, we need to dig further into the results.

11.3 Further principles of design

11.3.1 The number of factors vs. number of replicates trade-off

For any factor, more levels is usually better, but more levels is also more computationally intensive. Even modest increases in the number of factors for each level can create an explosion in the total number of simulation trials. Consider that a four-factor experiment with 2 levels each is only 16 conditions; with 3 levels each, it is 81 conditions; and with 4 levels each, it is 256 conditions. This exponential explosion is hard to navigate: more comprehensive designs require massively more computing. However, the computational cost can be tempered by reducing the number of replications per scenario.

For example, suppose you were initially planning on doing 1000 simulations per scenario, but then you realize that you need to incorporate an additional factor. After thinking about it, you suspect that this factor will not actually matter but you worry that other researchers might find it important. To address this potential concern, you could include the factor in your simulation design, say with four levels, but then reduce the number of replications to 250 simulations per scenario. The total work remains the same.

When analyzing the final simulation you would first verify you do not see trends along this new factor, and then marginalize out that factor (i.e., average your performance metrics across the factor) in your summaries of results. Marginalizing out a 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. We will discuss marginalization further in Chapter 12.

11.3.2 Preliminary explorations and multiple simulations

For factors you do not expect to matter, you can conduct preliminary simulations to verify they do not matter, and then drop them from consideration.

For example, in our cluster RCT example, we do not expect the size of the treatment effect to impact performance. Even so, we might still run an initial simulation with multiple levels of the treatment effect to verify our intuition is correct. If nothing changes, then we proceed with our single value for the ATE.

On the other hand, if we did find that estimator performances changed as we changed the treatment effect, then we would have a sign that there is some uncontrolled aspect of the simulation that was impacting the results, or that we had an error in our code. In either case, this would be an important finding that would need to be addressed before we could have confidence in the full simulation results.

One might also run distinct multi-factor simulations as well. For example, in the Cluster RCT study we might run a seperate multi-factor simulation where we vary p (proportion treated) over multiple levels, but keep $ICC = 0.20$ and $\alpha = 0.5$ both fixed. This second simulation would allow for evaluating how proportion treated matters in its own constillation of other factors.

To sharpen this point, consider six factors of 3 levels each. A full factorial experiment would be $3^6 = 729$ conditions. Three different simulations of different sets of four of the six factors would be $3^4 + 3^4 + 3^4 = 243$ conditions. At the extreme, some will vary one factor at a time, which would be $3 \times 6 = 18$ conditions. In this latter case, we could easily expand the resolution of each factor to many different levels.

11.3.3 Selecting number of replications

The number of replications per scenario R is a key design choice. For some performance measures, there are good initial rules of thumb.

For simulations focused on confidence interval coverage, rules of thumb can suffice. If the true coverage level is π, then the MCSE of coverage levels will be $\sqrt{\pi(1-\pi)/R}$, and the margin of error will be $\Phi^{-1}(0.975) \times \sqrt{\pi(1-\pi)/R}$. Solving this expression for R lets us determine the number of replications needed to attain a given margin of error. Thus, if coverage is expected to be around 0.95 and we want a Monte Carlo margin of error of approximately 0.01, we would need about $R = 2000$ replications. Using only $R = 500$ replications would yield a margin of error of approximately 0.02.

For simulations focused on estimator bias or RMSE, it is somewhat more difficult to find general rules of thumb. If σ is the true standard error of an estimator in a given condition, then the MCSE of bias will be $\sigma/\sqrt{R}$ and the

margin of error will be $\Phi^{-1}(0.975)\sigma/\sqrt{R}$. In practice, a common but arbitrary choice is to use $R = 1000$ replications per scenario, which yields a margin of error of around 0.06σ. Whether this is sufficiently small Monte Carlo error will depend on the context of the simulation. If the simulation involves examining several estimators to understand differences in their bias, then a margin of error of 0.06σ might be too large to identify systematic differences between the estimators.

In practice, published simulation studies typically use the same number of replications for every scenario examined. However, there are sometimes good reasons to deviate from this convention and use different R for different scenarios. One reason to do so is that some scenarios may be more computationally intensive than others. For instance, if the computation time increases with sample size, then it might be worth using fewer replications for the conditions with larger sample size.

Another reason to consider varying R is that Monte Carlo uncertainty itself can be strongly dependent on model parameters. Again consider a simulation focused on estimator bias, for which the MCSE will be $\sigma/\sqrt{R}$. If σ depends on any of the simulation factors, then the MCSE will vary across scenarios. In many instances, σ is roughly proportional to $N^{-1/2}$, where N is the sample size used for a given replication. Thus, σ will be large when N is small and σ will be smaller for scenarios where N is larger. Using a constant R for every scenario will mean that the MCSE will be smaller for large-N scenarios than for small-N scenarios. Alternately, we could set $R = R_0/N$, where R_0 is some base factor, so that the MCSE of bias will be roughly constant across scenarios with different sample sizes. The simulations reported in Pustejovsky (2014) used this strategy, setting $R = 10^7/N$ for simulations that examined sample sizes of $N = 20, 40, 80$, and 160.

11.4 Summary

In this chapter, we have moved from isolated simulation scenarios to fully specified multifactor simulation designs. Full factorial designs involve systematically varying multiple features of a data-generating process, which allows us to explore how the performance of an estimator depends on the context in which it is used. By thinking of simulations as designed experiments, we gain a principled framework for choosing parameters and curating a set of scenarios to explore. The result is a rich collection of simulation output that can capture bias, variability, uncertainty estimation, or testing behavior across a wide range of plausible conditions.

As we noted at the outset, the primary limitation of a simulation study is gen-

eralizability. Multifactor simulations are often designed to address this core concern. Of course, no simulation is entirely generalizable—even an extensive full factorial design will entail many stylized assumptions and artificial constraints. To withstand critiques about generalizability, simulations should be designed by aiming for comprehensiveness, by including conditions that are relevant to real data, and by including conditions that are relevant to previous methodological work. Although the result will surely fall short of perfect generalizability, we can still design simulations thoughtfully and as best we can, seeking to avoid obvious limitations that a critic might identify as a basis for dismissing our findings.

In our experience, it is quite rare to nail the choice of parameters on the first pass. You should expect to add and subtract from your identified set of simulation factors with each round of results. Of course, the purpose of revision should never be to make a particular estimator look better or to conceal problems with the performance of a method. Rather, it should be to identify the set of factors and parameter settings that are most relevant and most useful for investigating the questions at issue. After all, the final design that we arrive at needs to be well justified and defensible.

Well designed simulations can be quite large and complex, however, which creates a challenge for interpretation. With many factors, levels, and performance measures, it is no longer possible to understand results simply by inspecting a table of results. In the next few chapters, we will turn to the problem of analyzing and presenting simulation results, with an emphasis on graphical approaches that clarify how performance varies across conditions. Through further analysis, we will seek to identify systematic trends in estimator performance, understand the generality of identified patterns, and identify trade-offs between different methods.

Exercises

Exercise 11.1 (Meta-regression)**.** Exercise 6.14 described the random effects meta-regression model. List the focal, auxiliary, and structural parameters of this model, and propose a set of design factors to use in a multifactor simulation of the model. Create a list with one entry per factor, then create a dataset with one row for each simulation context that you propose to evaluate.

12

Exploring and presenting simulation results

Once we have performance measures for each method examined for each scenario of our study's design, the computationally challenging parts of a simulation study are complete, but several intellectually challenging tasks remain. The goal of a simulation study is to provide evidence to address a research question or questions, but performance measures (like numbers more generally) do not analyze themselves. Rather, they require interpretation, analysis, and communication in order to identify findings—that is, broad, potentially generalizable patterns of results—that are responsive the research questions. Good analysis aims to provide a clear understanding of how one or more of the simulation factors influence key performance measures of interest, the circumstances where a data analysis method works well or breaks down, and—in simulations that examine multiple methods—the conditions where one method performs better or worse than the alternatives.

In multi-factor simulations, the major challenge in analyzing simulation results is dealing with the multiplicity and dimensional nature of the results. For instance, in our cluster RCT simulation, we examined 270 different simulation scenarios, which vary along several factors. For each scenario, we calculated a whole suite of performance measures (bias, SE, RMSE, coverage, …), and we have these performance measures for each of the three estimation methods under consideration. We organized all these results as a data table with 810 rows (three rows per simulation scenario, with each row corresponding to a specific method) and one column per performance metric.

Navigating all of this can feel somewhat overwhelming. How do we understand trends in this complex, multi-factor data structure? Broadly, the answer is to think of the performance measures as outcomes of a designed experiment, so that we can bring to bear any and all of the data analysis techniques relevant to analyzing experimental designs involving real data.

In this chapter, we survey three main categories of analytic tools that can be used for exploring and presenting simulation results:

1. Tabulation

2. Visualization
3. Modeling

For each category of tools, we describe the logic behind how it can be applied, provide high-level examples drawn from the literature and our own work, and discuss its strengths and limitations.

In this and subsequent chapters, we assume that you will be sharing the findings from your simulations with an audience beyond yourself. Depending on your circumstances, that might be a broad audience of researchers and data analysts, with whom you will communicate through a scholarly article in a peer-reviewed methodology journal; it might be colleagues who are evaluating your proposal for an empirical study, where the simulations serve to justify the data analysis protocol; it might be a small group of collaborators, who will use the simulations to make decisions about how to design a study; or it might be fellow students of statistics, interested to read a blog post that discusses how a particular model or method works. These contexts differ in the format and level of formality used to communicate findings. Still, in any of these contexts, you will probably need to create a written explanation of your findings, which summarizes what you found and presents evidence to support your assertions and interpretations. We close this chapter with a discussion about creating such write-ups and distilling your analysis into a set of exhibits for presentation.

12.1 Tabulation

Simulation study results are often presented in tables. A table might be fine if it involves only a few numbers and a few targeted comparisons or if it is important to report *exact* values for some quantities. However, simulations usually produce lots of numbers and require making many comparisons. For example, you might need to show the relative performance of several alternative estimators or the performance of your estimators under different conditions for the data-generating model. This usually 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, neither you nor your audience will be particularly well served. Furthermore, in simulation, exact values of performance measures are not usually of interest—in fact, we rarely have them due to Monte Carlo simulation error. Because of this, tables risk providing a false sense of uncertainty—unless you also report MCSEs, which would create further clutter. In general, we believe tables rarely make the take-aways of a simulation readily apparent. As a rule of thumb, if you are ever tempted into putting your table in landscape mode to get it to fit on the page, think again. It is usually more useful and insightful

to present results in graphs (Gelman et al., 2002).

To illustrate, consider the following table of simulation results showing the false rejection rate, against an α of 0.10, for an estimator of an average treatment impact. We have two factors of interest, the treatment and control group sizes.

Table 12.1: Rejection rates for a two-sample t-test assuming unequal variances

nT	nC	reject
2	2	19
2	4	16
2	10	20
2	50	26
2	500	31
10	2	18
10	4	9
10	10	7
10	50	7
10	500	8
500	2	29
500	4	14
500	10	8
500	50	5
500	500	5

This is the classic table you might see in a paper: the table is structured as groups of results indexed by one factor, with a second factor that varies within each group. The table makes clear that the rejection rates are often well above 10% and that the rates are all far too high when there are few treatment units. Because of the ordering of rows, it is somewhat more difficult to see how the number of control units impacts the rate, and understanding the joint relationship between number of treatment and number of control requires extra thinking.

By contrast, a plot of these exact same numbers can make trends much clearer. Figure 12.1 is an "interaction plot" showing the "interaction" of the factors nT and nC. This plot makes apparent that we only achieve something close to a valid test if both nC and nT are above 50. Even if nC is 500, the rejection rate is elevated if nT is only 10. When nT is 2, then increasing nC actually *increases* the rejection rate, meaning that larger samples make things worse. None of these trends were obvious from looking at the raw table results.

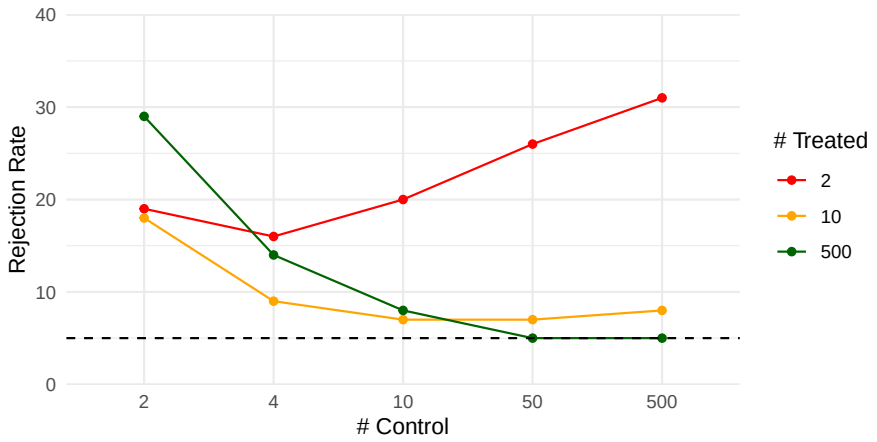


Figure 12.1: Rejection rates for a two-sample t-test assuming unequal variances.

12.1.1 Estimators of treatment variation

Tables do serve some purposes. In general, tables are more useful for displaying high-level summaries of findings, such as average performance across a range of simulated scenarios, rather than for reporting raw results. For example, in ongoing work, Miratrix has been studying the performance of a suite of estimators designed to estimate individual treatment effects. To test which estimators perform better or worse than the others, we designed a series of scenarios where we varied the data-generating model by a variety of factors. For each scenario, we calculated the performance of each estimator relative to the median of all estimators considered. We can then ask, do some methods perform better than their peers on average across all scenarios considered?

Table 12.2: Relative performance measures of methods for estimating individual treatment effects

model	bias	se	rmse	sd_bias	sd_se	sd_rmse	R2	sd_R2
BART S	-6	-28	-17	13	14	12	0.40	0.25
CF	-10	-2	-10	14	34	11	0.35	0.19
CF LC	-10	-1	-10	14	33	10	0.34	0.19
LASSO R	2	-21	-10	9	13	10	0.31	0.25
LASSO MCM EA	2	-20	-10	9	13	10	0.31	0.25
LASSO MOM DR	2	-21	-9	9	13	10	0.31	0.25
RF MOM DR	3	-12	-8	15	16	8	0.31	0.20
LASSO T	-1	-10	-6	16	23	9	0.32	0.23
LASSO T INT	-4	19	5	16	55	17	0.28	0.20

Table 12.2: Relative performance measures of methods for estimating individual treatment effects

model	bias	se	rmse	sd_bias	sd_se	sd_rmse	R2	sd_R2
LASSO MCM	20	4	8	20	10	8	0.14	0.15
LASSO MOM IPW	20	4	8	20	10	8	0.14	0.15
ATE	47	-60	9	48	7	17	NA	NA
RF T	-22	60	11	17	48	19	0.29	0.13
RF MOM IPW	15	27	12	31	20	13	0.16	0.12
OLS S	-26	87	23	28	83	34	0.31	0.22
BART T	-38	103	25	19	54	22	0.32	0.18
CDML	9	71	31	20	99	46	0.24	0.20

Table 12.2 gives an answer to this question: we evaluate each method along four metrics: relative bias, relative se, relative rmse, and R^2. To easily see which is good and which is bad, we averaged the performance measures across all scenarios and order the methods from highest average relative RMSE to lowest. Each cell of the table summarizes across 324 scenarios. The first several columns show relative performance. To calculate these values, for each method m, performance metric Q, and scenario s, we calculated $P_{ms} = Q_{ms}/median(Q_{ms})$, and then averaged the P_{ms} across the scenarios to get $\bar{P}_m$. Each method also has an R^2_{ms} value for each scenario; we simply take the average of these across all scenarios for the penultimate column. The standard deviation columns show the variation in performance across the full set of scenarios, providing some indication of how much the relative performance of a method changes from scenario to scenario. Representing this variation more explicitly might be better done with a visualization, as we explore shortly.

Overall, the table provides a nice, high-level summary of the results. It reports no less than four performance measures, one of which (the R^2) is on a different scale than the others; this is hard to do in a single visualiztion. Despite these strengths, we still do not feel that the tables makes the results particularly visceral. A visualization can make trends jump out much more clearly.

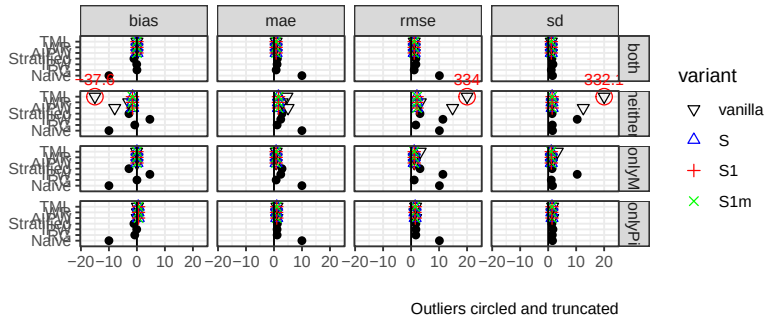
12.1.2 Missing data and causal inference

For another example, Zhou et al. (2015) examined the performance of a variety of methods for estimating causal effects in the presence of missing data. They present their simulation results in a series of tables. Their Table 2, for example, had four quadrants, each a distinct scenario. For example, the results for the scenario with misspecified outcome models and propensity score models looked like this:

method	bias	sd	rmse	mae
Naive	-10.00	1.56	10.12	9.97
RG	-0.77	1.50	1.69	1.12
IPW	4.63	10.38	11.37	2.41
Stratified	-2.85	1.41	3.18	2.87
AIPW	-7.93	12.55	14.84	5.02
AIPW-S	-1.53	1.42	2.08	1.59
AIPW-S1	-1.58	1.42	2.12	1.61
AIPW-S1m	-1.59	1.42	2.13	1.63
WR	-2.93	1.48	3.29	2.91
WR-S	-1.53	1.36	2.04	1.56
WR-S1	-1.58	1.36	2.09	1.59
WR-S1m	-1.59	1.36	2.09	1.61
TML	-37.60	332.08	334.03	4.62
TML-S	-1.53	1.42	2.08	1.59
TML-S1	-1.58	1.42	2.13	1.61
TML-S1m	-1.58	1.42	2.13	1.62

Reading the table involves scanning across many methods, some of which are variants of the same core type. And this is only a quarter of the data—there are three other scenarios with results like these on the same table!

The table does provide all the information, but instead we can use a visualization. For our visualization we group methods by main type, and truncate outliers to put more focus on central tendencies. Below we plot the full Table 2 in this way, as one example. Our table corresponds to the second row in the plot; the other rows are the tables not shown (but see the paper itself). The outliers are circled to make their truncation clear.



Our plot makes the overall story more clear: the variants within method are generally the same, except for the TML estimator under the “neither” scenario.

Many of the estimators have very similar performances to each other across the scenarios considered, but IPW approaches can be biased and more unstable.

12.2 Visualization

We see visualization as the primary technique for communicating simulation results. To illustrate some illustration principles, we next present a series of visualizations drawn from published simulations, each of which highlights some visualization design principles that we believe are broadly relevant and useful. In Chapter 13, we present a more detailed look at the process of developing a polished visualization through iterative refinement of a series of plots.

12.2.1 Relative performance of treatment effect estimators

In Section 12.1.1, we used a table to summarize the performance of a range of different methods of assessing individual-level variation in treatment effects. The table provided a compact high-level summary, but it could be improved by turning it into a visualization.

As a starting point, Figure 12.2 shows the same data as reported in the table, with the values of each performance measure depicted as bars (after rescaling `R2` by 100 to put it on a similar scale to the other measures). The Figure makes it easier to visually assess the extent to which the estimators follow consistent patterns in across the performance measures. However, this plot does not have any representation of the degree of variation across scenarios because the bars only represent the *average* performance across scenarios. We can elaborate this plot by using boxplots that summarize the distribution of performance measures across all scenarios examined.

The simulation evaluated performance across 324 distinct scenarios, and Figure 12.3 shows the range of relative performances for each estimator across all of them. Using boxplots instead of bars highlights how much the relative performance can change from scenario to scenario. Just as in Table 12.2 and Figure 12.2, we order the methods from highest average RMSE to lowest. We use little circles to represent the average performance across all the simulations as little circles (these would correspond to the bars in Figure 12.2). We truncate extreme values to make the plot more readable and bring focus to central tendencies. The x-axis is on a log scale, selected to navigate long tails and highlight relative performances. The log scaling also makes the scale of improved performance (less than x1) similar to worse performance (above x1).

Figure 12.3 depicts three different performance measures in a single plot. We dropped R^2 for this plot because the R^2 measure was in percentage points,

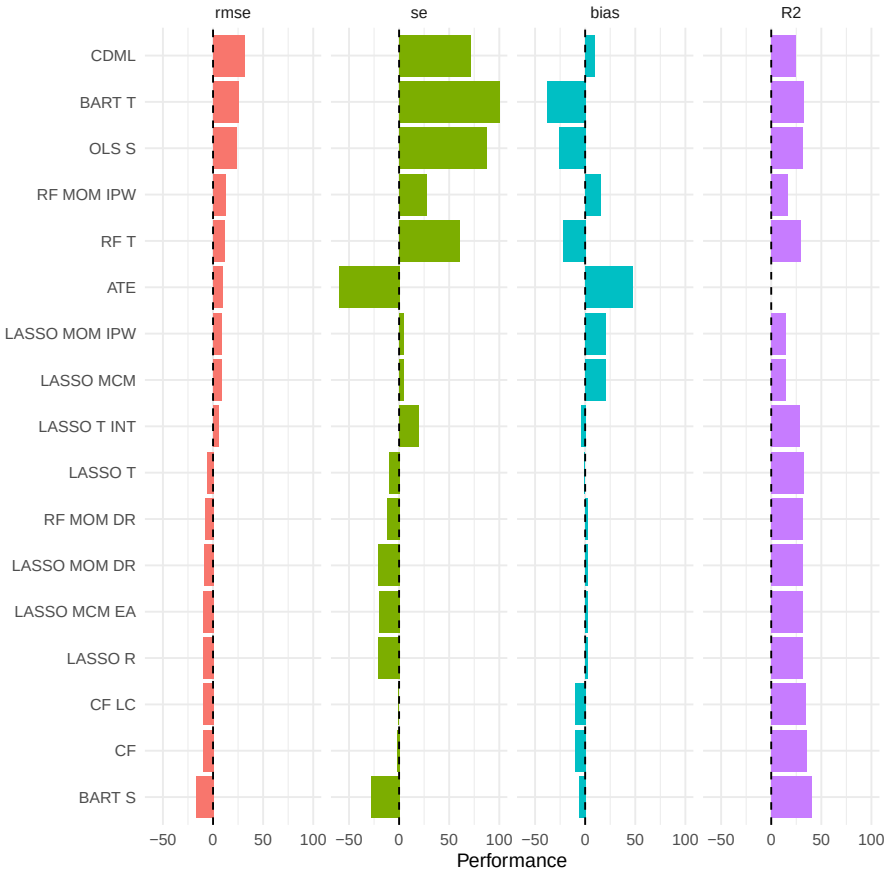


Figure 12.2: Average relative performance of individual treatment effect estimators.

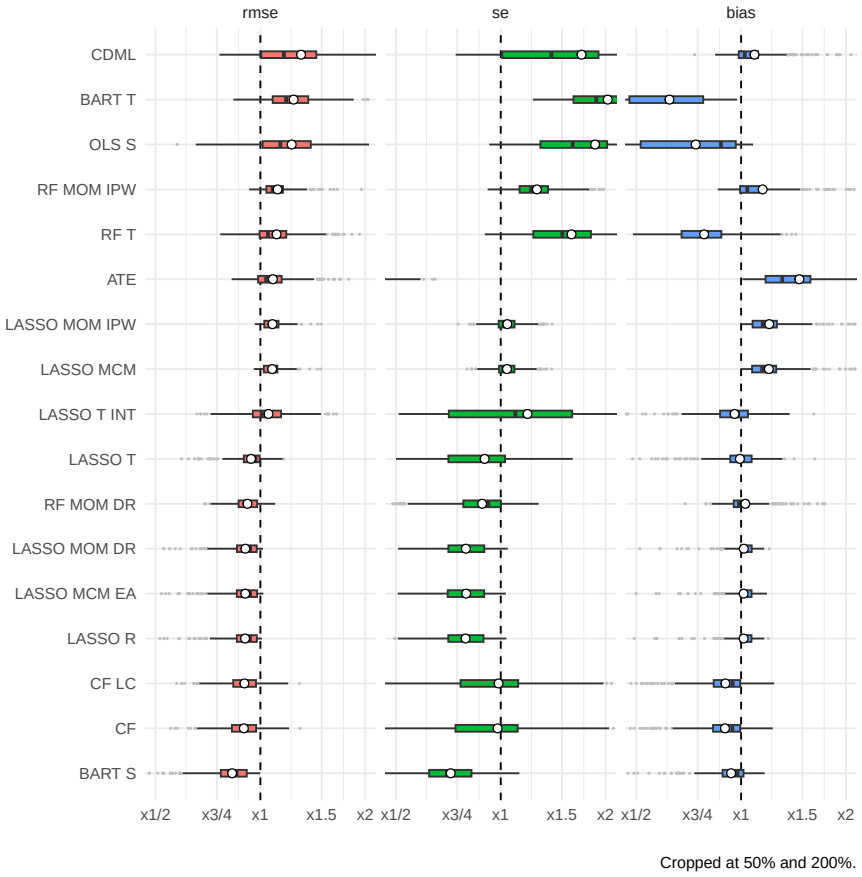


Figure 12.3: Relative performance of individual treatment effect estimators across all simulation scenarios.

and, due to estimation uncertainty, included negative values; this was not compatible with the log scaling.[1] We have lost something in order to gain something. Ultimately, which plot is best is a matter of aesthetic judgement. Pick your plot based on whether it is clearly communicating the message you are trying to get across.

12.2.2 Bias of log response ratios with auto-correlated data

Chen and Pustejovsky (2024) examined the performance of a variety of effect size estimators used for quantifying treatment effects in single-case designs (SCDs), which are repeated measures designs that involve measuring an outcome variable on a single individual both before and after the introduction of a treatment. Typical SCDs involve quite short time series with closely spaced repeated measurements, so there is concern about whether auto-correlation in the outcome measurements affects estimation of effect sizes. Chen and Pustejovsky (2024) reported some small simulations that examined the bias of different effect size estimators, including the log response ratio, under a data-generating process involving Poisson-distributed outcomes with a first-order auto-regression structure.

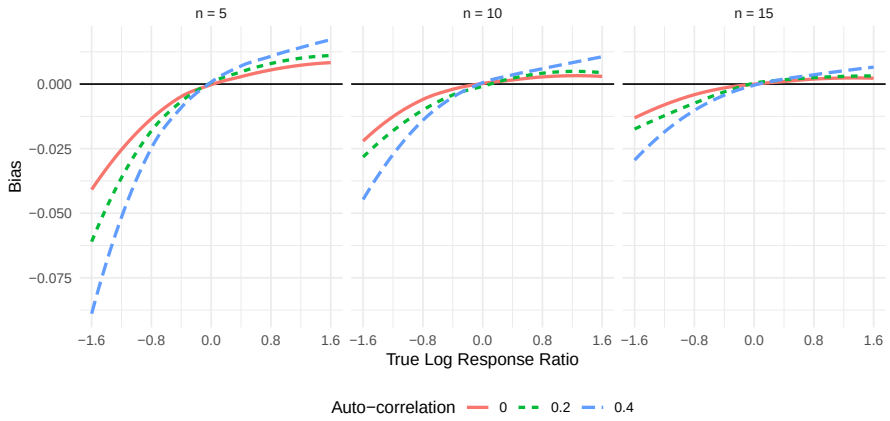


Figure 12.4: Bias of the LRR1 estimator of the log response ratio under a Poisson AR(1) model with baseline mean of $\mu_A = 10$, for varying log response ratio parameters, auto-correlation, and series length.

One of the most common visualizations of simulation results is simply a line chart showing how a performance metric changes in response to some factor

[1] To represent all four performance measures, we could create a separate plot of R^2 and add it to the right-hand side of Figure 12.3. The `patchwork` package (Pedersen, 2024) provides tools for creating composite plots such as this.

of interest.[2] Following the same construction as Figure 1 of Chen and Pustejovsky (2024), Figure 12.4 uses a simple line chart to depict how bias of the LRR1 effect size estimator is affected by the magnitude of the underlying parameter (on the horizontal axis) and the degree of auto-correlation, with varying levels of auto-correlation represented with different colors and different styles of line. We use multiple plots of a similar form to represent patterns across different numbers of data series lengths, from $n = 5$ to $n = 15$ measurements per phase. Figures such as Figure 12.4 clearly show how the bias is affected by the degree of auto-correlation, with larger positive auto-correlation amplifying the bias of the estimator. We also see that the bias is dampened as the series length increases.

12.2.3 Biserial correlation estimation

An extreme groups design is a clever way to evaluate the association between two variables when one of the variables is difficult or expensive to measure. It involves measuring one variable, X, on a sample of participants but measuring the other variable, Y, only for the subset of participants who have X scores in the highest or lowest part of the distribution. The highest and lowest groups are defined based on some cut-off percentiles, which might be based on either the sample distribution or the population distribution of X. The correlation between X and Y can be recovered by computing the biserial correlation coefficient between Y and an indicator for whether the participant is in the highest or lowest part of the distribution.

Pustejovsky (2014) used simulation to examine the bias of biserial correlation estimates under the extreme groups design. This simulation was a full factorial design with four manipulated factors, including the true correlation for a range of values, cut-off type, cut-off percentile, and sample size. Figure 12.5 reproduces a visualization of the bias of the biserial correlation estimator examined in Pustejovsky (2014). The correlation parameter, with 96 levels, runs along the horizontal axis and shows a smoothly varying bias curve. We use line type to represent the sample size, which makes it apparent that the bias shrinks as sample size increases. Finally, facetted plots are used to represent the remaining factors of cut-off type and cut-off percentile. All of the factors—and thousands of distinct simulation scenarios—are visible in a single plot.

To make Figure 12.5, we smoothed the lines with respect to the correlation parameter (ρ) using locally weighted linear regressions (i.e., LOESS smoothing). Smoothing is a nice tool for taking some of the Monte Carlo error out of a figure to more directly represent overall trends.

[2] In a discussion about the importance of simulation, Little (2013) provides several further examples of this type of plot, including figures showing an RMSE and confidence interval coverage for four estimators of a regression coefficient when using a calibration procedure. Antonakis et al. (2021) uses similar plots to report results from a simulation examining what happens when random effect assumptions are ignored in multilevel modeling.

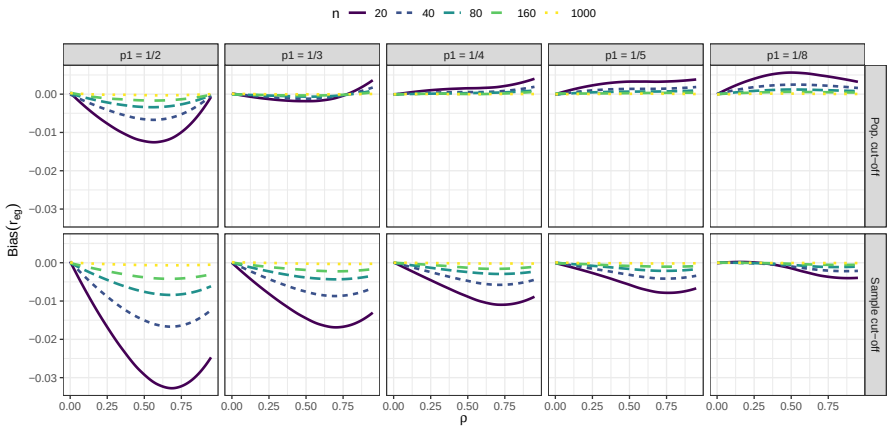


Figure 12.5: Bias of the biserial correlation estimate in an extreme groups design, for varying sample sizes, cut-off thresholds, and cut-off types. Source: Pustejovsky (2014).

This style of visualization, consisting of a grid of small plots, is called "many small multiples" and is beloved by Edward Tufte, who has written extensively on best practices for information design (see, for example, Tufte and Graves-Morris, 1983). Tufte likes many small multiples, in part, because the technique can be used to display many different variables in a single plot: here, our facets are organized by two (`p1` and the cut-off approach), and the plot within each facet represents three (the outcome of bias, ρ on the x-axis, and n with line type). In this four-factor design, the small multiple plot lets us fully represent a performance measure across all combinations of the factors.

12.2.4 Robust Hypothesis Testing for Meta-regression

In our next example, drawn from Tipton and Pustejovsky (2015), we explore Type-I error rates of small-sample corrected F-tests based on cluster-robust variance estimation in meta-regression models. The simulation aimed to compare 5 different small-sample corrections for hypothesis tests involving multiple constraints (i.e., multiple regression coefficients). This was another complex experimental design, which involved varying several factors:

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

The Type I error rates of all five tests are depicted in Figure 12.6.

As in the previous example, Figure 12.6 uses small multiples to show variation

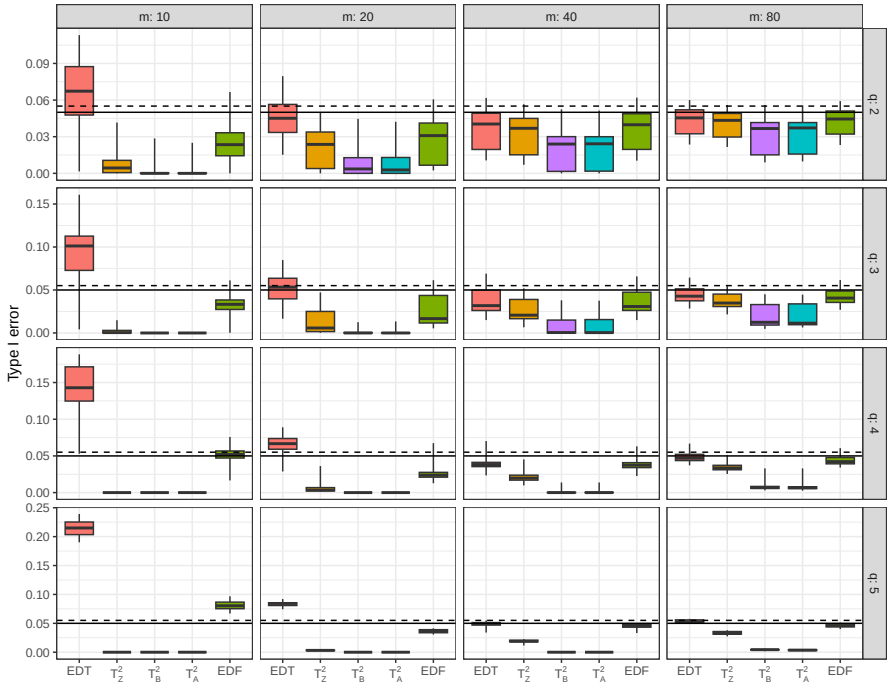


Figure 12.6: Type-I error rates of selected hypothesis testing procedures for meta-regression models with cluster-robust variance estimation. Source: Tipton and Pustejovsky (2015).

in the results across two simulation factors (sample size m and hypothesis dimension q). The x-axis shows each of the five hypothesis testing procedures under comparison. The boxplots depict the *distribution* of the performance measure across the remaining factors in the design; Each box shows the range, inter-quartile range, and median of the Type-I error rates for a given test procedure across the various covariates tested and different degrees of model misspecification. For reference, we add a solid line at the target .05 rejection rate and a dashed line at the upper bound of a 90% confidence interval for an estimate with a true rejection rate of .05. The spread of the boxes shows how some methods are more or less vulnerable to different types of misspecification. Some estimators (e.g., T_A^2) are clearly hyper-conservative, with very low rejection rates. Other methods (e.g., EDF), have a range of very high rejection rates when $m = 10$ but converge to the target level as the sample size incrases. For these tests, the degree of rejection rate depends on the degree of model mis-specification and the particular covariates tested (i.e., the remaining factors captured in the boxes).

12.2.5 Heat maps of coverage

For data with many levels of two different factors, one useful visualization technique is a heat map. For instance, Brown et al. (1977) studied methods of estimating the features of a binary (on/off) process occurring over time when that process is observed using momentary time sampling (MTS). MTS involves recording whether a process is on or off at specific moments that are evenly spaced over time. Under certain assumptions, the underlying process can be described by two parameters: prevalence, or the overall proportion of time that the process in on, and incidence, or the rate at which the process cycles between states. Brown et al. (1977) proposed maximum likelihood estimators (MLEs) for the prevalence and incidence parameters based on a sequence of MTS data.

Using simulation, Pustejovsky studied the performance of the maximum likelihood estimator and suggested an alternative of using penalized maximum likelihood and bootstrap confidence intervals. The simulation design involved varying three factors: prevalence, incidence, and the length of the MTS sequence. Because the prevalence parameter can take any value between 0 and 1 and the incidence parameter can take any positive value, we examined a large number of levels of each of these factors, with 10 levels for prevalence and 10 levels for incidence, plus 12 levels for sample size. We limited the levels of prevalence to run from 0.05 to 0.50 because the estimators work symmetrically for prevalence values above and below 0.50.

Figure 12.7 shows the coverage of parametric bootstrap confidence intervals for the prevalence parameter. The plot shows the combinations of prevalence and incidence as a grid for each sample size level. We break coverage into ranges of interest, with green being “good” (near 95%), yellow being “close”

(92.5% or above), blue and purple being conservative (above 95%) coverage, and red being poor (below 92.5%). For this kind of plotting to work effectively, we need the MCSE to be small enough that the structure of the coverage levels are apparent. The figure consists of two panels, with the left panel showing results for smaller sample sizes and the right panel showing results for the larger sample sizes. We *wrap* the small multiples by sample size, so that the overall figure design runs from smallest sample size on the upper left to largest sample size on the lower right. Wrappings can be useful if you have many levels of a single factor, although it does not take advantage of the two-dimensional layout of the grid (such as we saw with examples Section 12.2.3 and Section 12.2.4). Within each panel, results for the original MLE and the penalized MLE are arranged size-by-side to facilitate comparison between methods.

Looking across sample sizes in Figure 12.7, we see clear trends, where the coverage of both estimators degrades for low incidence rates. The field of green in the right column of each panel indicates that the penalized MLE works well across a large part of the parameter space. The left columns of each panel have less green, indicating that the original MLE has worse coverage Comparing left and right, the improvement of the penalized MLE over the original MLE is evident.

12.2.6 Head-to-head comparison of accuracy

Comparing the performance of different estimation methods is often a central focus of simulation studies, and so it is often important to construct figures to facilitate such comparisons. In Examples Section 12.2.1, Section 12.2.4, and Section 12.2.5, we saw figures designed to allow side-by-side comparisons of different estimators. A drawback of the figures presented in Examples Section 12.2.1 and Section 12.2.4 is that they aggregate the results over multiple scenarios, which makes it difficult to know whether one estimator out-performs *always* outperforms another or if it is worse under some conditions but better under others. For instance, in Figure 12.6, does the T_Z^2 test always have lower Type I error than the EDF test? Or are there conditions where EDF is more conservative? There is no way to tell from the results presented because the box-plots are summarizing over multiple scenarios.

Several other plotting techniques can be useful for presenting head-to-head comparisons of performance. To illustrate these, we use data from Chen and Pustejovsky (2025), who studied the performance of a variety of estimators for average effect size in meta-analyses involving dependent effect sizes and selective reporting bias. Each estimator examined had two versions, involving either a univariate working model (where all effect sizes were treated as independent) or a multivariate working model (that treats effect sizes drawn from the same study as correlated). Thus, one of several questions of interest in this simulation was about the degree to which using a multivariate working

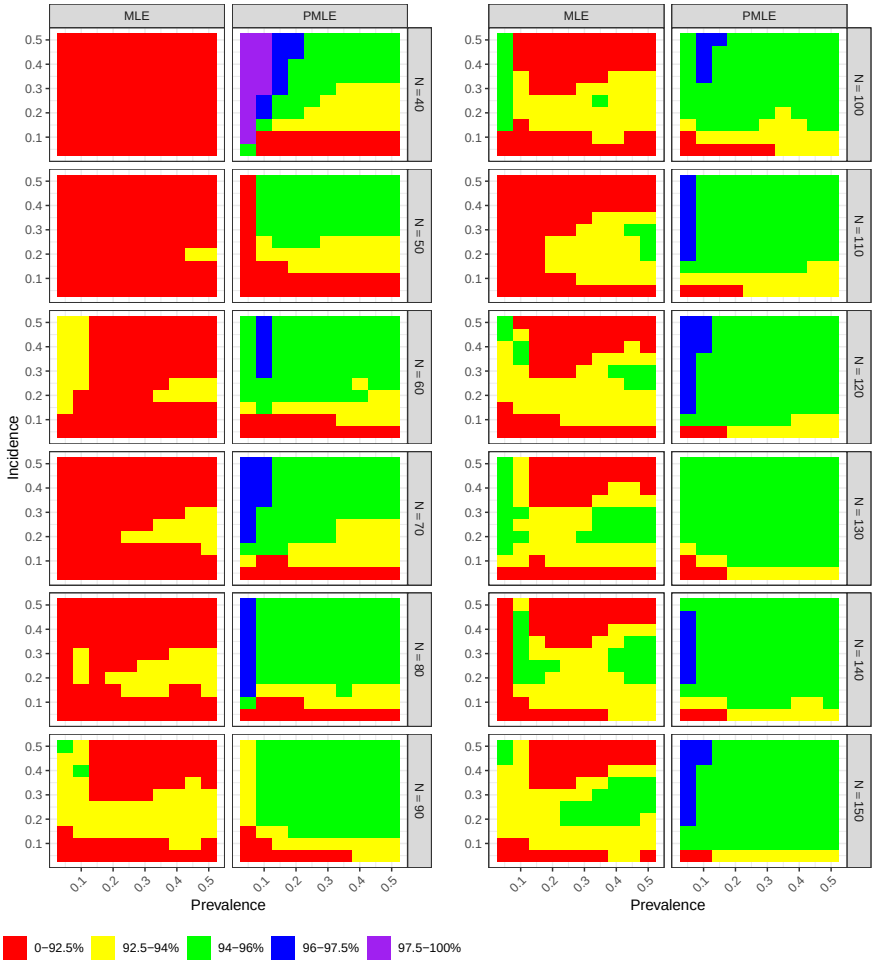


Figure 12.7: Coverage of bootstrap confidence intervals for maximum likelihood estimator and penalized maximum likelihood estimator of prevalence based on momentary time sampling data.

model led to improved accuracy of an estimator.

Figure 12.8 is adapted from Figure 5b of Chen and Pustejovsky (2025) and depicts a subset of the results from our full simulation, focusing on the RMSE of the univariate and multivariate versions of the estimators. Each panel corresponds to a separate estimator. Within a panel, the horizontal and vertical axes correspond to the RMSE of the univariate and multivariate estimator, respectively. Each point represents the performance of the estimators under a single simulation scenario; the color of the point corresponds to the selection weight used to generate the data. A dashed line at $y = x$ indicates the point where both versions have equal accuracy.

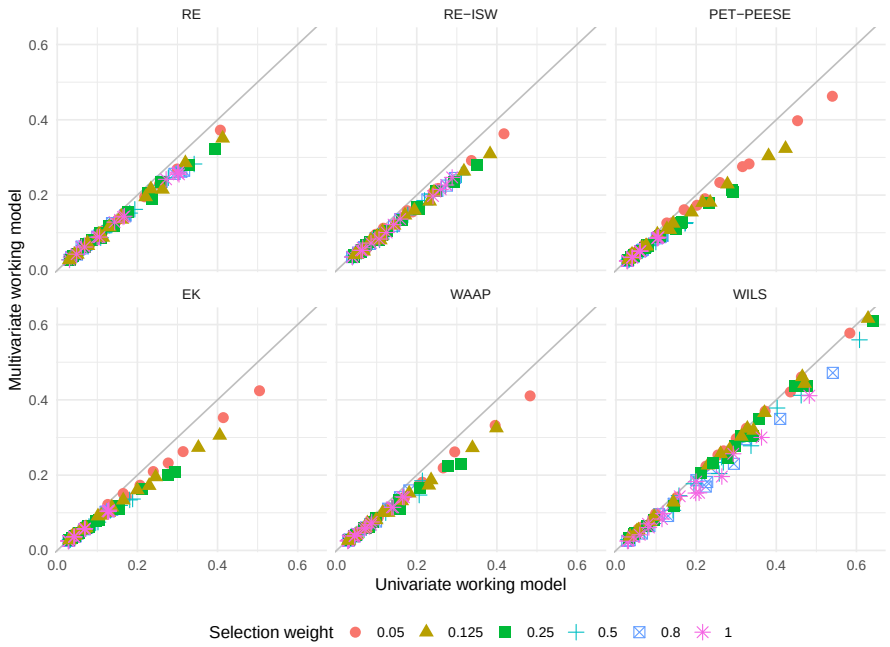


Figure 12.8: Comparison of the RMSE of multivariate working models versus the RMSE of univariate counterparts under a one-step selection process for $k = 30$ studies and $\bar{\rho} = 0.4$.

The head-to-head layout of Figure 12.8 makes it easy to see that the multivariate versions of each estimator tend to have slightly lower RMSE than their univariate counterparts and that this is true not only on average, but also across the conditions depicted in the figure. Furthermore, the performance advantage of the multivariate estimators seems to be approximately multiplicative: using the multivariate version instead of the univariate version reduces RMSE by close to a constant percentage. However, the figure is still limited to a subset of conditions because depicting every sample size and every setting

for $\bar{\rho}$ would make it too noisy to see these trends. A further drawback of this layout is that it is difficult to see the relationship for scenarios where both versions of an estimator have relatively low RMSE; instead, the eye tends to be drawn to outliers (such as the red dots with the highest values of RMSE).

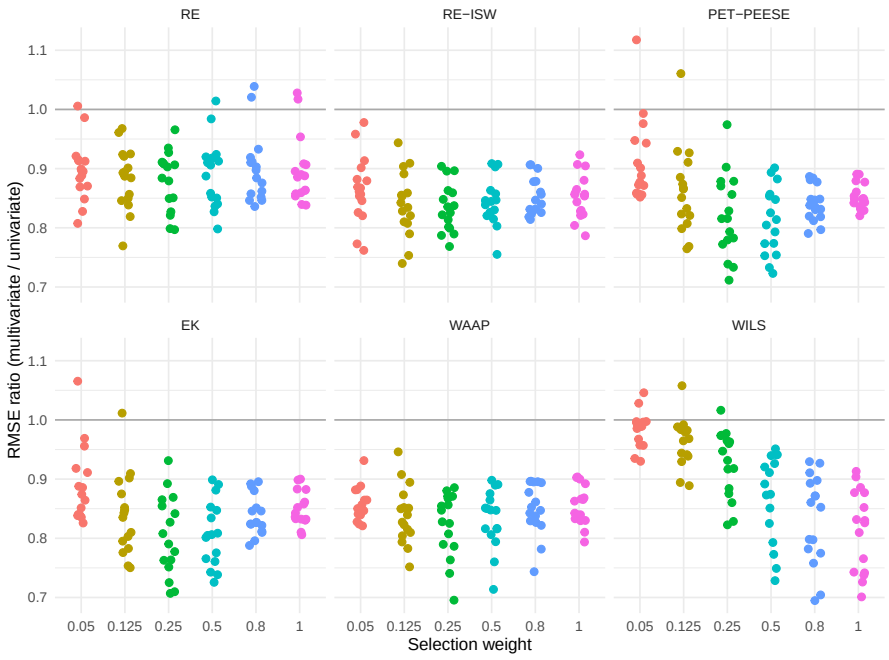


Figure 12.9: Ratio of RMSE of multivariate working models to the RMSE of univariate counterparts under a one-step selection process for $k = 30$ studies and $\bar{\rho} = 0.4$.

Figure 12.9 illustrates another way to depict head-to-head comparisons of the accuracy of two estimators. Rather than plotting one versus the other, we have calculated and plotted the ratio of the RMSE for the multivariate estimator relative to the RMSE of the univariate version. Thus, RMSE ratios less than one correspond to scenarios where the multivariate version is more accurate. Each point still corresponds to a unique scenario, and the number of points in each panel is identical to the number in Figure 12.8. This representation makes it much easier to see how relative accuracy is affected by the selection weight; for instance, the advantage of the multivariate versions of the PET/PEESE, EK, and WILS estimators tend to weaken for smaller selection weights. Further, plotting at the RMSE ratio has freed up one dimension of the graph, which could allow us to represent one further dimension of the simulation design in the figure.[3] The main drawback of plotting the ratio is

[3]Figure 12.9 uses color to represent different levels of the selection weight parameter, even

that the figure does not provide information about the absolute magnitude of the RMSE.

12.3 Modeling

Simulations are designed experiments, often with a full factorial structure. The results are datasets in their own right, just as if we had collected data in the wild. We can therefore leverage classical techniques for analyzing full factorial experiments. For example, regressing a performance measure against our factor levels provides a quantitative description of how the different levels influence performance, holding the other factors constant. This type of regression has been described as a "meta-regression" (Kleijnen, 1981; Friedman and Pressman, 1988; Gilbert and Miratrix, 2024) because it involves a regression model where the response variable is itself a summary estimate of some performance measure.[4] The technique also has ties to meta-analysis (see, e.g., Borenstein et al., 2021), which involves summarizing and examining trends across sets of empirical findings from different studies.

One useful summary of a meta-regression model is an analysis of variance (ANOVA) table that decomposes the variation in the performance measure into different sources. In the language of a full factorial experiment, the ANOVA table typically includes both "main effects" and "interaction effects." Main effects characterize the extent to which a particular factor systematically influences performance, when averaging across all other factors in the design. By aggregating over other factors, main effects will capture the broadest possible patterns; if we see a trend when we average over all the other varying factors, then we can infer that the finding is relevant across the many different simulation contexts explored, rather than being an idiosyncratic feature of a specific and narrow context. Interaction effects characterize the degree to which patterns or trends observed in one factor are themselves variable, changing depending on the level of another factor. Interactions provide an indication of how the influence of a particular factor might matter more or less, depending on other aspects of the context.

In simulations that involve comparing multiple methods, we might approach meta-regression models in one of two ways. One possibility would be to include

though this is redundant with the horizontal axis of each panel. We have used this redundant mapping to emphasize the parallels with Figure 12.8, even though it is not strictly necessary. We could instead add further results to the figure by using color to represent different levels of the $\bar{\rho}$ parameter.

[4] Meta-regression can also be used to account for Monte Carlo uncertainty in some contexts, which can be especially important when the number of iterations per scenario is low. Gilbert and Miratrix (2024) examine this point in greater detail.

the method itself as a factor in the meta-regression. With this approach, the estimated main effect for each method will describe the extent to which a method is higher or lower than the baseline method, on average across all the simulation scenarios. The main effects of the simulation factors will tell us the extent to which each factor influences the performance measure on average across all the methods considered. For instance, we might expect that the true standard error goes down as sample size increases, following a consistent pattern across all methods. With this approach, we would need to examine interactions between method and the other factors, to see how the methods vary in the extent to which they are influenced by the simulation scenario.

Another approach would be to use separate meta-regression models for each of the multiple methods. This approach makes it somewhat easier to understand the influence of the simulation design factors on the performance of each method. Using separate meta-regressions, we might find that a certain factor has a very strong influence on the bias and RMSE of one particular method, whereas it has little or no influence on the performance of a different method. The challenge with this approach is that it becomes more difficult to identify patterns that hold across all methods or to draw comparisons across different methods.

12.3.1 Comparing methods for cross-classified data

Lee and Pustejovsky (2023) were interested in evaluating how different modeling approaches perform when analyzing cross-classified data structures in which students were nested both in schools and in neighborhoods. To do this, we conducted a multi-factor simulation to compare three methods: a cross-classified random effects model (CCREM), a simple, ordinary least squares model with two-way cluster-robust variance estimation (CRVE), and a two-way fixed effects model, also with CRVE. The simulation was complex, involving several factors including the magnitude of effect (r), the number of schools (H), the number of students per school (J), a school-level intra-unit correlation coefficient (IUCC), and whether the data-generating process involved violations of certain assumptions of the CCREM.

To understand which factors had the most influence on the bias of the three methods, we used an ANOVA model. We were interested in understanding the factors that influenced bias both when all assumptions of the CCREM held and when each of several assumptions was violated. We therefore computed four separate ANOVA summaries—one for each scenario where all assumptions were met, where homoskedasticity was violated, where exogeneity was violated, and where there were unmodeled random slopes. This approach let us understand which of the remaining factors were important influences on the parameter bias of the methods examined.

The supplementary materials of Lee and Pustejovsky (2023) (Table S5.2, avail-

able at https://osf.io/hy73g and reproduced as Table 12.3) reported the results of these four ANOVA models, with columns corresponding to each of the four assumption scenarios and rows corresponding to factors manipulated within that context. The magnitude of the effects is expressed as partial η^2 effect sizes; Small, medium, and large effects are marked to make them jump out to the eye.

Table 12.3: Partial η^2 values from ANOVA of parameter bias in a simulation of cross-classified data

Source	Assumptions Met	Homoskedasticity Violated	Exogeneity Violated	Random Slopes
Method	0.000	0.006	0.995 L	0.000
Effect Size (r)	0.131 M	0.008	0.020 S	0.142 L
Number of Schools (H)	0.014 S	0.113 M	0.188 L	0.001
Students per School (J)	0.016 S	0.016 S	0.747 L	0.110 M
IUCC	0.007	0.007	0.033 S	0.073 M
method × r	0.006	0.006	0.007	0.012 S
method × H	0.004	0.002	0.157 L	0.000
method × J	0.002	0.000	0.878 L	0.010
method × IUCC	0.012 S	0.003	0.037 S	0.000
r × H	0.103 M	0.010	0.059 S	0.377 L
r × J	0.006	0.024 S	0.008	0.051 S
r × IUCC	0.065 M	0.025 S	0.084 M	0.136 M
H × J	0.002	0.014 S	0.062 M	0.105 M
H × IUCC	0.024 S	0.008	0.034 S	0.137 M
J × IUCC	0.004	0.088 M	0.013 S	0.029 S

Note: (S)mall = .01, (M)edium = .06, (L)arge = .14

In the second column of Table 12.3, we see that when model assumptions are met or only homoscedasticity is violated, the choice of method (CCREM, OLS-CRVE, FE-CRVE) has almost no impact on parameter bias ($\eta^2 = 0.000$ to 0.006). However, under an exogeneity violation, method choice has a large effect ($\eta^2 = 0.995$), indicating that some methods (e.g., OLS-CRVE) have much more bias than others. Other factors such as the effect size of the parameter and the number of schools can also show moderate-to-large impacts on bias in several conditions. Table 12.3 also shows how an interaction between simulation factors can matter. For example, interactions between method and number of schools, or students per school, can really impact bias under the Exogeniety Violated condition; this means the different methods respond differently as sample size changes. Overall, the ANOVA exercise was a useful

preliminary step in our analysis because it provided a succinct summary of how some aspects of the simulation's design matter more and some matter less. We used the results to inform the design of visualizations for illustrating key findings from the simulation.

12.3.2 Biserial correlations, revisited

In the biserial correlation example described in Section 12.2.3, we saw that bias can change notably across scenarios considered, and that several factors appear to be driving these changes. These factors also seem to have complex interactions: note how when p1 = 0.5, we get larger dips than when p1 = 1/8. Figure 12.5 gives a sense of the complex combination of factors that influence bias and also suggests some possibilities for further analysis. In particular, the figure suggests that the bias grows smaller as the sample size increases. We used this observation to develop a model in which the bias is inversely proportional to the sample size, as in

$$B_{pqrs} = \beta_{pqr} \times \frac{1}{N_s} + e_{pqrs},$$

where B_{pqrs} is the estimated bias for cut-off level p, cut-off type q, true correlation level r, and sample size level s, N_s is the actual value of sample size level s, and e_{pqrs} is an error capturing the Monte Carlo uncertainty in the bias estimate.

Because the model reduces the dimension of the simulation's design, relying on it allow us to simplify further analysis and presentation of the simulation results. For instance, we used the model to motivate construction of Figure 12.10, which shows how β_{pqr} varies for different cut-off values, cut-off types, and correlation parameters. The results presented are aggregated across the sample size factor because we have fully accounted for how it affects bias.

12.4 Reporting

Just as with empirical data analysis, the analysis that you present in an outward-facing report will usually be only a summary of all of the analysis that you conduct to arrive at your findings. Typically, you will need to make many figures and tables, and potentially explore several different modeling techniques in order to identify the important patterns in a set of simulation results. You need to work through such analysis in order to develop a strong understanding of what is going on in a set of simulation results. You will then need to clarify the story by distilling your full analysis into a selection of figures, tables, and summaries that explain your findings while maintaining

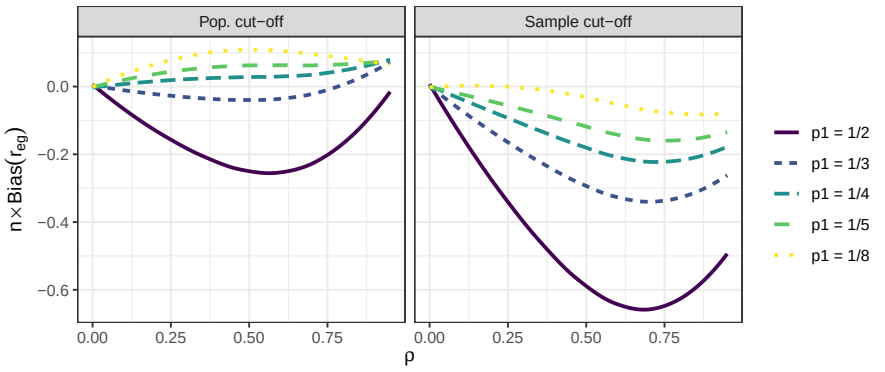


Figure 12.10: N times the bias of the biserial correlation estimate in an extreme groups design as a function of ρ, cut-off thresholds, and cut-off types.

honesty and transparency. Some of the remainder will then become supplementary materials that contain further detail to both enrich your main narrative and demonstrate that you are not hiding anything.

When presenting results, it is important to not pummel readers with a deluge of tables, figures, and observations. Instead, strive to present summaries of results that clearly illustrate the main findings from the study—that is, the patterns and trends that most directly respond to the research questions—and also draw attention to any results that are unusual or anomalous. Often, an audience will be best served with a few well-chosen figures. In the text of a write-up, you might also include a few specific numerical comparisons. Do not include too many of these, and be sure that it is clear why the numerical comparisons you include are important.

Findings from a simulation study are by definition a simplified summary of a complex phenomenon. Alert readers will know this and will be cautiously attuned to (or outright skeptical about) details not represented in the final set of exhibits. To support transparency, you should also provide code for reproducing the entire simulation, along with data files containing the full detail of your simulation results. Sharing code allows any interested reader to rerun the simulation and conduct your analysis themselves or to rerun your simulation under different conditions. Sharing a data file of simulation results allows readers to verify or further probe the patterns that you identify in your write-up, to visualize or analyze results in novel ways, or to examine subsets of conditions that might be of particular interest.

13

Building good visualizations

Visualization should nearly always be the first step in analyzing simulation results. In the prior chapter, we saw a variety of examples primarily taken from published work. Those visualizations were not the initial ones created for those research projects. In practice, getting to a good visualization often requires creating *many* different graphs to look at different aspects of the data. From that pile of graphs, you would then curate and refine those that communicate the overall results most cleanly.

In our work, we find we often generate a series of R Markdown reports with comprehensive sets of charts targeting our various research questions. These initial documents are then discussed internally by the research team.

In this chapter we first discuss four essential tools that we frequently use to make these initial sets of graphs:

1. **Subsetting**: Multifactor simulations can be complex and confusing. Sometimes it is easier to first explore a subset of the simulation results, such as a single factor level.
2. **Many small multiples**: Plot many results in a single plot, with facets to break up the results by simulation factors.
3. **Bundling**: Group the results by a primary factor of interest, and then plot the performance measure as a boxplot so you can see how much variation there is within that factor level.
4. **Aggregation**: Average the performance measure across some of the simulation factors, so you can see overall trends with respect to the remaining factors.

To illustrate these tools, we walk through them using our running example of comparing methods for analyzing a Cluster RCT. We will start with an investigation of bias.

As a reminder, in our Cluster RCT example, we have three methods for estimating the average treatment effect: linear regression of the student-level outcome onto treatment (with cluster-robust standard errors); aggregation, where we regress the cluster-level average outcome onto treatment (with heteroskedastic robust standard errors); and multilevel modeling with random

effects. We want to know if these methods are biased for our defined estimand, which is the cluster-average treatment effect. We have five simulation factors: school size (`n_bar`), number of schools (`J`), the intraclass correlation (`ICC`), the degree of variation in school size (`alpha`), and the relationship between school size and treatment effect (`size_coef`).

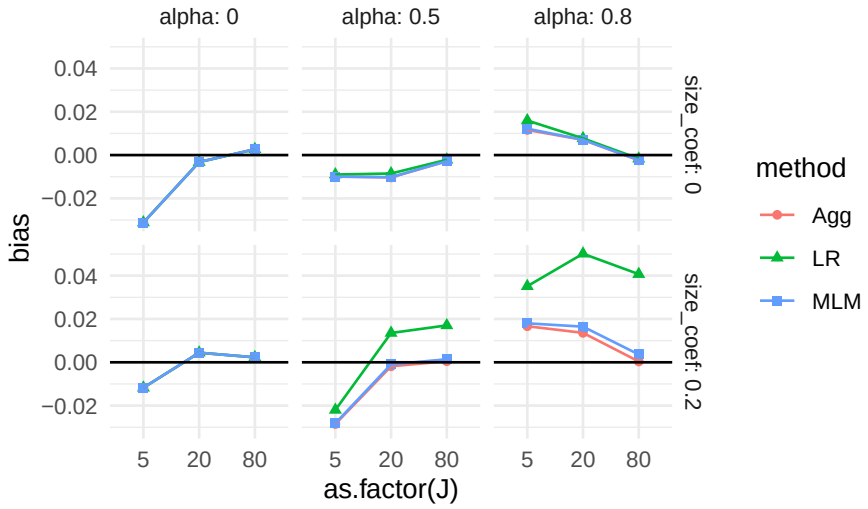
Once we go through the four core tools, we continue our evaluation of our simulation to show how we can assess other performance metrics of interest (true standard error, RMSE, estimated standard errors, and coverage) using these tools. We do not dive deeply into validity or power; see Chapter 19 for more on those measures.

13.1 Subsetting and Many Small Multiples

"Small Multiples" is when you make a plot that is actually a bunch of related plots. These plots can themselves be organized in a plot-like structure, with an x-axis depending on one categorical variable, and the y-axis another (this is a "facet grid").

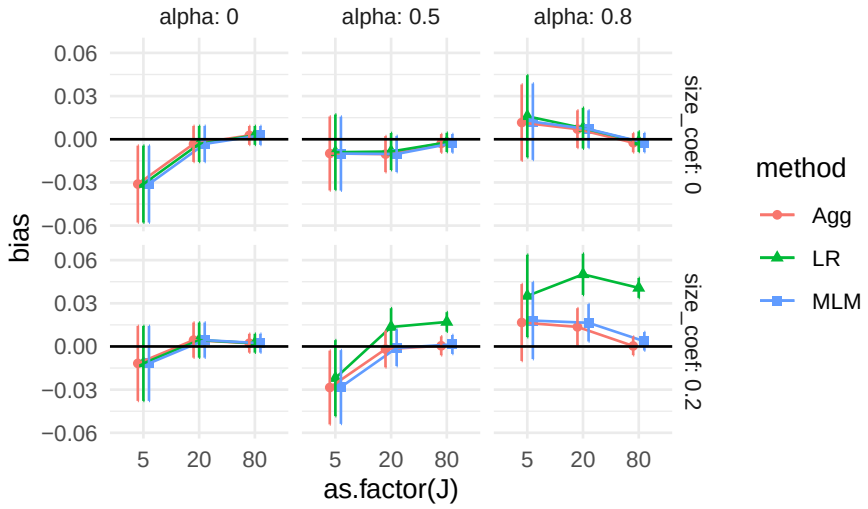
Generally, with small multiples, it is relatively straightforward to look at *three simulation factors*. This is because, with a facet grid, you can easily plot five aspects of your data: two for the facet x- and y-axis arrangement, one for the within-facet x-axis, and one for the within-facet y-axis, and one for color/line type. Of the five aspects, we usually use one of these for method (if we are comparing methods) and one for the performance metric (the outcome), giving three remaining aspects to assign to our simulation factors. In cases with more than three factors, an easy initial approach is pick three favorite factors and then filter all the simulation results down to specific levels for the remaining factors.

For example, for our simulation, we might target the ICC of 0.20, taking it as a "reasonable" value that, given our substance matter knowledge, we know is frequently found in empirical data. We might further pick $\bar{n} = 80$, as our middle level for cluster size. This leaves three factors (`J`, `size_coef`, and `alpha`), allowing us to plot all of our simulation results for our subset. We then decide to plot bias (our performance measure) as a function of `J`, with different lines for the different methods, and facets for the different levels of `size_coef` and `alpha`.

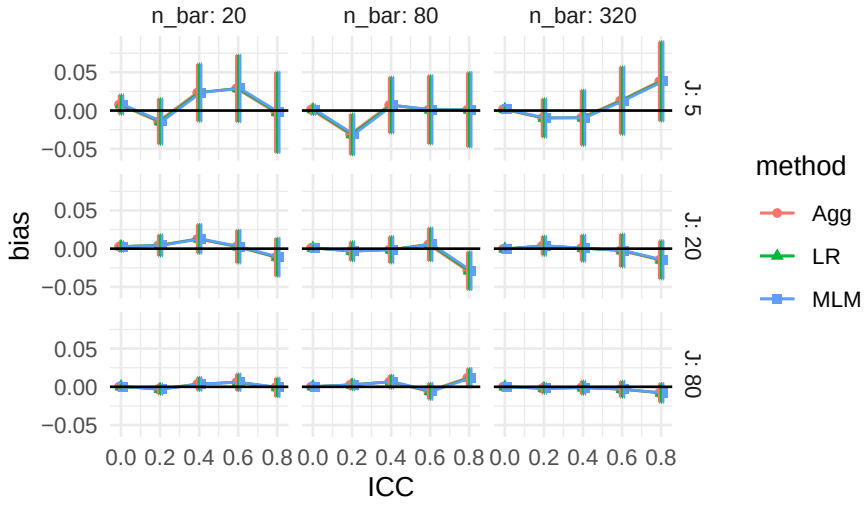


Each point is one of our methods in one of our simulation scenarios. We are looking at the raw results. We connect the points with lines to help us see trends within each of the small multiples. The lines help us visually track which group of points goes with which.

We immediately see that when there is a lot of site variation (`alpha = 0.80`), and it relates to outcome (`size_coef=0.2`), linear regression is very different from the other two methods. We also see that when `alpha` is 0 and `size_coef` is 0, we may also have a negative bias when $J = 5$. Before we get too excited about this surprising result, we add MCSEs to our plot to see if this is a real effect or just noise:



Our confidence intervals exclude zero! Our excitement mounts. We next subset our overall results again to check all the scenarios with `alpha = 0`, and `size_coef = 0` to see if this is a real effect. We can still plot all our data, as for that subset of scenarios we still only have three factors, `ICC`, `J_bar`, and `n_bar`, left to plot (we omit the code as it is similar to above).



It appears that our `ICC=0.20` and `n_bar=80` scenario was just a fluke. Almost all of our confidence intervals easily cover 0, and we are not seeing any major trends. Overall we conclude that indeed, when there is no variation, or when the variation does not relate to impact, nothing is particularly biased. Inter-

estingly, we further see that all methods seem to give identical estimates when there is no site variation, regardless of ICC, J, or `n_bar` (identical estimates is the easiest explanation of all the estimated biases and associated MCSEs across the three methods being identical).

Subsetting is a very useful tool, especially when the scope of the simulation feels overwhelming. And as we just saw, it can also be used as a quick validity check: subset to a known context where we know nothing exciting should be happening to verify that indeed nothing is there.

Subsetting allows for a deep dive into specific context. It also can make it easier to think through what is happening in a complex context; think of it as a flashlight, shining attention on one part of your overall simulation or another, to focus attention and reduce complexity. Sometimes we might even just report the results for a subset in our final analysis and put the analysis of the the remaining scenarios elsewhere, such as an online supplemental appendix. In this case, it would then be our job to verify that our reported findings on the main results indeed were echoed in the set-aside runs.

Subsetting is useful, but if you do want to look at all your simulation results at once, you need to somehow aggregate or group your results to make them all fit on the plot. We next present bundling, a way of keeping the core idea of small multiples to show all of the raw results, but now in a semi-aggregated way.

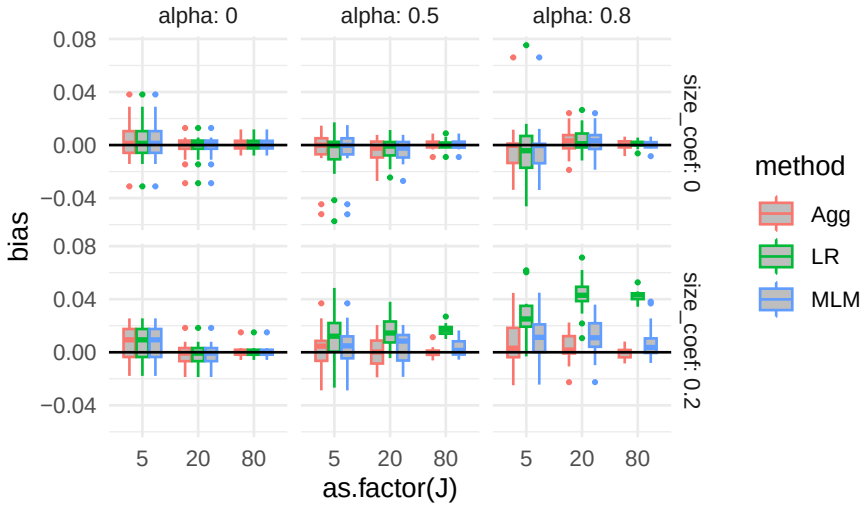
13.2 Bundling

When faced with many simulation factors, we can *bundle* the simulations into groups defined by a selected primary factor of interest, and then plot each bundle with a boxplot of the distribution of a selected performance criteria. Each boxplot shows the central measure of how well an estimator worked across a set of scenarios, along with a sense of how much that performance varied across those scenarios in the box. If the boxes are narrow, then we know that the variation across simulations within the box did not impact performance much. If the boxes are wide, then we know that the factors that vary within the box matter a lot for performance.

With bundling, we generally 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 (wider) than the truth. Consider a case where all the scenarios within a box have zero *true* bias; if the MCSE were large, the *estimated* biases would still vary and we would see a wide boxplot when we should not.

To illustrate bundling, we replicate our small subset figure from above, but instead of each point (with a given `J`, `alpha`, and `size_coef`) just being the single scenario with `n_bar=80` and `ICC = 0.20`, we plot all the scenarios (across the `n_bar` and `ICC` values) in a boxplot at that location. We put the boxes for the three methods side-by-side to directly compare them:

```
ggplot( sres, aes( as.factor(J), bias, col=method,
                   group=paste0(method, J) ) ) +
  facet_grid( size_coef ~ alpha, labeller = label_both ) +
  geom_boxplot( width=0.7, fill="grey", outlier.size = 0.5 ) +
  geom_hline( yintercept = 0 ) +
  theme_minimal()
```



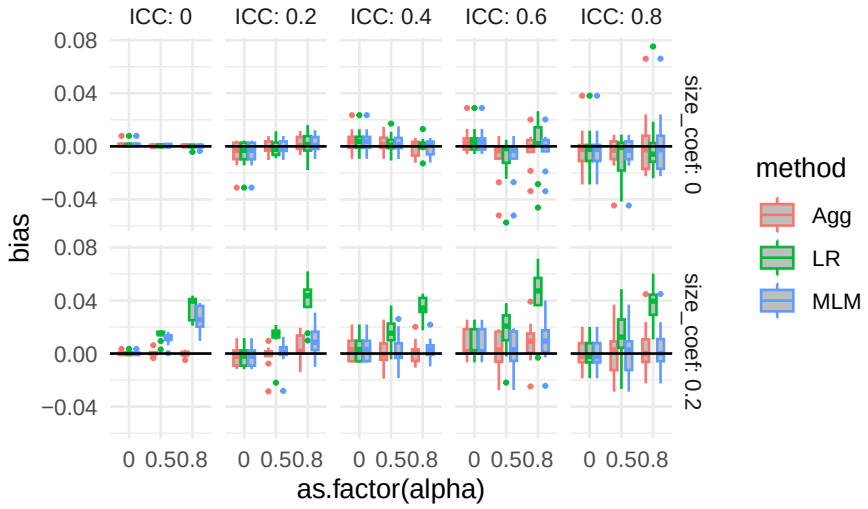
All of our simulation trials are represented in this plot. Each box is a collection of trials. E.g., for `J = 5`, `size_coef = 0`, and `alpha = 0.8` each of the three boxes contains 15 scenarios representing the varying ICC and cluster size. Here are the 15 results in the `J = 5` box in the top right facet for the Aggregation method:

```
filter( sres,
        J == 5,
        size_coef == 0,
        alpha == 0.8,
        method=="Agg" ) %>%
  dplyr::select( n_bar, J, size_coef, ICC, alpha, bias, bias_mcse ) %>%
  arrange( bias ) %>%
  knitr::kable( digits = 2 )
```

n_bar	J	size_coef	ICC	alpha	bias	bias_mcse
80	5	0	0.6	0.8	-0.03	0.02
320	5	0	0.8	0.8	-0.02	0.02
20	5	0	0.6	0.8	-0.02	0.02
80	5	0	0.8	0.8	-0.02	0.03
20	5	0	0.4	0.8	-0.01	0.02
20	5	0	0.2	0.8	-0.01	0.01
320	5	0	0.2	0.8	0.00	0.01
320	5	0	0.4	0.8	0.00	0.02
20	5	0	0.0	0.8	0.00	0.01
80	5	0	0.4	0.8	0.00	0.02
320	5	0	0.0	0.8	0.00	0.00
80	5	0	0.0	0.8	0.00	0.00
320	5	0	0.6	0.8	0.01	0.02
80	5	0	0.2	0.8	0.01	0.01
20	5	0	0.8	0.8	0.07	0.03

Our bias boxplot makes some trends clear. For example, we see that there is no bias, on average, for any method when the size coefficient is 0 and alpha is 0, especially when $J = 80$. When the size coefficient is 0.2, we also see LR jump out from the others when `alpha` is not 0.

The apparent outliers (long tails) for some of the boxplots suggest that the two remaining factors (ICC and cluster size) could relate to the degree of bias. They could also be due to MCSE, and given that we primariy see these tails when J is small, this is a real concern. MCSE aside, a long tail means that some scenario in the box had a high level of estimated bias. We could try bundling along different aspects to see if either of the remaining factors (e.g., ICC) explains these differences. Here we try bundling cluster size and number of clusters (we again omit the code).



We have some progress now: the long tails are primarily when the ICC is high, but we also see that MLM has bias when the ICC is 0, if alpha is nonzero.

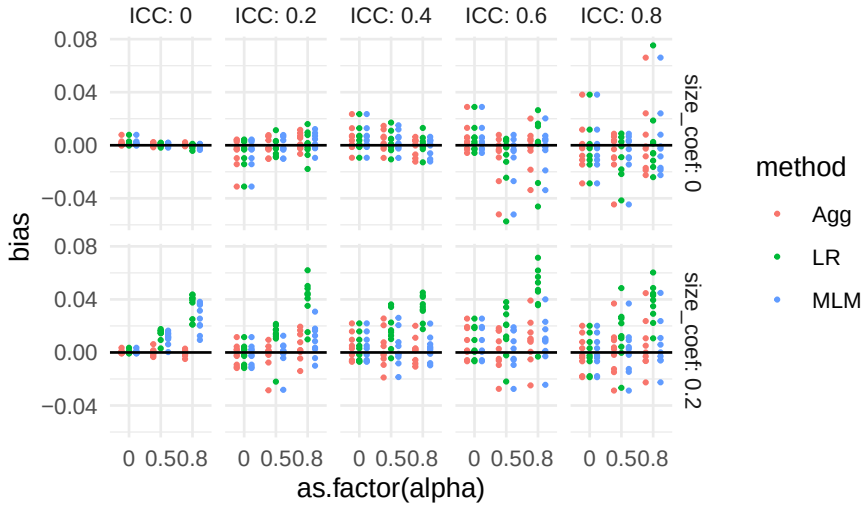
Estimates of bias will be more unstable in smaller samples sizes, because the estimators themselves are more unstable (recall the MCSE of bias is $SE/\sqrt{R}$, with R the number of trials and SE is the true standard error of the estimator). Because of this instability, the tails could still be MCSE, with some of our bias estimates being large due to random chance. Or perhaps there is still some specific combination of factors that allow for large bias (e.g., perhaps small sample sizes makes our estimators more vulnerable to bias). In an actual analysis, we would make a note to investigate these anomalies later on.

In general, trying to group your simulation scenarios so that their boxes are generally small is a good idea; small boxes means that you have found a representation of the data where you know what is driving the variation in your performance measure, and that the factors bundled inside the boxes are less important. This might not always be possible, if all your factors matter; in this case the width of your boxes tells you to what extent the bundled factors matter relative to the factors explicitly present in your plot.

One might wonder, with only few trials per box, whether we should instead look at the individual scenarios. Unfortunately, that gets a bit cluttered:

```
ggplot( sres, aes( as.factor(alpha), bias, col= method,
                   group=paste0(alpha,ICC,method) ) ) +
  facet_grid( size_coef ~ ICC, labeller = label_both ) +
  geom_point( size = 0.5,
              position = position_dodge(width=0.7 ) ) +
  geom_hline( yintercept = 0 ) +
```

```
  theme_minimal()
```



Using boxplots, even over such a few number of points, notably clarifies a visualization.

13.3 Aggregation

Boxplots can make seeing trends more difficult, as the eye is drawn to the boxes and tails, and the range of your plot axes can be large due to needing to accommodate the full tails and outliers of your results; this can compress the mean differences between groups, making them look small. They can also be artificially inflated, especially if the MCSEs are large. Instead of bundling, we can therefore aggregate, where we average all the scenarios within a box to get a single number of average performance, to see overall trends.

When we aggregate, i.e. average over some of the factors, we collapse our simulation results down to fewer moving parts. Aggregation across factors is better than not having varied those factors in the first place! A performance measure averaged over a factor is a more general answer of how things work in practice than having not varied the factor at all.

For example, if we average across ICC and site variation, and see that our methods had different degrees of bias as a function of J, we would know that the found trend is a general trend across a range of scenarios defined

by different ICC and site variation levels, rather than a specific one tied to a single ICC and amount of site variation. Our conclusions would then be more general: if we had not explored more scenarios, we would not have any sense of how general our found trend might be.

That said, if some of our scenarios had no bias, and some had large bias, when we aggregated we would report that there is generally a moderate amount of bias. This would not be entirely faithful to the actual results. But when the initial boxplots show results generally in one direction or another, then aggregation can be quite faithful to the spirit of the results.

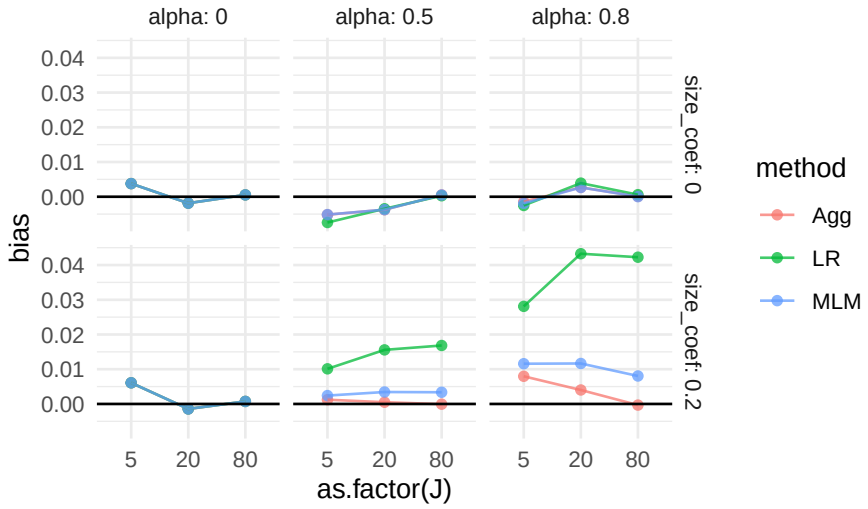
A major advantage of aggregation over the bundling approach is we can have fewer replications per scenario. If the number of replicates within each scenario is small, then the performance measures for each scenario is estimated with a lot of error; the aggregate, by contrast, will be an average across many more replicates and thus give a good sense of *average* performance. The averaging, in effect, gives a lot more replications per aggregated performance measure.

For our cluster RCT, we might aggregate our bias across our sample sizes as follows:

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

We now have a single bias estimate for each combination of size_coef, J, and alpha; we have collapsed 15 scenarios into one overall scenario that generalizes bias across different average cluster sizes and different ICCs. We can then plot, using many small multiples:

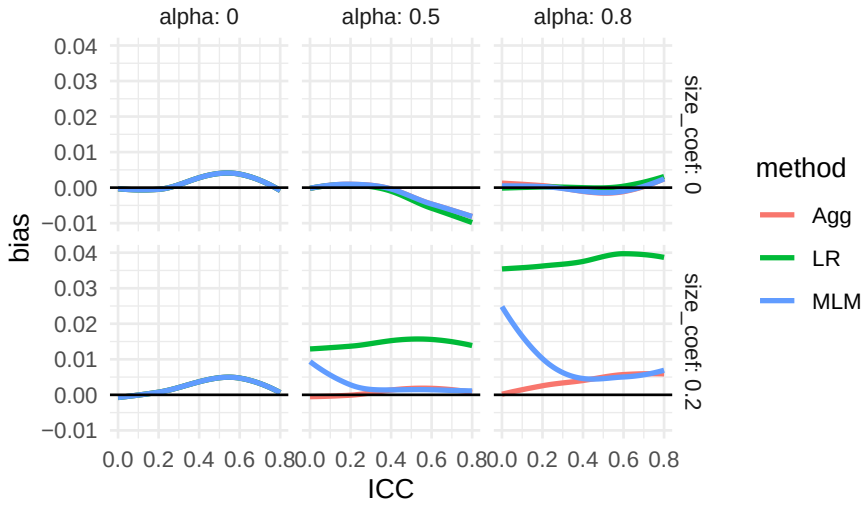
```
ggplot( ssres, aes( as.factor(J), bias, col=method, group=method ) ) +
  facet_grid( size_coef ~  alpha, labeller = label_both ) +
  geom_point( alpha=0.75 ) +
  geom_line( alpha=0.75 ) +
  geom_hline( yintercept = 0 ) +
  theme_minimal()
```



We now see quite clearly that as `alpha` grows, linear regression gets more biased if cluster size relates to average impact in the cluster (`size_coef = 0.2`). Our finding makes sense given our theoretical understanding of the problem—if size is not related to treatment effect, it is hard to imagine how varying cluster sizes would cause much bias.

We are looking at an interaction between our simulation factors: we only see bias for linear regression when cluster size relates to impact and there is variation in cluster size. We also see that all the estimators have near zero bias when there is no variation in cluster size or the cluster size does not relate to outcome, as shown by the top row and left column of facets. Finally, we see the methods all likely give the same answers when there is no cluster size variation, given the overplotted lines on the left column of the figure.

So far we have learned that MLM does seem to react to ICC, and that LR reacts to `alpha` and `size_coef` in combination. We can continue to explore by changing our x-axis, e.g., to ICC. More broadly, with many levels of a factor, as we have with ICC, we can let ggplot aggregate directly by taking advantage of `geom_smooth()`. This leads to the following:



Our story is fairly clear now: LR is increasingly biased as alpha grows large and the cluster size relates to impact. MLM is also biased in these cases, but only when ICC is low (this is because it is driving towards person-weighting when there is little cluster variation).

Our figure does have what appears to be a negative bias trend for `alpha = 0.5` and higher `ICC`—we expect this is Monte Carlo error, as we do not have a theoretical reason to expect bias here; simulations will often have unfortunate, and misleading, patterns that are driven by random chance. We could (and probably should) run further analyses or a second simulation to check our assumptions here.

Aggregation is powerful, but it can be misleading 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 different sample sizes, 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 does not change massively over the marginalized-out factors. To achieve this, we can often average over a relative measure (such as standard error divided by the standard error of some baseline method), which tend to be more invariant and comparable across scenarios. We will see more examples of this kind of aggregation later on.

13.3.0.1 Some notes on how to aggregate

Some performance measures are biased with respect to the Monte Carlo uncertainty. The estimated standard error, for example, is biased; the variance, by contrast, is not. The RMSE is biased, the MSE is not.

When aggregating, therefore, it is often best to aggregate the unbiased performance measures, and then calculate the biased ones from those. For example, to estimate aggregated standard error you might do the following:

```
agg_perf <- sres %>%
  group_by( ICC, method, alpha, size_coef ) %>%
  summarise( SE = sqrt( mean( SE^2 ) ) )
```

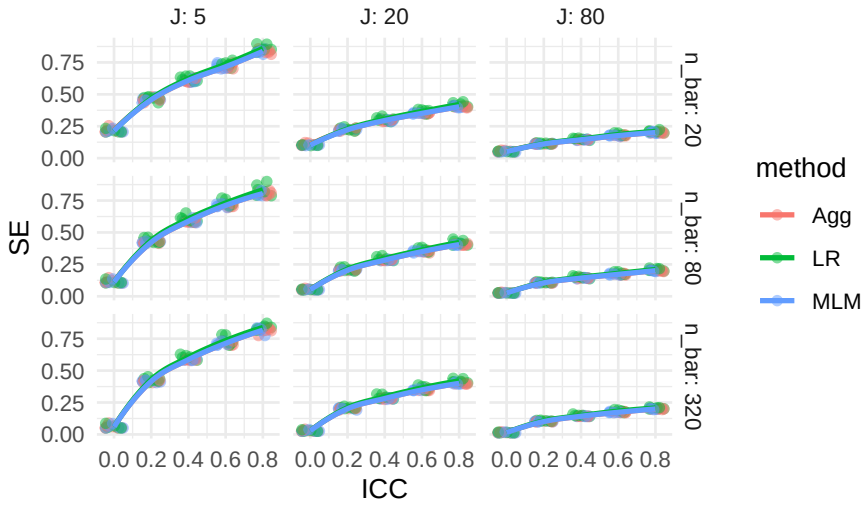
MCSE can sometimes distort an aggregation of a performance measure. For example, if you are looking at the magnitude of bias ($|bias|$), then you can run into issues when the biases are close to zero, if they are measured noisily. In particular, imagine you have two scenarios with true bias of 0.0, but your MCSE is 0.02. In one scenario, you estimate a bias of 0.017, and in the other -0.023. If you average the estimated biases, you get -0.003, which suggests a small bias as we would wish. Averaging the absolute biases, on the other hand, gives you 0.02, which could be deceptive. With high MCSE and small magnitudes of bias, looking at average bias, not average $|bias|$, is safer.

Alternatively, you can use the formula $RMSE^2 = Bias^2 + SE^2$ to back out the average absolute bias from the RMSE and SE. In this approach, first aggregate to get overall RMSE and SE, and then calculate $Bias^2$ as $RMSE^2 - SE^2$.

13.4 Comparing true SEs with standardization

We just did a deep dive into bias. Uncertainty (standard errors) is another primary performance criterion of interest. As an initial exploration, we plot the standard error estimates from our Cluster RCT simulation, using smoothed lines to visualize trends. We use `ggplot`'s `geom_smooth` to aggregate over `size_coef` and `alpha`, which we leave out of the plot. We include individual data points to visualize variation around the smoothed estimates:

```
ggplot( sres, aes( ICC, SE, col=method ) ) +
  facet_grid( n_bar ~  J, labeller = label_both ) +
  geom_jitter( height = 0, width = 0.05, alpha=0.5 ) +
  geom_smooth( se=FALSE, alpha=0.5 ) +
  theme_minimal()
```

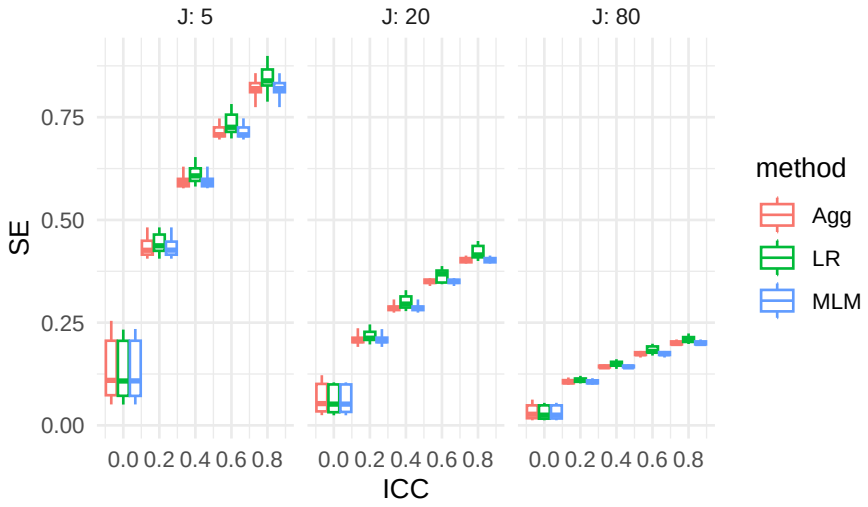


We observe several broad trends in the standard error behavior. First, standard error increases with the intraclass correlation (ICC). This is as expected: greater similarity within clusters reduces the effective sample size. Second, standard error decreases as the number of clusters (J) increases, which reflects the benefit of having more independent units of analysis. In contrast, increasing the cluster size (n_bar) has relatively little effect on the standard error (the rows of our facets look about the same). Lastly, all methods show fairly similar levels of standard error overall.

While we can extract all of these observations from the figure, the figure is not ideal for *comparing* our methods. The dominant influence of design features like ICC and sample size obscures our ability to detect meaningful differences between methods. In other words, even though SE changes across scenarios, it's difficult to tell which method is actually performing better within each scenario.

We can also view the same information by bundling over the left-out dimensions. We add `n_bar` to our bundles, instead of faceting by it, because maybe it does not matter that much:

```
ggplot( sres, aes( ICC, SE, col=method, group=paste0( ICC, method ) ) ) +
  facet_grid( . ~  J, labeller = label_both ) +
  geom_boxplot( width = 0.2, coef = Inf) +
  scale_x_continuous( breaks = unique( sres$ICC)) +
  theme_minimal()
```



Our plots are still dominated by the strong effects of ICC and the number of clusters (J). When performance metrics like standard error vary systematically with design features, it becomes difficult to compare methods meaningfully across scenarios.

To address this, we shift our focus through standardization. We want to shift from observations such as:

> "All my SEs are getting smaller,"

to conclusions such as:

> "Estimator 1 has systematically higher SEs than Estimator 2 across scenarios."

Simulation results are often driven by broad design effects, which can obscure the specific methodological questions we care about. We can therefore standardize, by calculating a relative performance measure, to bring our desired comparisons to the forefront. One straightforward strategy for standardization is to compare each method's performance to a designated baseline. In this example, we use Linear Regression (LR) as our baseline.

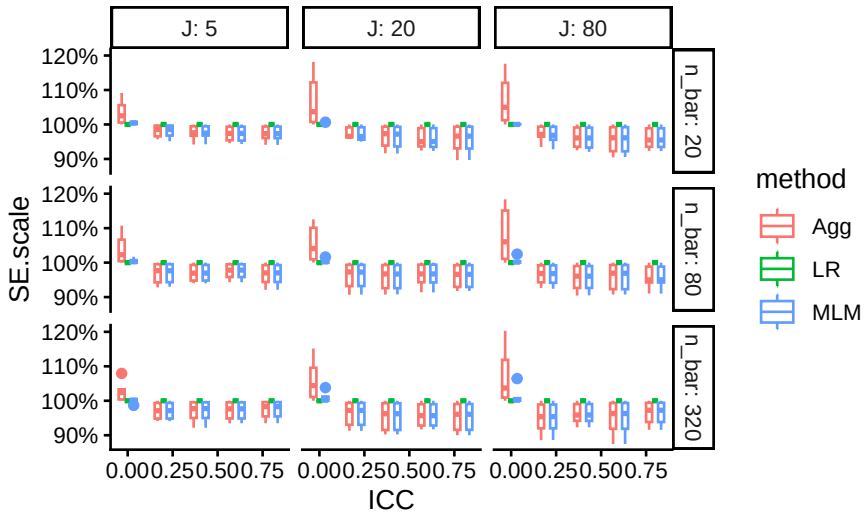
We standardize by, for each simulation scenario, dividing each method's SE by the SE of LR, to produce `SE.scale`. This relative measure, `SE.scale`, allows us to examine how much better or worse, across our scenarios, each method performs relative to a chosen reference method.

```
ssres <-
  sres %>%
  group_by( n_bar, J, ATE, size_coef, ICC, alpha ) %>%
  mutate( SE.scale = SE / SE[method=="LR"]) %>%
```

```
  ungroup()
```

We can then treat `SE.scale` as a measure like any other. Here we bundle, showing how relative SE changes by J, `n_bar` and ICC:

```
ggplot( ssres, aes( ICC, SE.scale, col=method,
                    group = interaction(ICC, method) ) ) +
  facet_grid( n_bar ~  J, labeller = label_both ) +
  geom_boxplot( position="dodge", width=0.1 ) +
  scale_y_continuous( labels = scales::percent_format() )
```

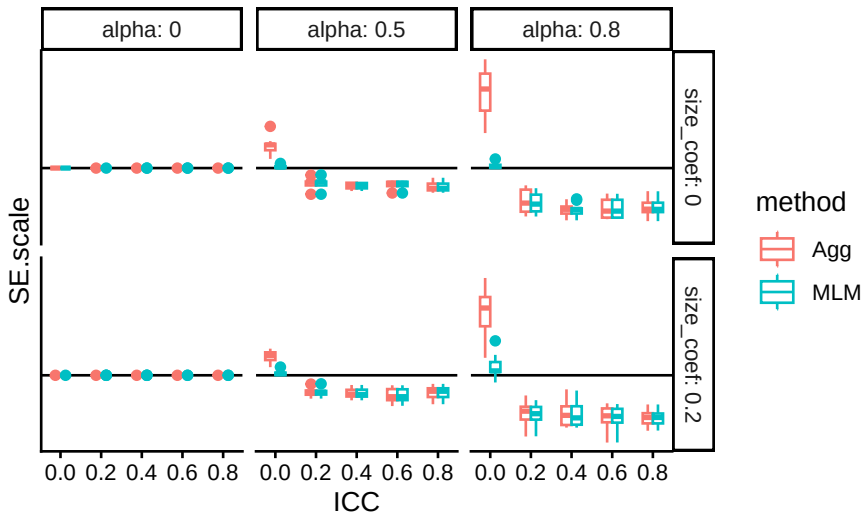


The figure above shows how each method compares to LR across simulation scenarios. Aggregation clearly performs worse than LR when the Intraclass Correlation Coefficient (ICC) is zero. However, when the ICC is greater than zero, Aggregation yields improved precision. The Multilevel Model (MLM), in contrast, appears more adaptive. It captures the benefits of aggregation when the ICC is high, but avoids the precision cost when ICC is zero. This adaptability makes MLM appealing in practice when the ICC is unknown or variable across contexts.

In looking at the plot we are seeing essentially identical rows and fairly similar columns. This suggests we should bundle the `n_bar` to get a cleaner view of the main patterns, and that we could also bundle over `J` as well. We finally can drop the `LR` results entirely, as it is the reference method and always has a relative SE of 1.

```
ssres %>%
  filter( method != "LR" ) %>%
```

```
  ggplot( aes( ICC, SE.scale, col=method,
               group = interaction(ICC, method) ) ) +
  facet_grid( size_coef ~ alpha, labeller = label_both ) +
  geom_hline( yintercept = 1  ) +
  geom_boxplot( position="dodge", width=0.1 ) +
  scale_x_continuous( breaks=unique( ssres$ICC ) ) +
  scale_y_continuous( breaks=seq(90, 125, by=5 ) )
```



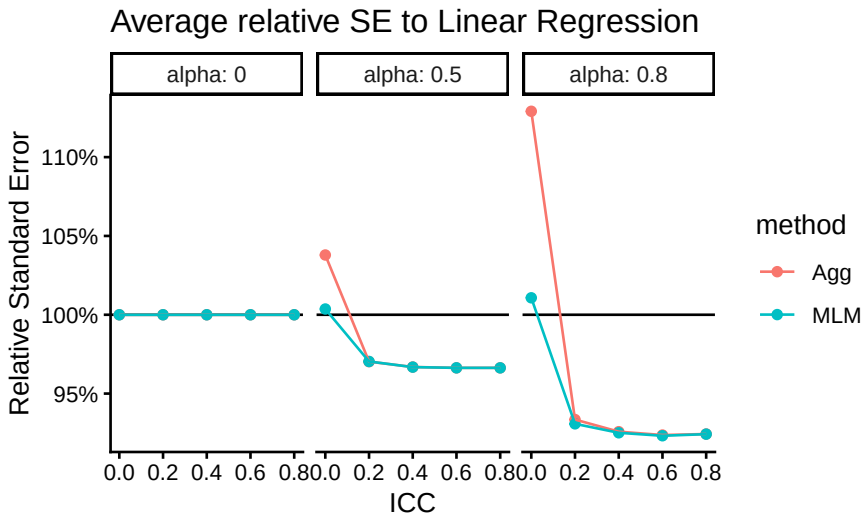
The pattern is clear: when ICC = 0, Aggregation performs worse than LR, and MLM performs about the same. But as ICC increases, Aggregation and MLM both improve, and perform about the same to each other. This highlights the robustness of MLM across diverse conditions.

As a warning regarding Monte Carlo uncertainty: when standardizing results, it is important to remember that uncertainty in the baseline measure (here, LR) propagates to the standardized values. This should be considered when interpreting variability in the scaled results. Uncertainty for relative performance is generally tricky to assess.

To clarify the main patterns, we then aggregate our SE.scale across the bundled simulation settings—relative performance is on the same scale, so averaging is now a natural thing to do. We have aggregated out sample sizes, and we go further and remove `size_coef` since it does not seem to matter much, given the above plot:

```
s2 <-
  ssres %>%
  group_by( ICC, alpha, method ) %>%
```

```
  summarise( SE.scale = mean( SE.scale ) ) %>%
  filter( method != "LR" )

ggplot( s2, aes( ICC, SE.scale, col=method ) ) +
  facet_wrap( ~ alpha, labeller = label_both ) +
  geom_hline( yintercept = 1 ) +
  geom_point() + geom_line() +
  scale_y_continuous( labels = scales::percent_format() ) +
  labs( title = "Average relative SE to Linear Regression",
        y = "Relative Standard Error" )
```



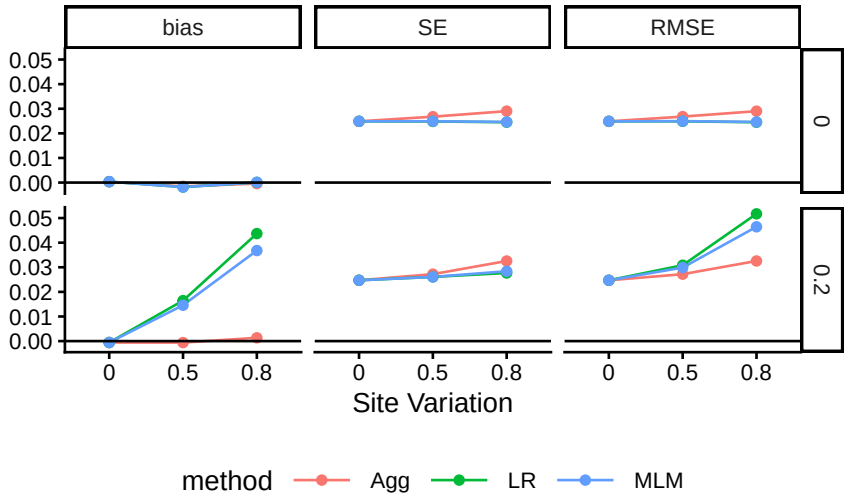
Our aggregated plot of the precision of aggregation and MLM relative to Linear Regression gives a simple story clearly told. The performance of aggregation improves with ICC. MLM also has benefits over LR, and does not pay much cost when ICC is low.

13.5 The Bias-SE-RMSE plot

We can visualize bias and standard error together, along with RMSE, to get a rich picture of performance. To illustrate, we subset to our scenarios where there is real bias for both LR and MLM (i.e., when ICC is 0; see findings under bias from above). We also subset to our middle values of `n_bar = 80` and our large `J=80`, where uncertainty is small and thus the relative role of

bias may be large.

```
bsr <- sres %>%
  filter( n_bar == 80, J==80, ICC == 0 )

bsr <- bsr %>%
  dplyr::select( -R, -power, -ESE_hat, -SD_SE_hat ) %>%
  pivot_longer( cols=c("bias","SE","RMSE"),
                names_to = "measure",
                values_to = "value" ) %>%
  group_by( measure, size_coef, method, ICC, alpha ) %>%
  summarise( value = mean( value ),
             n = n() )

bsr$measure = factor( bsr$measure,
                      levels=c("bias", "SE", "RMSE"),
                      labels =c("bias", "SE", "RMSE" ) )

ggplot( bsr, aes( as.factor(alpha), value, col=method,
                  group = method )) +
  facet_grid( size_coef ~ measure ) +
  geom_line() + geom_point() +
  labs( y = "", x = "Site Variation" )  +
  geom_hline( yintercept = 0 ) +
  theme( legend.position="bottom",
         legend.direction="horizontal",
         legend.key.width=unit(1,"cm"),
         panel.border = element_blank() )
```



The combination of bias, standard error, and RMSE provides a rich and informative view of estimator performance. The top row represents settings where effect size is independent of cluster size, while the bottom row reflects a correlation between size and effect. We see how bias, SE and RMSE grow as site variation increases (moving rightward in each panel). Notably, when effect size is related to cluster size (bottom row), both linear regression and MLM exhibit significant bias, leading to notable increase in RMSE over SE. In contrast, when effect size is unrelated to cluster size (top row), all methods show minimal bias, and the SEs are about the same; that said, we see aggregation paying a penalty as variation cluster size increases. Overall, we see RMSE is primarily driven by SE.

The Bias-SE-RMSE visualization directly illustrates the canonical relationship:

$$\text{RMSE}^2 = \text{Bias}^2 + \text{SE}^2$$

The plot shows overall performance (RMSE) decomposed into into its two fundamental components: systematic error (bias) and variability (standard error). Here we see how bias for LR, for example, is dominant when site variation is high. The differences in SE across methods are small and are thus not the main reason for differences in overall estimator performance; bias is the main driver.

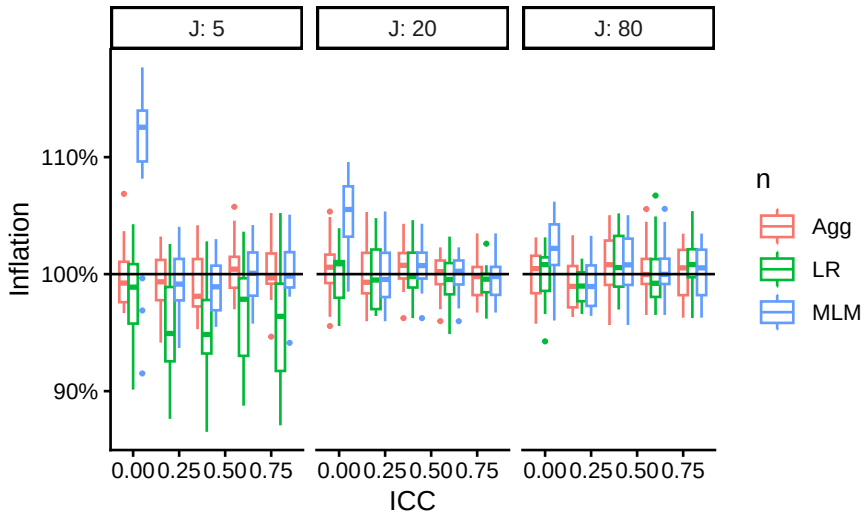
This is the kind of diagnostic plot we often wish were included in more applied simulation studies.

13.6 Assessing the quality of the estimated SEs

So far we have examined the performance of our *point estimators*. We next look at ways to assess our *estimated* standard errors. A good first question is whether they are *calibrated* (about the right size on average) across all the scenarios.

When assessing estimated standard errors it is very important to see if they are *reliably* the right size across the scenarios, making bundling an especially important tool here. We first see if the average estimated SE, relative to the true SE, is usually around 1 across all scenarios:

```
sres <- sres %>%
  mutate( inflate = ESE_hat / SE  )

ggplot( sres,
        aes( ICC, inflate, col=method,
             group = interaction(ICC,method) ) ) +
    facet_grid( . ~ J, labeller = label_both) +
  geom_boxplot( position="dodge", outlier.size=0.5 ) +
  geom_hline( yintercept=1 ) +
  labs( color="n", y = "Inflation" ) +
  scale_y_continuous( labels = scales::percent_format() )
```

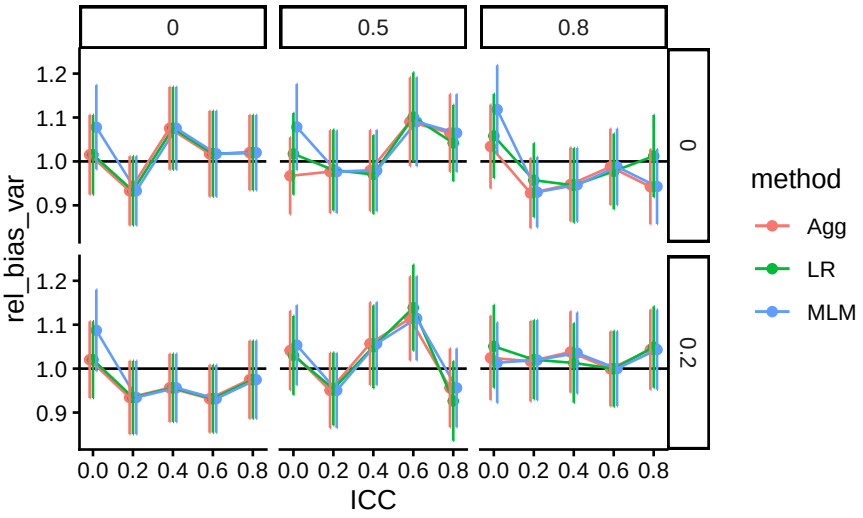


We see that, for the most part, our estimated SEs are about right, on average, across all scenarios. When the ICC is 0 and J is small, the MLM SEs are

clearly too high. We also see that when J is 5, the LR estimator tends to be a bit low.

We next start exploring to dig into why our boxplots are wide. In particular, we want to see if other factors dictate when the SEs are biased. We first subset to the $J = 80$ scenarios to see if those box widths could just be due to the MCSEs. The `simhelpers` `calc_relative_var()` method gives mcses for relative bias of an estimated *variance* to the true *variance*. We thus square our estimated SEs to get variance estimates, and then use that function to see if the relative variance estimates are biased:

```
se_res <- res %>%
  group_by( n_bar, J, ATE, size_coef, ICC, alpha, method ) %>%
  summarize( calc_relative_var( estimates = ATE_hat,
                                var_estimates = SE_hat^2,
                                criteria = "relative bias" ) )

se_res %>%
  filter( J == 80, n_bar == 80 ) %>%
  ggplot( aes( ICC, rel_bias_var, col=method ) ) +
  facet_grid( size_coef ~ alpha  ) +
  geom_hline( yintercept = 1 ) +
  geom_point( position = position_dodge( width=0.05) ) +
  geom_line( position = position_dodge( width=0.05) ) +
  geom_errorbar( aes( ymin = rel_bias_var - 1.96*rel_bias_var_mcse,
                      ymax = rel_bias_var + 1.96*rel_bias_var_mcse ),
                 position = position_dodge( width=0.05 ),
                 width = 0 )
```

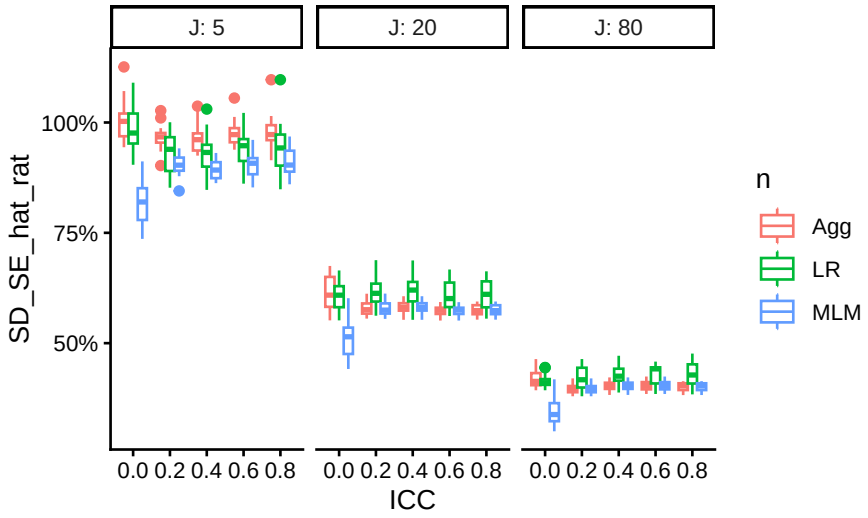


In looking at this plot, we see no real evidence of miscalibration: our confidence intervals are generally covering 1, meaning our average estimated variance is about the same as the true variance. This makes us think the boxes for $J = 80$ in the prior plot are wide due to MCSE rather than other simulation factors driving some slight miscalibration. We might then assume this applies to the $J = 20$ case as well.

13.6.1 Stability of estimated SEs

We can also look at how stable the estimated SEs are, relative to the actual uncertainty they are trying to capture. We do this by calculating the standard deviation of the estimated standard errors and compare that to the standard deviation of the point estimate. This is related to the coefficient of variation of `SE_hat`.

```
sres <- mutate( sres,
                SD_SE_hat_rat = SD_SE_hat / SE )

ggplot( sres,
        aes( ICC, SD_SE_hat_rat, col=method,
             group = interaction(ICC,method) ) ) +
    facet_grid( . ~ J, labeller = label_both) +
  geom_boxplot( position="dodge" ) +
  labs( color="n" ) +
  scale_y_continuous( labels = scales::percent_format() ) +
  scale_x_continuous( breaks = unique( sres$ICC ) )
```



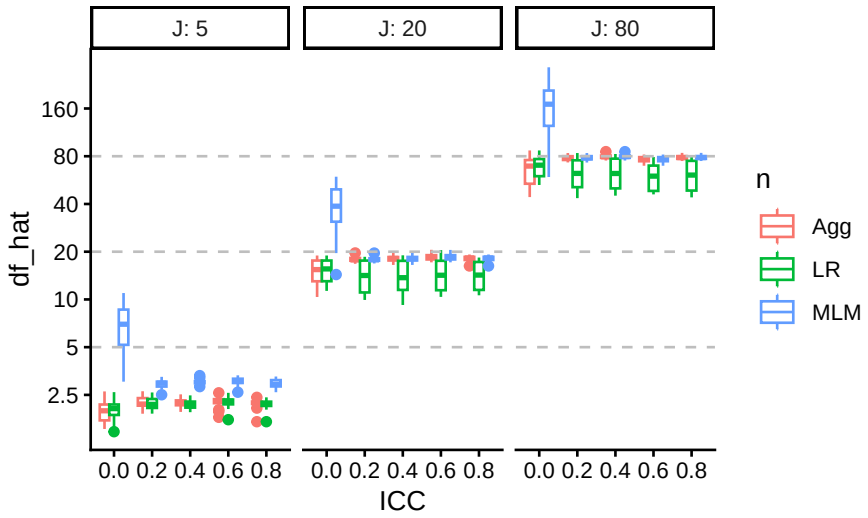
Overall, we have a lot of variation in the estimated SEs, relative to the actual uncertainty. We also see that MLM has more reliably estimated SEs than other methods when ICC is small. Aggregation has relatively more trouble estimating uncertainty when J is small. Finally, LR's SEs are slightly more unstable, relative to the other methods, when J is larger.

Assessing the stability of standard errors is usually very in the weeds of a performance evaluation. It is a tricky measure: if the true SE is high for a method, then the relative instability will be lower, even if the absolute instability is the same. Thinking through what might be driving what can be difficult, and is often not central to the main purpose of an evaluation. People often look at confidence interval coverage and confidence interval width, instead, to assess the quality of estimated SEs.

We can instead try using the degree of freedom calculations we saw in Section 9.2.2. We plug in $\widehat{df} = \frac{2\bar{V}^2}{S_V^2}$ for each scenario and plot:

```
sres <- mutate( sres,
                df_hat = 2 * ( ESE_hat^4 ) / ( SD_SE_hat^4 ) )
ggplot( sres,
        aes( ICC, df_hat, col=method,
             group = interaction(ICC,method) ) ) +
    facet_grid( . ~ J, labeller = label_both) +
  geom_boxplot( position="dodge" ) +
  labs( color="n" ) +
  scale_y_log10( breaks = c( 2.5, 5, 10, 20, 40, 80, 160 ),
                 labels = c( 2.5, 5, 10, 20, 40, 80, 160 )
                 ) +
  geom_hline( yintercept = c(5,20,80), lty=2, col="grey" ) +
```

```
  scale_x_continuous( breaks = unique( sres$ICC ) )
```



The multilevel model generally has higher estimated degrees of freedom, especially when ICC is low, and Linear Regression has generally lower degrees of freedom overall (except when ICC is zero).

13.7 Assessing confidence intervals

Coverage is a blend of how accurate (unbiased) our estimates are and how good our estimated SEs are. To assess coverage, we first calculate confidence intervals using the estimated effect, estimated standard error, and degrees of freedom. Once we have our calculated t-based intervals, we can average them across runs to get average width and coverage using `simhelpers`'s `calc_coverage()` method. A good confidence interval estimator would be one which is generally relatively short while maintaining proper coverage.

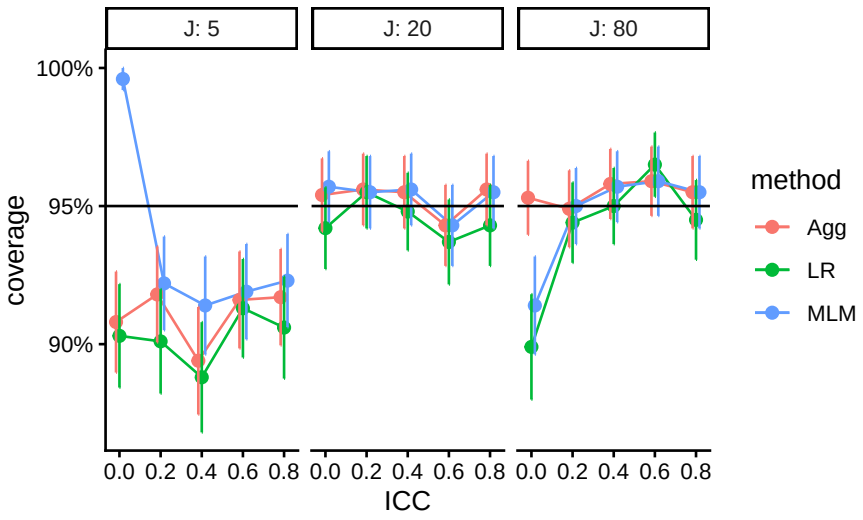
Our calculations are as so:

```
res$df = res$J
res <- mutate( res,
               tstar = qt( 0.975, df=df ),
               CI_low = ATE_hat - tstar*SE_hat,
               CI_high = ATE_hat + tstar*SE_hat,
               width = CI_high - CI_low )
```

```
covres <- res %>%
  group_by( n_bar, J, ICC, alpha, size_coef, method, ATE ) %>%
  summarise( calc_coverage( lower_bound = CI_low,
                            upper_bound = CI_high,
                            true_param = ATE ) ) %>%
  ungroup()
```

We then look at those simulations with a relationship of site size and impact. We subset to `n_bar = 80`, `size_coef = 0.2` and `alpha = 0.5` to simplify for our initial plot:

```
c_sub <- covres %>%
  dplyr::filter( size_coef != 0, n_bar == 80, alpha == 0.5 )

ggplot( c_sub, aes( ICC, coverage, col=method, group=method ) ) +
  facet_grid( . ~ J, labeller = label_both ) +
  geom_line( position = position_dodge( width=0.05)) +
  geom_point( position = position_dodge( width=0.05), size = 2 ) +
  geom_errorbar( aes( ymax = coverage + 2*coverage_mcse,
                      ymin = coverage - 2*coverage_mcse ), width=0,
                 position = position_dodge( width=0.05) ) +
  geom_hline( yintercept = 0.95 ) +
  scale_y_continuous( labels = scales::percent_format() )
```

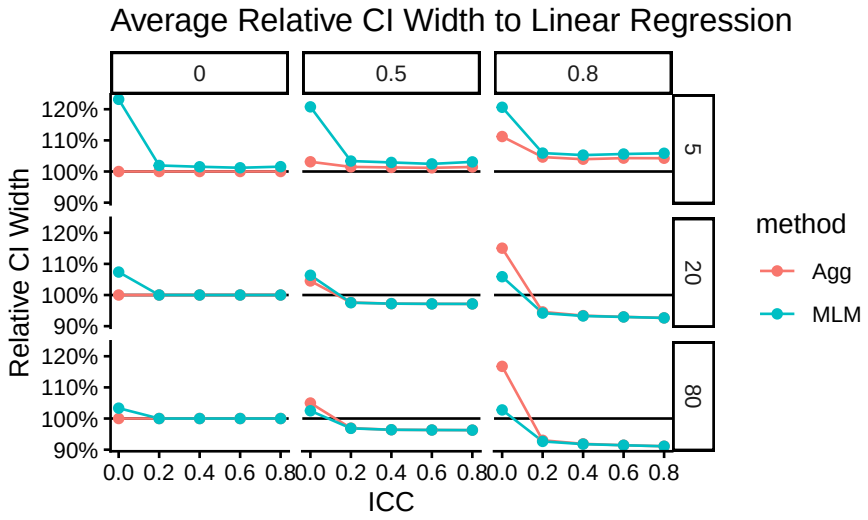


Generally coverage is good unless J is low or ICC is 0. Monte Carlo standard error based confidence intervals on our performance metrics indicate that, in some settings, the observed coverage is reliably different from the nominal 95%,

suggesting issues with estimator bias, standard error estimation, or both. We might then want to see if these results are general across the other simulation scenarios (see exercises).

For confidence interval width, we can calculate the average width relative to the width of LR across all scenarios:

```
covres <- covres %>%
  group_by( n_bar, J, ICC, alpha, size_coef ) %>%
  mutate( width_rel = width / width[method=="LR"] ) %>%
  ungroup()

c_agg <- covres %>%
  group_by( ICC, J, alpha, method ) %>%
  summarise( coverage = mean( coverage ),
             coverage_mcse = sqrt( mean( coverage_mcse^2 ) ) / n(),
             width_rel = mean( width_rel ) ) %>%
  filter( method != "LR" )

ggplot( c_agg, aes( ICC, width_rel, col=method, group=method ) ) +
  facet_grid( J ~ alpha ) +
  geom_hline( yintercept = 1 ) +
  geom_line() + geom_point() +
  labs( y = "Relative CI Width",
        title = "Average Relative CI Width to Linear Regression" ) +
  scale_y_continuous( labels = scales::percent_format() )
```



Confidence interval width serves as a proxy for precision. Narrow intervals

suggest more precise estimates. We see MLM has wider intervals, relative to LR, when ICC is low. When there is site variation, both Agg and MLM have shorter intervals. This plot essentially echos our standard error findings, as expected. There are mild differences due to differences in how the degrees of freedom are calculated, however.

13.8 Conclusions

Visualization is rarely a single plot; it is the process of iterating from messy, exploratory graphs to a small set of figures that communicate the main findings. In this chapter we walked through four core tools for that process—subsetting, many small multiples, bundling, and aggregation—and showed how they can be used together to manage the complexity of multifactor simulations. Subsetting helps with orientation and sanity checks, small multiples reveal interactions, bundling summarizes variability across scenarios, and aggregation highlights broad trends when the raw results are too noisy or too high dimensional to interpret directly.

We illustrated these tools using the Cluster RCT case study, beginning with bias and then extending the same workflow to standard errors, RMSE, estimated standard errors, and confidence interval performance. A recurring theme was that raw performance measures are often dominated by design effects such as ICC and sample size, which can obscure the method comparisons we care about. Standardization—especially expressing performance relative to a baseline method—can make comparisons far clearer, though it also propagates uncertainty from the baseline and should be interpreted with care.

Ultimately, the goal of visualization is to discover and communicate structure: where is performance stable and when does it change? The plots in this chapter provide a template for an overall workflow, and we encourage you to use the code and approach in your own work. There are further tools, and further concerns, that build on this core set of techniques; that is what we discuss in the following chapter.

Exercises

In these exercises we continue to work with the simulation results from the Cluster RCT example in this chapter. We provide the saved simulation results in the file `simulation_CRT.rds`, and load and process them as follows:

```
library( tidyverse )
library( simhelpers )

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

```
# A tibble: 6 x 16
  n_bar     J   ATE size_coef   ICC alpha   seed runID method
  <dbl> <dbl> <dbl>     <dbl> <dbl> <dbl>  <dbl> <int> <chr>
1   320     5   0.2       0.2   0.8   0.5 103553     1 MLM
2   320     5   0.2       0.2   0.8   0.5 103553     1 LR
3   320     5   0.2       0.2   0.8   0.5 103553     1 Agg
4   320     5   0.2       0.2   0.8   0.5 103553     2 MLM
5   320     5   0.2       0.2   0.8   0.5 103553     2 LR
6   320     5   0.2       0.2   0.8   0.5 103553     2 Agg
# i 7 more variables: ATE_hat <dbl>, SE_hat <dbl>,
#   p_value <dbl>, message <dbl>, warning <dbl>, error <dbl>,
#   seconds <dbl>
```

```
sres <- res %>%
  group_by( n_bar, J, ATE, size_coef, ICC, alpha, method ) %>%
  summarise( calc_absolute( estimates = ATE_hat,
                            true_param = ATE,
                            criteria = c("bias","stddev",
                                         "rmse")),
             calc_relative_var( estimates = ATE_hat,
                                var_estimates = SE_hat^2,
                                criteria = "relative bias" ),
             power = mean( p_value <= 0.05 ),
             ESE_hat = sqrt( mean( SE_hat^2 ) ),
             SD_SE_hat = sqrt( sd( SE_hat^2 ) ),
             .groups = "drop"
  ) %>%
  rename( R = K_absolute,
          RMSE = rmse,
          RMSE_mcse = rmse_mcse,
          SE = stddev,
          SE_mcse = stddev_mcse ) %>%
  dplyr::select(  -K_relvar ) %>%
  ungroup()
names( sres )
```

```
 [1] "n_bar"            "J"                "ATE"
 [4] "size_coef"        "ICC"              "alpha"
 [7] "method"           "R"                "bias"
[10] "bias_mcse"        "SE"               "SE_mcse"
```

```
[13] "RMSE"              "RMSE_mcse"         "rel_bias_var"
[16] "rel_bias_var_mcse" "power"             "ESE_hat"
[19] "SD_SE_hat"
```

Exercise 13.1 (Checking bias)**.** Under the discussion of aggregation, we closed with a plot suggesting negative bias as ICC increases when `size_coef=0`. Investigate that finding, either by digging into the results more or by running additional simulations. *Do not* re-run the entire simulation; sometimes a targeted investigation can be just as effective for answering a specific question.

Exercise 13.2 (Assessing uncertainty)**.** Select a plot from this chapter that does not have monte carlo standard errors. Add monte carlo standard errors to that selected plot.

Exercise 13.3 (Assessing power)**.** Make a plot showing how power changes as J changes. As an extra step, add MCSEs to the plot to assess whether you have a reasonable numbers of replications in the simulation.

Exercise 13.4 (Going deeper with coverage)**.** For our cluster RCT, we saw coverage is low in some circumstances. Explore the full set of simulation results, possibly adding your own analyses, to ascertain why coverage is low. Is it due to estimator bias? Unreliably estimated standard errors? Something else? Make a final plot and draft a short write-up that captures how coverage is vulnerable for the three estimators.

Exercise 13.5 (Making another plot for assessing SEs)**.** In the main chapter we examined how SE changes as a function of various simulation factors. Now generate a plot to see whether and when cluster size meaningfully helps precision, and explain what you find.

Exercise 13.6 (Pearson correlations with a bivariate Poisson distribution)**.** In Section 10.3, we generated results for a multifactor simulation of confidence intervals for Pearson's correlation coefficient under a bivariate Poisson data-generating process. Create a plot that depicts the coverage rate of the confidence intervals for ρ across all four simulation factors. Write a brief explanation of how the plot is laid out and explain why you chose to construct it as you did.

14

Special Topics on Reporting Simulation Results

In this chapter we cover a few special topics on reporting simulation results. We first walk through some examples of how to do regression modeling, including using regression trees to identify what factor combinations are most important for explaining a set of results. We then dive more deeply into what to do when you have only a few iterations per scenario, and then we discuss what to do when you are evaluating methods that sometimes fail to converge or give an answer.

14.1 Using regression to analyze simulation results

Chapter 12 provided some examples of using regression and ANOVA on a set of simulation results to summarize overall patterns across scenarios. In this chapter we will provide some further in-depth examples along with the R code for doing this sort of thing.

14.1.1 Example 1: Biserial, revisited

As our first in depth example, we walk through the analysis that produces the final ANOVA summary table for the biserial correlation example in Chapter 12. In the visualization there, we saw that several factors impacted bias. The eta table presented later in that same chapter then decomposed the variance in performance across several factors so we could see which simulation factors mattered most for bias.

To build that table, we first fit a regression model, regressing bias on all the simulation factors. We first convert each factor to a factor variable, so that R does not assume a continuous relationship for numeric values.

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

```
  tidy() %>%
  knitr::kable( digits = 4 )
```

term	estimate	std.error	statistic	p.value
(Intercept)	0.0014	0.0002	8.6793	0.0000
fixedSample cut-off	-0.0036	0.0001	-34.7969	0.0000
rho	-0.0094	0.0007	-12.6188	0.0000
I(rho^2)	0.0072	0.0008	9.4772	0.0000
p1.L	0.0046	0.0001	39.5291	0.0000
p1.Q	-0.0016	0.0001	-13.7454	0.0000
p1.C	0.0008	0.0001	6.9747	0.0000
p1^4	-0.0001	0.0001	-0.9581	0.3381
n	0.0000	0.0000	20.3964	0.0000

The above printout gives main effects for each factor, averaged across the others. Because `p1` and `n` are ordered factors, the `lm()` command automatically generates linear, quadratic, cubic and fourth order contrasts for them. We smooth our `rho` factor, which has many levels of a continuous measure, with a quadratic curve. We could instead use splines or some local linear regression if we were worried about model fit for a complex relationship.

The main effects are summaries of trends across contexts. For example, averaged across the other contexts, the "sample cutoff" condition is around 0.004 lower than the population (the baseline condition).

As shown in Chapter 12, we use ANOVA to get a sense of the major sources of variation in the simulation results (e.g., identifying which factors have negligible/minor influence on the bias of an estimator). To do this, we use `aov()` to fit an analysis of variance model:

```
anova_table <- aov(bias ~ rho * p1 * fixed * n, data = r_F)
knitr::kable( summary(anova_table)[[1]],
              digits = c(0,4,4,1,5) )
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
rho	1	0.0024	0.0024	280.8	0.00000
p1	4	0.0236	0.0059	677.3	0.00000
fixed	1	0.0159	0.0159	1821.4	0.00000
n	1	0.0054	0.0054	625.8	0.00000
rho:p1	4	0.0017	0.0004	49.5	0.00000
rho:fixed	1	0.0034	0.0034	395.1	0.00000
p1:fixed	4	0.0017	0.0004	48.3	0.00000
rho:n	1	0.0007	0.0007	83.1	0.00000
p1:n	4	0.0073	0.0018	209.4	0.00000

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
fixed:n	1	0.0049	0.0049	561.4	0.00000
rho:p1:fixed	4	0.0005	0.0001	13.6	0.00000
rho:p1:n	4	0.0005	0.0001	14.8	0.00000
rho:fixed:n	1	0.0011	0.0011	122.4	0.00000
p1:fixed:n	4	0.0005	0.0001	15.1	0.00000
rho:p1:fixed:n	4	0.0002	0.0000	4.3	0.00167
Residuals	4760	0.0414	0.0000	NA	NA

The advantage here is the multiple levels of our categorical factors get bundled together in our table of results, making a tidier display. Note we are now including interactions between our simulation factors. The prior linear regression model was just estimating main effects of the factors, and not estimating these more complex relationships.

The eta table in Chapter 12 is a summary of this anova table, which we generate as follows:

```
library(lsr)
etaSquared(anova_table) %>%
  as.data.frame() %>%
  rownames_to_column("source") %>%
  mutate( order = 1 + str_count(source, ":" ) ) %>%
  group_by( order ) %>%
  arrange( -eta.sq, .by_group = TRUE ) %>%
  relocate( order )
```

We group the results by the order of the interaction, so that we can see the main effects first, then two-way interactions, and so on. We then sort within each group to put the high importance factors first. The resulting variance decomposition table shows the amount of variation explained by each combination of factors.

14.1.2 Biserial correlations, revisited

In the biserial correlation example described in Section 12.2.3, we saw that bias can change notably across scenarios considered, and that several factors appear to be driving these changes. These factors also seem to have complex interactions: note how when p1 = 0.5, we get larger dips than when p1 = 1/8. Figure 12.5 gives a sense of the complex combination of factors that influence bias and also suggests some possibilities for further analysis. In particular, the figure suggests that the bias grows smaller as the sample size increases. Developing a model might let us draw more specific conclusions about how bias relates to sample size.

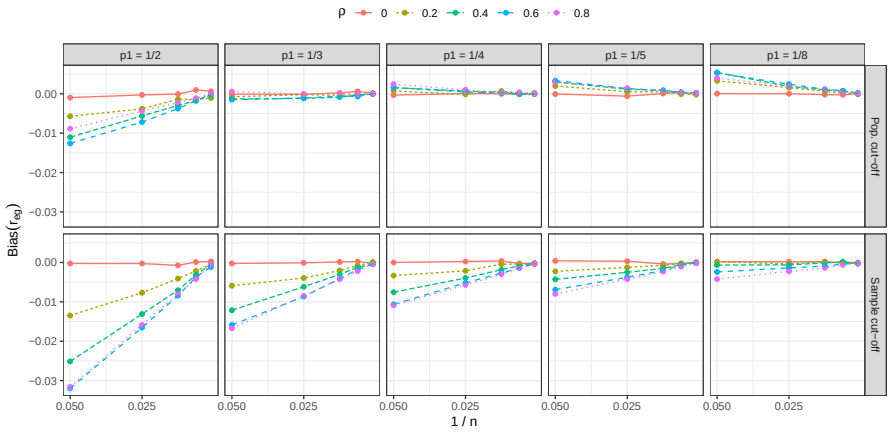


Figure 14.1: Bias of the biserial correlation estimate in an extreme groups design as a function of inverse sample size, cut-off thresholds, and cut-off types, for selected values of ρ.

To get an initial sense of how bias relates to sample size, we created Figure 14.1, which follows a similar layout to Figure 12.5 but shows the inverse sample size on the horizontal axis of each plot, with different values of the correlation parameter corresponding to different colors and line types. For clarity, the plot shows only a small subset of the correlation levels in the full design. Figure 14.1 strongly suggests that the bias is inverse proportional to the sample size. If we let B_{pqrs} be the estimated bias for cut-off level p, cut-off type q, true correlation level r, and sample size level s and let N_s be the actual value of sample size level s, we posit that

$$B_{pqrs} = \beta_{pqr} \times \frac{1}{N_s} + e_{pqrs},$$

where e_{pqrs} is an error capturing the Monte Carlo uncertainty in the bias estimate. If this model is correct, it would allow us to simplify further analysis of the simulation results because it reduces the dimension of the design. Rather than dealing with the 5 distinct levels of the sample size factor, we can reduce it to a single-parameter relationship and focus the remaining analysis on how β_{pqr} varies across the remaining three factors in the design. Furthermore, thinking in terms of a formal model also allows us to test whether the posited relationship provides an adequate fit to the data.

Under the assumptions of the model, we can estimate β_{pqr} simply by calculating $N_s \times B_{pqrs}$ and averaging across values of N_s. Figure 14.2 plots $N_s \times B_{pqrs}$, following the same layout as our original Figure 12.5. Individual points correspond to $N_s \times B_{pqrs}$ for each unique scenario in the simulation; the yellow points are for $N_s = 1000$, for which the Monte Carlo error is much larger

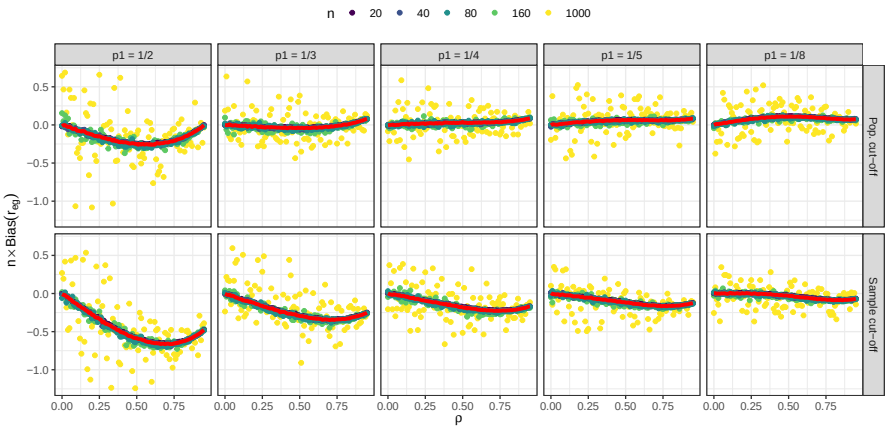


Figure 14.2: N times the bias of the biserial correlation estimate in an extreme groups design as a function of ρ, cut-off thresholds, and cut-off types.

than the other sample sizes. The red lines correspond to the estimated bias coefficients β_{pqr}, calculated by taking weighted averages across sample sizes. It is apparent that the red lines capture the form of the bias quite well, which suggests that we can use our model coefficients for further analysis. For example, we could use ANOVA to understand the influence of the remaining factors on β_{pqr}. We could also create a more succinct summary of the bias relationships by plotting β_{pqr} for varying cut-off values, cut-off types, and correlation parameters, as in Figure 14.3.

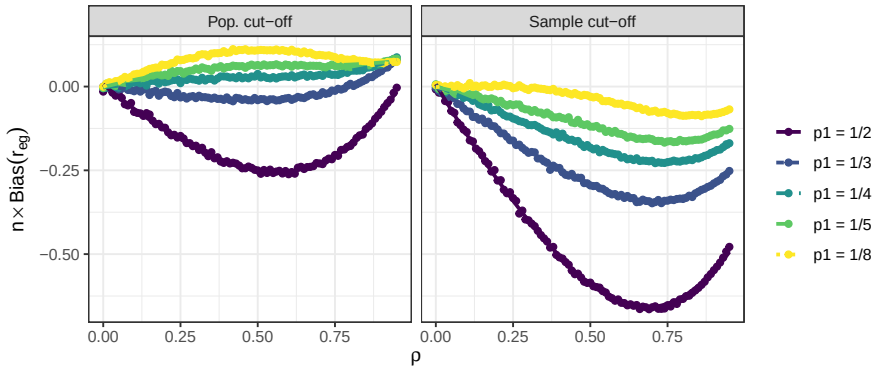


Figure 14.3: N times the bias of the biserial correlation estimate in an extreme groups design as a function of ρ, cut-off thresholds, and cut-off types.

14.1.3 Example 2: Cluster RCT example, revisited

When we have several methods to compare, we can use meta-regression to understand how these methods change as other simulation factors change. We next illustrate this with our running Cluster RCT example.

We again turn our simulation levels (except for ICC, which has several levels) into factors, so R does not assume that sample size, for example, should be treated as continuous:

```
sres_f <-
  sres %>%
  mutate(
    across( c( n_bar, J, size_coef, alpha ), factor ),
    ICC = as.numeric(ICC)
  )

# Run the regression
M <- lm( bias ~ (n_bar + J + size_coef + ICC + alpha) * method,
         data = sres_f )

# View the results
tidy( M ) %>%
  knitr::kable( digits = 3 )
```

term	estimate	std.error	statistic	p.value
(Intercept)	0.002	0.003	0.872	0.384
n_bar80	-0.003	0.002	-1.295	0.196
n_bar320	-0.001	0.002	-0.657	0.511
J20	-0.002	0.002	-0.991	0.322
J80	-0.002	0.002	-0.885	0.376
size_coef0.2	0.003	0.002	1.537	0.125
ICC	0.001	0.003	0.272	0.786
alpha0.5	-0.002	0.002	-1.179	0.239
alpha0.8	0.001	0.002	0.419	0.676
methodLR	-0.012	0.004	-3.060	0.002
methodMLM	0.001	0.004	0.191	0.849
n_bar80:methodLR	0.000	0.003	0.037	0.971
n_bar320:methodLR	0.000	0.003	-0.004	0.997
n_bar80:methodMLM	0.000	0.003	-0.170	0.865
n_bar320:methodMLM	-0.001	0.003	-0.362	0.718
J20:methodLR	0.005	0.003	1.722	0.085
J80:methodLR	0.006	0.003	1.946	0.052
J20:methodMLM	0.001	0.003	0.355	0.723
J80:methodMLM	0.001	0.003	0.420	0.675
size_coef0.2:methodLR	0.016	0.002	6.741	0.000

term	estimate	std.error	statistic	p.value
size_coef0.2:methodMLM	0.003	0.002	1.294	0.196
ICC:methodLR	0.000	0.004	-0.062	0.951
ICC:methodMLM	-0.006	0.004	-1.482	0.139
alpha0.5:methodLR	0.006	0.003	2.210	0.027
alpha0.8:methodLR	0.017	0.003	5.868	0.000
alpha0.5:methodMLM	0.001	0.003	0.434	0.664
alpha0.8:methodMLM	0.003	0.003	1.091	0.276

With even a modestly complex simulation, we can quickly generate a lot of regression coefficients, making our meta-regression somewhat hard to interpret. The above model does not even have interactions between the simulation factors, even though the plots we have seen strongly suggest interactions exist. That said, picking out the significant coefficients is a quick way to obtain clues as to what is driving performance. For instance, several features interact with the LR method for bias. The other two methods seem less impacted.

14.1.3.1 Using LASSO to simplify the model

We can simplify a meta regression model using LASSO regression, to drop coefficients that are less relevant. This requires some work to make our model matrix of dummy variables with all the interactions. If using LASSO, we recommend fitting a separate model to each method being considered; the set of fit LASSO models can then be compared to see which methods react to what factors, and how.

We first illustrate with LR, and then extend to all three. To use the LASSO we have to prepare our data first by hand—this involves converting all our factors to sets of dummy variables for the regression. We also generate all interaction terms up to the cubic level.

```
library(modelr)
library(glmnet)

sres_f_LR <- sres_f %>%
  filter( method == "LR" )

# Create model matrix
form <- bias ~ ( n_bar + J + size_coef + ICC + alpha )^3
X <- model.matrix(form, data = sres_f_LR)[, -1]
#          The [,-1] drops the intercept
dim(X)
```

```
[1] 270  71
```

```
# Fit LASSO
fit <- cv.glmnet(X, sres_f_LR$bias, alpha = 1)

# Non-zero coefficients
coef(fit, s = "lambda.1se") %>%
  as.matrix() %>%
  as.data.frame() %>%
  rownames_to_column("term") %>%
  filter(abs(lambda.1se) > 0) %>%
  knitr::kable(digits = 3)
```

term	lambda.1se
(Intercept)	0.003
size_coef0.2	0.003
size_coef0.2:alpha0.8	0.023

Note we have 71 covariates due to the many, many interactions and the fact that our sample sizes, etc., are all factors, not continuous.

When using regression, and especially LASSO, which levels are baseline can impact the final results. We have our smallest sample sizes, no variation, 0 ICC, and no `size_coef` as baseline. We might imagine that other choices of baseline could suddenly make other factors appear with large coefficients. One trick to avoid selecting a baseline is to give dummy variables for all the factors, and fit LASSO with the colinear terms. Due to regularization, this would still work; we do not pursue this here, however.

We next bundle the above to make three models, one for each method. We first rescale ICC to be on a 5 point scale to control it's relative coefficient size to the dummy variables, and then add a new feature of "zeroICC" as well (recalling the prior plots that showed ICC being 0 was unusual).

```
meth = c( "LR", "MLM", "Agg" )
sres_f$zeroICC = ifelse( sres_f$ICC == 0, 1, 0 )
sres_f$ICCsc = sres_f$ICC * 5 # rescale ICC to be on a 5 point scale

models <- map( meth, function(m) {
  sres_f_LR <- sres_f %>%
    filter( method == m )

  form <- bias ~ ( n_bar + J + size_coef + ICCsc + alpha + zeroICC )^3
  X <- model.matrix(form, data = sres_f_LR)[, -1]
  fit <- cv.glmnet(X, sres_f_LR$bias, alpha = 1)

  coef(fit, s = "lambda.min") %>%
```

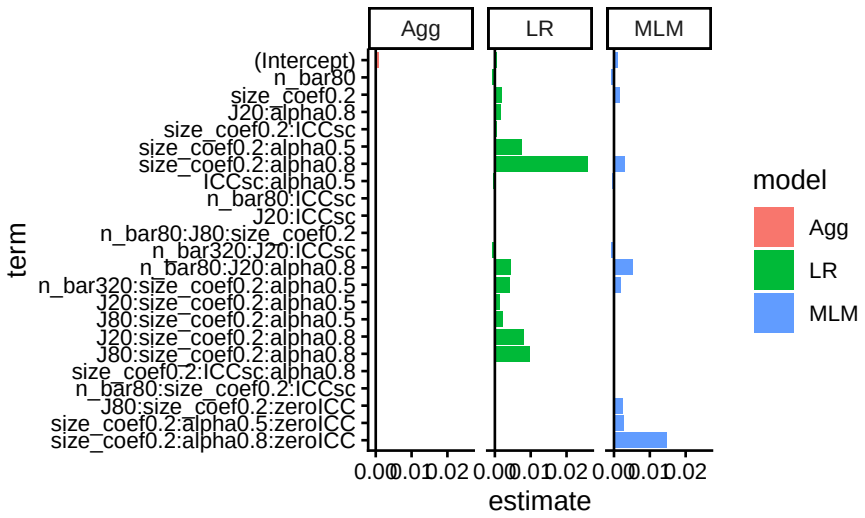
```
    as.matrix() %>%
    as.data.frame() %>%
    rownames_to_column("term") %>%
    rename( estimate = lambda.min ) %>%
    filter(abs(estimate) > 0)
} )

models <-
  models %>%
  set_names(meth) %>%
  bind_rows( .id = "model" )

m_res <- models %>%
  dplyr::select( model, term, estimate ) %>%
  pivot_wider( names_from="model", values_from="estimate" ) %>%
  mutate(order = str_count(term, ":")) %>%
  arrange(order) %>%
  relocate(order)

options(knitr.kable.NA = '')
m_res %>%
  knitr::kable( digits = 3 ) %>%
  print( na.print = "" )
```

order	term	LR	MLM	Agg
0	(Intercept)	0.000	0.001	0.001
0	n_bar80	-0.001	-0.001	
0	size_coef0.2	0.002	0.002	
1	J20:alpha0.8	0.002		
1	size_coef0.2:ICCsc	0.001		
1	size_coef0.2:alpha0.5	0.008		
1	size_coef0.2:alpha0.8	0.026	0.003	
1	ICCsc:alpha0.5	0.000	-0.001	
1	n_bar80:ICCsc		0.000	
1	J20:ICCsc		0.000	
2	n_bar80:J80:size_coef0.2	0.000		
2	n_bar320:J20:ICCsc	-0.001	-0.001	
2	n_bar80:J20:alpha0.8	0.004	0.005	
2	n_bar320:size_coef0.2:alpha0.5	0.004	0.002	
2	J20:size_coef0.2:alpha0.5	0.001		
2	J80:size_coef0.2:alpha0.5	0.002		
2	J20:size_coef0.2:alpha0.8	0.008		

```
|     2|J80:size_coef0.2:alpha0.8      |  0.010|       |       |
|     2|size_coef0.2:ICCsc:alpha0.8    |  0.000|       |       |
|     2|n_bar80:size_coef0.2:ICCsc     |       |  0.000|       |
|     2|J80:size_coef0.2:zeroICC       |       |  0.002|       |
|     2|size_coef0.2:alpha0.5:zeroICC  |       |  0.003|       |
|     2|size_coef0.2:alpha0.8:zeroICC  |       |  0.015|       |
```

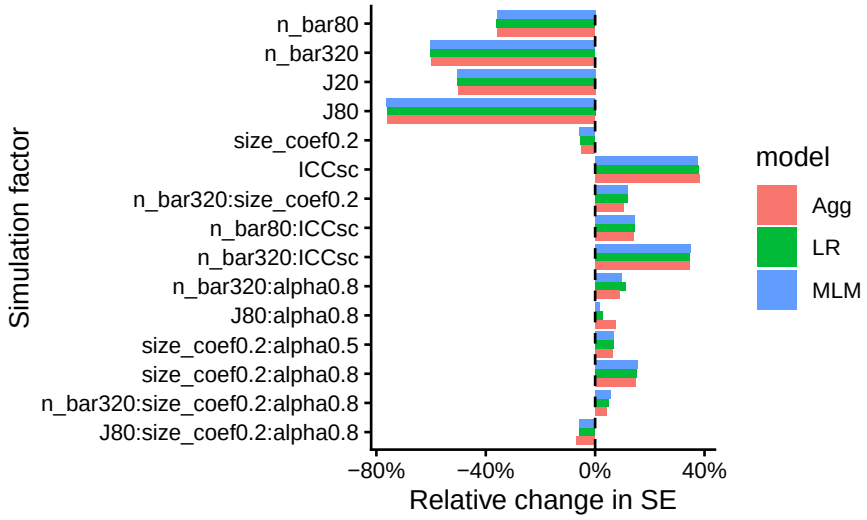
Of course, this is table is hard to read. Better to instead plot the coefficients:

```
lvl = m_res$term
m_resL <- m_res %>%
  pivot_longer( -c( order, term ),
                names_to = "model", values_to = "estimate" ) %>%
  mutate( term = factor(term, levels = rev(lvl) ) )

ggplot( m_resL,
        aes( x = term, y = estimate,
             fill = model, group = model ) ) +
  facet_wrap( ~ model ) +
  geom_bar( stat = "identity", position = "dodge" ) +
  geom_hline(yintercept = 0 ) +
  coord_flip()
```



Here we see how both LR and MLM stand out, but for different simulation factor combinations (see, e.g., the interaction of zeroICC, alpha being 0.8, and size_coef being 0.2). Our coefficient plot provides some understanding of how the methods are similar, and dissimilar.

For another example we turn to the standard error. Here we regress $log(SE)$ onto the coefficients. We then exponentiate the estimated coefficients to get the relative change in SE as a function of the factors. We can interpret an exponentiated coefficient of, for example, 0.64 for MLM for `n_bar80` as a 36% reduction of the standard error when we increase n_bar from the baseline of 20 to 80. We use ordinary least squares and include all interactions up to three way interactions. We will then simply drop all the tiny coefficients, rather than use the full LASSO machinery, to simplify our output. This results in a plot similar to the above:



Our plot clearly shows that the three methods are basically the same in terms of uncertainty estimation, with a some differences when alpha is 0.8. We also see some interesting trends, such as the impact of n_bar declines when ICC is higher (see the positive interaction terms at right of plot).

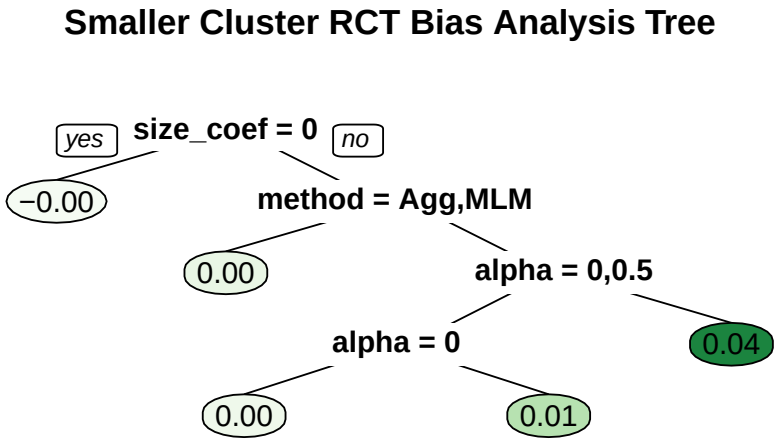
14.2 Using regression trees to find important factors

With more complex experiments, where the various factors are interacting with each other in strange ways, it can be a bit tricky to decipher which factors are important and what patterns are stable. Another exploration approach we can use are regression trees.

We wrote a utility method, a wrapper to the `rpart` package, to do this (script here). Fitting a tree that regresses our performance onto the factors will show us which factors are most related to performance. Here, for example, we see

what predicts larger bias amounts in our cluster RCT running example:

```
source( here::here( "code/create_analysis_tree.R" ) )

set.seed(12411)
create_analysis_tree( sres_f,
                      outcome = "bias",
                      predictor_vars = c("method", "n_bar", "J",
                                         "size_coef", "ICC", "alpha"),
                      tree_title = "Smaller Cluster RCT Bias Analysis Tree",
                      max_leaves = 5 )
```

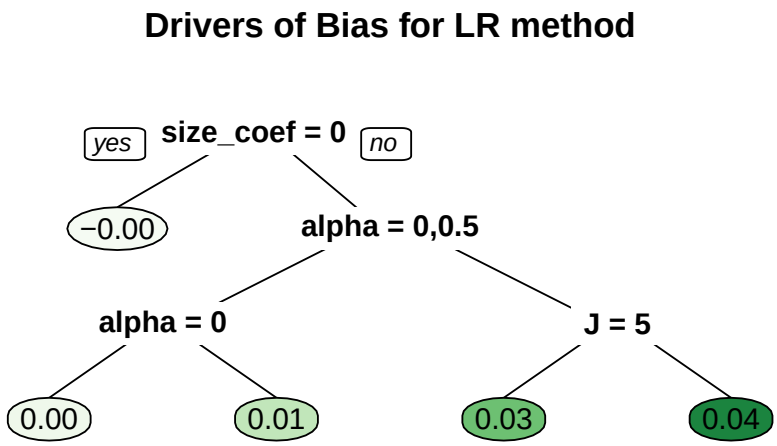


This tree gives a very straightforward story: if `size_coef` is not 0 and we are using LR, then alpha drives bias.

Note that the default pruning for a tree is based on a cross-fitting evaluation, but as our sample size is not too terribly high (just the number of simulation scenarios fit), cross-fitting is quite unstable. Rerunning the code with a different seed will generally give a different tree. To control stability, we find that it is often worth forcibly controlling the size of the tree. As trees are built greedily, forcibly trimming often leaves you only with the big things. As we are using these trees as exploratory tools describing our simulation results, we do not need to worry about whether it is truly the best fit; we are creating visualizations, not doing formal testing.

We can also zero in on specific methods to understand how they engage with the simulation factors, by fitting a tree to just that method's results, as so:

```
create_analysis_tree( filter( sres_f, method=="LR" ),
                      outcome = "bias",
                      min_leaves = 5,
                      predictor_vars = c("n_bar", "J",
                                         "size_coef", "ICC", "alpha"),
                      tree_title = "Drivers of Bias for LR method" )
```



Here we force more leaves to get at some more nuance. We again immediately see, for the LR method, that bias is large when we have non-zero size coefficient *and* a large alpha value. We also see that J may have some additional effect.

Generally we would not use trees like this for a final reporting of results, but they can be important tools for *understanding* your results, which leads to how to make and select more conventional figures for an outward facing document.

14.3 Analyzing results with few iterations per scenario

When each simulation iteration is expensive to run (e.g., if fitting your model takes several minutes), then running thousands of iterations for many scenarios may not be computationally feasible. But running simulations with only a small number of iterations will yield very noisy estimates of estimator performance for that scenario.

Now, if the methods being evaluated are substantially different, then differ-

ences in performance might still be evident even with only a few iterations. More generally, however, the Monte Carlo Standard Errors (MCSEs) may be so large that you will have a hard time discriminating between systematic patterns and noise.

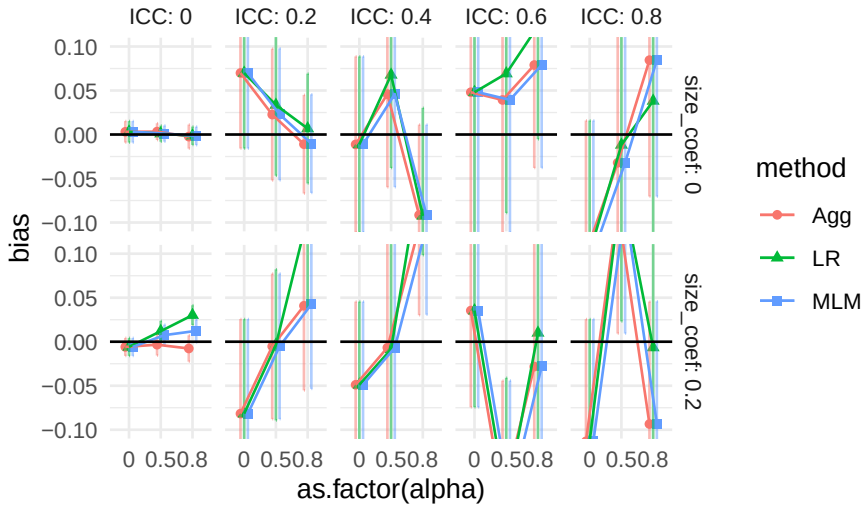
One tool to handle few iterations is aggregation: if you average across scenarios, those averages will have more precise estimates of (average) performance than the estimates of performance within the scenarios. Do not, by contrast, trust the bundling/boxplot approach–the MCSEs will make your boxes wider, and give the impression that there is more variation across scenarios than there really is.

Meta regression approaches such as we saw above can also be useful: a regression effectively averages performance across scenario, and give summaries of overall trends. You can even fit random effects regression, specifically accounting for the noise in the scenario-specific performance measures. For more on using random effects for your meta regression see Gilbert and Miratrix (2024).

14.3.1 Example: ClusterRCT with only 25 replicates per scenario

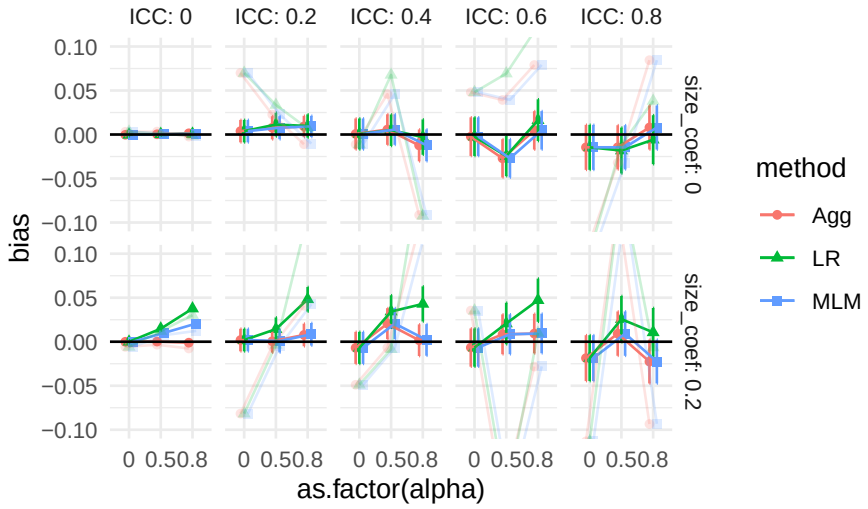
Previously, we have analyzed the results of our cluster RCT simulation with 1000 iterations per scenario. But say we only had 25 iterations per scenario. Using the prior chapter as a guide, we next recreate some of the plots to show how MCSE can distort the picture of what is going on, and how averaging can mitigate this somewhat.

We start again with bias, and generate a single plot of the raw results for a subset of our scenarios, along with error bars from the MCSEs.



Other than the ICC = 0 case, we see substantial amounts of uncertainty, making it very hard to tell the different estimators apart. (Our uncertainty is much less when ICC is 0 because our estimators are far more precise due to not having cluster variation to contend with.) In the top row, second plot from left, we also see that the three estimators are co-dependent: they all react similarly to the same datasets. In other words, the point estimates are correlated: if a dataset happens to have high-outcome units in the treated group, all the estimators will give larger estimates of an average effect. The seeming trend we are seeing is not systematic bias, but rather shared random variation.

Here is the same plot with the full 1000 replicates, with the 25 replicate results overlaid in light color for comparison:



With 1000 iterations, MCSEs are around $1/\sqrt{40} = 16\%$ the size, as we would expect (generally MCSEs are on the order of $1/\sqrt{R}$, where R is the number of replicates, so to halve a MCSE you need to quadruple the number of replicates). Also note the ICC=0.2 top facet has shifted to a flat, slightly elevated line: we do not yet know if the elevation is real, just as we did not know if the dip in the prior plot was real, but we do know that whatever might be happening is of a much smaller scale than suggested by the prior plot. Our confidence intervals mostly all still include 0: it is possible there is no bias at all when the size coefficient is 0 (in fact we are fairly sure this is indeed the case).

Moving back to our "small replicates" simulation, we can use aggregation to smooth out some of our uncertainty. For example, if we aggregate across 9 scenarios, our number of replicates goes from 25 to 225 and our MCSEs should be about a third the size. To calculate an aggregated MCSE, aggregate the scenario-specific MCSEs as follows:

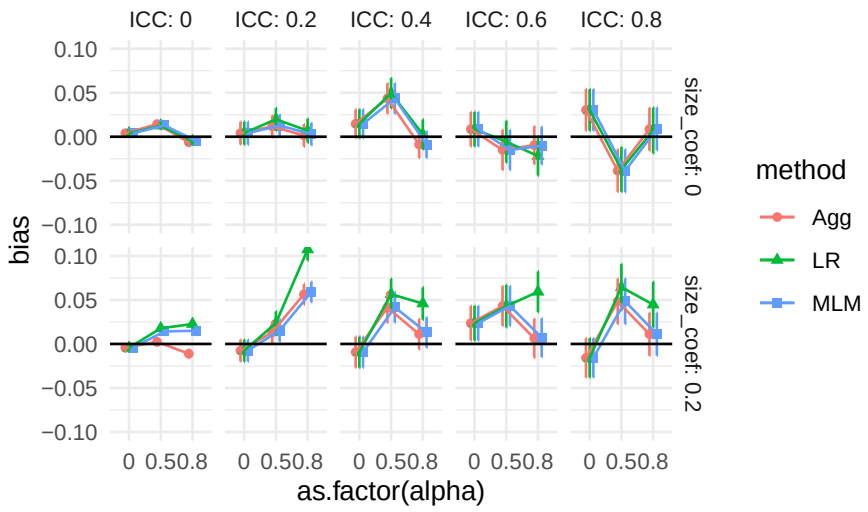
$$MCSE_{agg} = \sqrt{\frac{1}{K^2}\sum_{k=1}^{K} MCSE_k^2}$$

where $MCSE_k$ is the Monte Carlo Standard Error for scenario k, and K is the number of scenarios being averaged. Assuming a collection of estimates are independent, the overall SE^2 of an average is the average SE^2 divided by K.

In code we have:

Recall that the `SE` variable is simply the standard deviation of the estimates.

We can then make our aggregated bias plot, aggregating across `n_bar` and `J`:



Even with the additional replicates per point, we see noticeable noise in our plot: look at the top-right ICC of 0.8 facet, for example. Also note how our three methods continue to track each other up and down in the top row, due to shared error. This shared error can be deceptive. It can also be a boon: if we are explicitly comparing the performance of one method vs another, we can subtract out the shared error, similar to what happens in a blocked experiment (Gilbert and Miratrix, 2024). We discuss this approach next.

14.4 Multilevel Meta Regressions

One way to get a more precise head to head comparison of the estimators is to fit a multilevel regression model to our raw simulation results with a random effect for dataset. This can be especially useful as a tool for analyzing results with few iterations.

We continue the running example of the prior section and fit such a model, taking advantage of the fact that estimated bias is simply the average of the error across replicates. We first make a unique ID for each scenario and dataset, and then fit the model with a random effect for both. The first random effect allows for specific scenarios to have more or less bias beyond what our model predicts. The second random effect allows for a given dataset to have a larger or smaller error than expected, shared across the three estimators.

```
library(lme4)
res_small <- res_small %>%
  mutate(
    error = ATE_hat - ATE,
    simID = paste(n_bar, J, size_coef, ICC, alpha, sep = "_"),
    dataID = paste( simID, runID, sep="_" ),
    J = as.factor(J),
    n_bar = as.factor(n_bar),
    alpha = as.factor(alpha),
    size_coef = as.factor(size_coef)
  )

M <- lmer(
  error ~ method + (1|dataID) + (1|simID),
  data = res_small
)
arm::display(M)
```

```
lmer(formula = error ~ method + (1 | dataID) + (1 | simID), data = res_small)
            coef.est coef.se
(Intercept) 0.01     0.01
methodLR    0.01     0.00
methodMLM   0.00     0.00

Error terms:
 Groups   Name        Std.Dev.
 dataID   (Intercept) 0.39
 simID    (Intercept) 0.04
 Residual             0.06
---
number of obs: 20250, groups: dataID, 6750; simID, 270
AIC = -24176.3, DIC = -24254
deviance = -24221.2
```

We can look at how much each source of variation explains the overall error:

```
ranef_vars <-
  as.data.frame(VarCorr(M)) %>%
  dplyr::select(grp = grp, sd = vcov) %>%
  mutate( sd = sqrt(sd),
          ICC = sd^2 / sum(sd^2 ) )

knitr::kable(ranef_vars, digits = 2)
```

grp	sd	ICC
dataID	0.39	0.97
simID	0.04	0.01
Residual	0.06	0.02

The random variation for `simID` captures unexplained variation due to the interactions of the simulation factors. Relative to the dataset variation, it seems to be a trivial amount. This makes sense: each datasets is unbalanced due to random assignment, and that estimation error, shared across all three estimators, is part of the dataset random effect.

So far we have not included any simulation factors: we are pushing variation across simulation into the random effect terms. We can instead include the simulation factors as fixed effects, to see how they impact bias.

```
M2 <- lmer(
  error ~ method*(J + n_bar + ICC + alpha * size_coef) + (1|dataID) + (1|simID),
  #error ~ method*(J * n_bar * alpha * size_coef + ICC) + (1|dataID) + (1|simID),
  data = res_small
)
```

We then extract the fixed effects and sort by p-value to see which factors are most important for bias. We adjust our p-values with a simple Benjamini-Hochberg false discovery rate correction to try and cut down the number of terms to those that are most likely to be real, and not just noise.

```
library( broom.mixed )
terms <- tidy( M2 ) %>%
  dplyr::filter( effect == "fixed" ) %>%
  dplyr::select( term, estimate, std.error, p.value )  %>%
  mutate(p_fdr = p.adjust(p.value, method = "BH") )

terms %>%
  filter( p_fdr < 0.10 ) %>%
  arrange( p_fdr ) %>%
  knitr::kable( digits=c(3) )
```

term	estimate	std.error	p.value	p_fdr
methodLR:alpha0.8:size_coef0.2	0.04	0.005	0	0

We see what we have seen before: the LR method is notably biased when `size_coef` is non-zero and `alpha` is high. We have estimated how bias varies with method and simulation factor, while accounting for the uncertainty in the simulation.

Finally, we can see how much variation has been explained by the simulation factors by comparing the random effect variances:

```
ranef_vars1 <-
  as.data.frame(VarCorr(M)) %>%
  dplyr::select(grp = grp, sd = vcov) %>%
  mutate( sd = sqrt(sd),
          ICC = sd^2 / sum(sd^2 ) )

ranef_vars2 <-
  as.data.frame(VarCorr(M2)) %>%
  dplyr::select(grp = grp, sd = vcov) %>%
  mutate( sd = sqrt(sd),
          ICC = sd^2 / sum(sd^2 ) )

rr = left_join( ranef_vars1, ranef_vars2, by = "grp",
                suffix = c(".null", ".full") )

rr <- rr %>%
  mutate( var.red = sd.full^2 / sd.null^2 )
knitr::kable(rr, digits = 2)
```

grp	sd.null	ICC.null	sd.full	ICC.full	var.red
dataID	0.39	0.97	0.39	0.97	1.00
simID	0.04	0.01	0.03	0.01	0.84
Residual	0.06	0.02	0.06	0.02	0.98

The `var.red` column is how much variation is explained by our model of similation factors. Our model is explaining an estimated 16% of the variation across simulation scenarios. This implies that there is a lot of variation due to interactions between the simulation factors that are not in our model.

14.5 What to do with warnings in simulations

Sometimes our analytic strategy might give some sort of warning (or fail altogether). In Section 7.4 we saw how to trap such warnings or errors. In particular, in Section 7.4.1 we illustrated error trapping on the multilevel model for the cluster RCT experiment by revising our `analysis_MLM()` method to an `analysis_MLM_safe()` method, which gave output from the MLM method that looked like this:

```
set.seed(101012)  # (I picked this to show an issue.)
dat <- gen_cluster_RCT( J = 50, n_bar = 100, sigma2_u = 0 )
analysis_MLM_safe( dat )
```

```
# A tibble: 1 x 6
   ATE_hat SE_hat p_value message warning error
     <dbl>  <dbl>   <dbl> <lgl>   <lgl>   <lgl>
1 -0.00880 0.0278   0.751 TRUE    TRUE    FALSE
```

For this dataset, a convergence issue that gave rise to the indicated message and warning.

Once the full simulation is run, we can examine the patterns of these warnings and errors to see if they are a serious or minor problem. We first do this by calculating the rate of errors or warning messages as a performance measure in its own right. Especially if we plan on possibly dropping some trials, it is important to see how often our estimators are acting peculiarly.

For example, in our cluster RCT running example, we know that ICC is an important driver of when convergence issues might occur, so we can explore how often we get a convergence message by ICC level by calculating percent change of a message, warning, or error:

```
res %>%
  dplyr::filter( method == "MLM" ) %>%
  group_by( method, ICC ) %>%
  summarise( message = 100*mean( message ),
             warning = 100*mean( warning ),
             error = 100*mean( error ) ) %>%
  pivot_wider( names_from = method,
               values_from=c( "message", "warning", "error" ) )
```

```
# A tibble: 5 x 4
    ICC message_MLM warning_MLM error_MLM
  <dbl>       <dbl>       <dbl>     <dbl>
1   0        49.0        0.0556          0
2   0.2       1.28       0.00741         0
3   0.4       0.359      0.00185         0
4   0.6       0.124      0               0
5   0.8       0.0278     0               0
```

We see that when the ICC is 0 we get a lot of convergence issues (nearly half of our runs have a message), but as soon as we pull away from 0 it drops off considerably. Thankfully, we never saw actual errors and our warnings were very rare.

At this point we have to decide whether to drop the problematic runs or keep them. Ideally, failure would not be too common, meaning we could drop or

keep the problematic iterations without really impacting our overall results.

In our case, for example, it should not matter much, except possibly when ICC = 0, and we know the convergence issues are driven by 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 is actually falling apart, or if it is just throwing warnings.

Importantly, if the decision is to drop, then the entire simulation iteration, including the other estimators, should be dropped even if the other estimators worked fine! If there is something particularly unusual about the dataset that caused the concern, then dropping for one estimator and keeping for the others 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. Consider, for example, if the dropped datasets were fundamentally harder to evaluate.

14.6 Conclusions

Designed experiments in simulation are only as useful as the summaries you make of their output. In this chapter, we showed a few advanced ways of moving from raw simulation outputs to interpretable summaries using regression, ANOVA, LASSO, trees, and multilevel meta-models. These tools let you parse the main effects of your simulation factors along with their interactions, and can help turn many scenario-level estimates into clear statements about where methods work and where they break. They can also help protect against how Monte Carlo simulation error can potentially distort findings.

We also focused on two specific areas where analyzing a set of results can be tricky: when each iteration is expensive (so you do not have too many of them), and when sometimes your estimators fail or perform poorly in a way that raises warnings or flags. In the former case, aggregation and multilevel models can be especially useful when replicates are expensive. In the latter, evaluate when failures occur, and then either drop them consistently or explicitly model them.

Overall, these tools give a way to go deeper into a set of simulation results. Often the findings generated by these tools would not be presented in the main text, but they can be important supporting detail, especially when placed in a supplement.

15

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 this part by reiterating this point, and then discussing several ways researchers might design their simulations so they more arguably produce real evidence.

When our work is done, ideally we will have generated simulations that provide a sort of "consumer product testing" for the statistical methods we are exploring. Clearly, it is important to do some consumer product tests that are grounded in how consumers will actually use the product in real life—but before you get to that point, it can be helpful to cover an array of conditions that are probably more extreme than what will be experienced in practice (simulations akin to dropping things off buildings or driving over them with cars or whatever else). For simulation, extreme levels of a factor make understanding the role they play more clear. Extreme contexts can also help lay bare the limits of our methods.

"Extreme" can go in both the good and bad directions. Extreme contexts can include best-case (or at least optimistic) scenarios for when our estimator of interest will excel, giving a nominal upper bound on how well things could go. Usually extreme simulations are simple ones, with fairly straightforward data generating processes that do not attempt to capture the complexities of real data. Such extreme simulations might be considered overly *clean* tests.

Extreme and clean simulations are also usually easier to write, manipulate, and understand. Such simulations can help us learn about the estimators themselves. For example, we can use simulation to uncover how different aspects of a data generating process affect the performance of an estimator. To discover this, we would need a data generation process with clear levers controlling those aspects. Simple simulations can be used to push theory further—we can experiment to gain an inclination of whether a novel approach to something is a good idea at all, or to verify we are right about an intuition or derivation about how well an estimator can work.

But such simulations cannot be the end of our journey. In general, a researcher should work to ensure their simulation evidence is *relevant*. The evidence should inform how things are likely to work out in practice. A set of simulations only using unrealistic data generating processes may not be that useful.

Unless the estimators being tested have truly universally stable properties, we will not learn much from a simulation if it is not relevant to the problem at hand. After exploring the overly extreme or overly clean, we have to circle back to providing testing of the sorts of situations an eventual user of our methods might encounter in practice. So how can we make our simulations more relevant?

15.1 Making simulations relevant

In the following subsections we go through a range of general strategies for making relevant simulations:

1. Break symmetries and regularities.
2. Use extensive multi-factor simulations.
3. Use simulation DGPs from prior literature.
4. Pick simulation factors based on real data.
5. Resample real data directly to get authentic distributions.
6. Design a fully calibrated simulation.

15.1.1 Break symmetries and regularities

In a series of famous causal inference papers (Lin, 2013; Freedman, 2008), researchers examined when linear regression adjustment of a randomized experiment (i.e., when controlling for baseline covariates in a randomized experiment) could cause problems. Critically, if the treatment assignment is 50%, then the concerns that these researchers examined do not come into play, as asymmetries between the two groups get perfectly cancelled out. That said, if the treatment proportion is more lopsided, then under some circumstances you can get bias and invalid standard errors, depending on other structures of the data.

Simulations can be used to explore these issues, but only if we break the symmetry of the 50% treatment assignment. When designing simulations, it is worth looking for places of symmetry, because in those contexts estimators will often work better than they might otherwise, and other factors may not have as much of an effect on performance as anticipated.

Similarly, in recent work on best practices for analyzing multisite experiments (Miratrix et al., 2021), we identified how different estimators could be targeting different estimands. In particular, some estimators target site-average treatment effects, some target person-average treatment effects, and some target a kind of precision-weighted blend of the two. To see this play out in practice, our simulations needed the sizes of sites to vary, and also the pro-

portion of treated within site to vary across sites. If we had run simulations with equal site size or equal proportion treated, we would not see the broader behavior that separates the estimators considered.

Overall, it is important to make your data generation processes irregular.

15.1.2 Use extensive multi-factor simulations.

"If a single simulation is not convincing, use more of them," is one principle a researcher 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 of your DGP that is not varying is the real culprit. If this aspect is unrealistic, then the findings, across the board, may be less relevant. Thus, pick the factors one varies with care, and pick factors in anticipation of reviewer critique.

15.1.3 Use simulation DGPs from prior literature.

If a relevant 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. In effect, use previously published simulations to beat them at their own game. Using an established DGP makes it (more) clear that you are not fishing: you are using something that already exists 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. Indeed, such a benchmarking approach is often taken in the machine learning literature, where different canonical datasets are used to benchmark the performance of new methods.

15.1.4 Pick simulation factors based on real data

Use real data to inform choice of factor levels or other data generation features. For example, in James's work on designing methods for meta-analysis, there is often a question of how big simulated sample sizes should be and how many different simulated outcomes per study there should be when generating sets of hypothetical studies to be included in hypothetical meta-analyses. In one project, to make such simulations realistic, James obtained data from a set of past real meta-analyses and fit parametric models to these features, and then used the resulting parameter estimates as benchmarks for how large and how varied the simulated meta analyses should be.

When doing this, we recommend not using the exact estimated values, but instead using them as a point of reference. For instance, say we find that the

distribution of study sizes fits a Poisson(63) pretty well. We might then then simulate study sizes using a Poisson with mean parameters of 40, 60, or 80 (where 40, 60, and 80 would be one factor in our multifactor experiment).

15.1.5 Resample real data directly to get authentic distributions

You can use real data directly in your simulation. For example, say you need a population distribution for a slew of covariates. Rather than using something artificial (such as a multivariate normal), you can pull population data on a collection of covariates from some administrative data and then use those data as a population distribution. You then sample (probably with replacement) rows from your reference population to generate your simulation sample.

The appeal here is that your covariates will then have distributions that are much more authentic than some multivariate normal–they will include both continuous and categorical variables, they will tend to be skewed, and they will be correlated in different ways that are more interesting than anything someone could make up.

Long and Ervin (2000) used an approach like this in their study of heteroscedasticity-robust standard errors. They wanted to explore how well these standard errors worked in practice, with a focus on how high leverage points could undermine their effectiveness. Leverage is purely a function of the covariate distribution, so they needed to have an authentic covariate distribution in order to properly explore this issue. Generating realistic leverage is hard to do with a parametric model, so instead they used examples from real life to make the simulation more relevant.

Resampling the real data, and then adding an outcome model on top of that, is a form of **semi-synthetic** simulation, to acknowledge the hybrid approach, or **plasmode simulation**, which has ties to electronic health records research. These approaches are often used in causal inference work, where investigators attempt to determine which of the many observational study tools might work best, and when. This context is particularly hard because even with real data in hand, there is no observed ground truth as to what the actual treatment effects are; everything must be estimated.

A key advantage of a semi-synthetic or plasmode approaches is that they can naturally adapt to contexts, such as electronic health records research, where covariate spaces can be extraordinarily high-dimensional, with complex dependence structures that purely parametric simulation cannot plausibly reproduce (Franklin et al., 2014). By simply bootstrapping the covariates from the raw data, all the dependencies are preserved. A design question, if working in a causal inference framework, is then whether any treatment indicator should be included along with the covariates or generated independently; see, e.g., Shaw et al. (2025), who argue that sampling the treatment can cause

estimators to appear spuriously biased in some cases.

15.1.6 Build a calibrated simulation

Extending some of the prior ideas even further, one practice in increasing vogue is to generate *calibrated simulations*. See Kern et al. (2014) for an example of this in the context of generalizing experiments. This paper has one of the earliest uses of this term, as far as we know.

A calibrated simulation is one that is tailored to a specific applied context, with a DGP that is close to what we think the real world is like in that context. Usually, this means we try to generate data that looks as much like a target dataset as possible, with the same covariate distributions and the same relationships between covariates and outcomes. These simulations are not particularly general, but instead are designed to more narrowly inform what how well things would work for a particular circumstance.

Once we have a fully calibrated DGP, targeting a specific dataset, we can then modify it by adjusting sample sizes or other aspects to explore a space of DGPs that are all connected to this core initial calibrated DGP. We can thus still build more general evidence, but the evidence will always be anchored by our initial target.

We usually would build a calibrated simulation out of existing data. As discussed in the prior section, we might start with sampling 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.

Just putting a model on top of a real dataset does not necessarily achieve true calibration. The relationship between our covariates and outcome would need to, even if it is model dependent, be realistic in order to truly assess how well the estimators work in practice. It is very easy to accidentally put a very simple model in place for this final component, thus making a calibrated simulation that is quite naive in, perhaps, the very way that counts.

One approach to get a more realistic model between covariates and outcomes is to fit a flexible model to the original data, and then use the model to predict outcomes from the covariates. One would then generate residuals to add to the predictions, ideally in a way that preserves the heteroscedasticity and overall outcome distribution of the original data. For example, in the subsection below, we provide a copula based approach to doing this, which ensures the outcome distribution is in line with the original data while allowing control over how tightly coupled the covariates are to the outcome. We do not say this is the only approach, but rather one of a large range of possibilities.

Clearly, these calibration games can be fairly complex. Calibrated simulations frequently do not lend themselves to a clear factor-based structure that have

levers that change targeted aspects of the data generating process (although sometimes you can craft your approach to make such levers possible). In exchange for this loss of clean structure, we end up with a simulation that might be more faithfully capturing aspects of a specific context, making our simulations more relevant to answering the narrower question of "how will things work here?" even if they are less relevant for answering the question of "how will things work generally?"

Semi-synthetic approaches occupy the middle ground between purely synthetic simulation (where every variable is fabricated from pre-specified distributions and the data bear no necessary resemblance to any real population) and fully empirical analysis (where ground truth treatment effects may be unobservable, making evaluation of an estimator difficult).

One canonical motivation for a semi-synthetic approach is the fundamental problem of causal inference: because we never observe counterfactual outcomes (e.g., the outcome we would have seen for a treated unit if it had not been treated) we cannot soly rely on the raw data if we are estimated in the "real" treatment effects. A famous example of a semi-synthetic approach is the IHDP benchmark, in which Hill (2011) removed a subset of treated units from the Infant Health and Development Program randomized trial to introduce selection bias, and then replaced the observed outcomes with simulated potential outcomes generated from a nonparametric Bayesian model; this benchmark subsequently anchored a large fraction of the machine-learning literature on heterogeneous treatment-effect estimation. See, for example the Atlantic Causal Inference Conference data challenge, discussed in Dorie et al. (2019). Gentzel et al. (2021) further systematized this approach, walking through different ways of generating data for simulation evaluation of causal inference methods.

15.2 Case study: Using copulas to generate calibrated data

In this section, we borrow from a blog post[1] to illustrate how to use copulas to generate data that is very calibrated to a target dataset. In this example, we will generate new observations (X, Y), where X is a vector of covariates (some continuous, some categorical) and Y is a continuous outcome of an arbitrary distribution as described by a target dataset.

A copula is a mathematical device that is used to describe a multivariate distribution in terms of its marginals and the correlation structure between them, and we used it to generate new Y_i values based on the X_i in a way that

[1] See (https://cares-blog.gse.harvard.edu/post/copulas-for-simulation/)

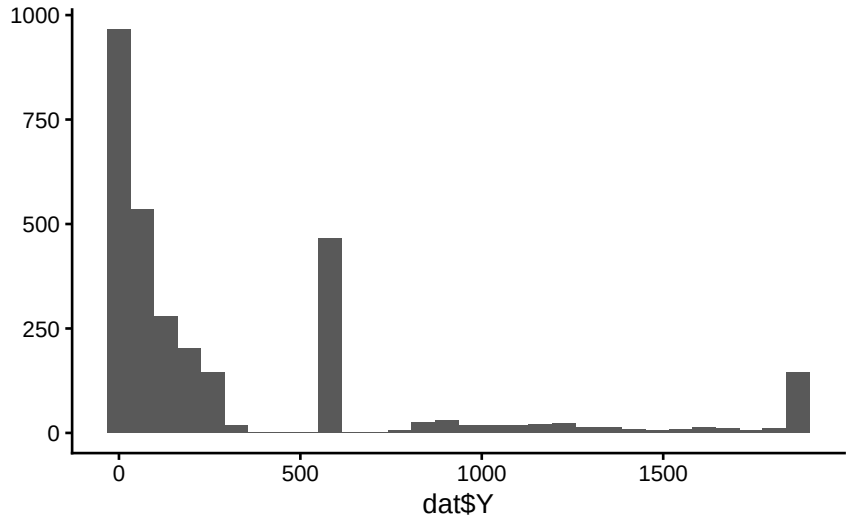
maintained the same marginal distributions for all variables as the empirical dataset.

We found the copula to have several benefits. First, the copula approach can give a different Y for a given X. By contrast, simply bootstrapping our empirical data (a common way to generate synthetic data that looks like an empirical target) would not achieve this. Second, we could control the tightness of the coupling between X and Y, while maintaining a fairly complex and nonlinear relationship between X and Y, and while maintaining possible heteroskedasticity found in the initial dataset.

To illustrate, we will use a (fake) dataset with four covariates, X1, X2, X3, and X4, and an outcome, Y. Here are a few rows of this "empirical" data that we are trying to match:

X1	X2	X3	X4	Y
3	1	5.71	2.04	1076.05
5	1	1.50	3.23	550.00
2	1	3.44	1.18	24.99
-1	1	2.29	0.25	16.24
2	1	7.15	1.51	1870.79

Our outcome in this case is a bit funky:



The covariates have a correlation structure with themselves and also Y. Here are the pairwise correlations on the raw values:

```
print( round( cor( dat ), digits = 2 ) )
```

```
      X1    X2   X3    X4    Y
X1  1.00 -0.24 0.36  0.82 0.51
X2 -0.24  1.00 0.20 -0.19 0.17
X3  0.36  0.20 1.00  0.31 0.68
X4  0.82 -0.19 0.31  1.00 0.44
Y   0.51  0.17 0.68  0.44 1.00
```

We want to generate synthetic data that captures the correlation structure of our empirical data along with the marginal distributions of each covariate and outcome. We will do this via the following steps:

Steps for making data with a copula:

1. Predict Y given the full set of covariates X using a machine learner.
2. Tie $\hat{Y}_i$ and Y_i together with a copula.
3. Generate new Xs (in our case, via the bootstrap).
4. Generate a predicted $\hat{Y}_i$ for each X_i using the machine learner.
5. Generate the final, observed, Y_i using our copula.

15.2.1 Step 1. Predict the outcome given the covariates

We can use some machine learning model to make predictions of Y given X. Here we use a random forest:

```
library(randomForest)
mod <- randomForest( Y ~ .,
                     data=dat,
                     ntree = 200 )
dat$Y_hat = predict( mod )
```

Our predictive model for Y can be anything we want: we are just trying to get some predicted Y that is appropriately tied to the X.

Our predictions will be under dispersed and not natural in appearance. For starters, the standard deviation of the $\hat{Y}$ will be less than the original Y, because our predictions are only getting at the structural aspects, not the residual noise:

```
tibble( sd_Yhat = sd( dat$Y_hat ),
        sd_Y = sd( dat$Y ),
        cor = cor( dat$Y_hat, dat$Y ) ) %>%
  knitr::kable( digits = c(1,1,2) )
```

sd_Yhat	sd_Y	cor
333.2	493.3	0.91

Our synthetic Y values will need to be "noised up" so they look more natural. We also want them to match the empirical distribution of our original data.

15.2.2 Step 2. Tie $\hat{Y}$ to Y with a copula

The first step of the copula approach is to make a function that converts empirical data to normalized z-scores based on their percentile rank, as we will be doing all our data generation in "normal z-score space." In other words, we take a value, calculate its percentile, and then calculate what z-score that percentile would have, if the value came from a standard normal distribution.

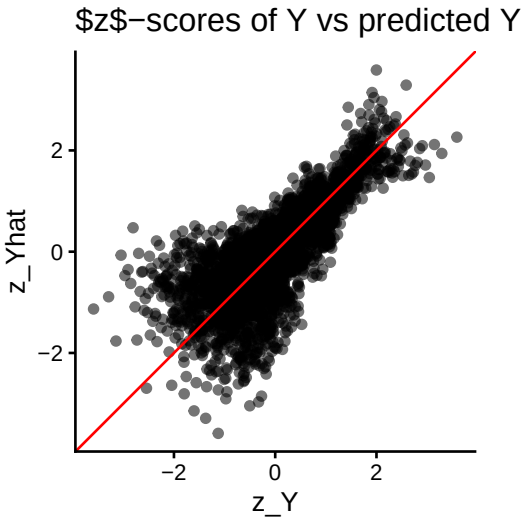
Here is a function to do this:

```
convert_to_z <- function( Y ) {
  K = length(Y)
  r_Y <- rank( Y, ties.method = "random" )/K - 1/(2*K)
  z <- qnorm( r_Y )
  z
}
```

We calculate percentiles by ranking the values, and then taking the midpoint of each rank. If we had 20 values, for example, we would then have percentiles of 5%, 15%, ..., 95%. (We can't have 0 or 100% since this will turn into Infinity when converting to a z-score.)

Using our function, we convert both our Y and predicted Y to these normal z-scores:

```
z_Y = convert_to_z( dat$Y )
z_Yhat = convert_to_z( dat$Y_hat )

qplot( z_Y, z_Yhat, asp = 1, alpha=0.05 ) +
  geom_abline( slope=1, intercept=0, col="red" ) +
  labs( title = "$z$-scores of Y vs predicted Y" ) +
  coord_fixed() +
  theme( legend.position = "none" )
```



Note how the z-scores have smoothed our Y distribution due to the random ties, breaking up our spikes of identical values. We can then see how coupled the predicted and actual are:

```
stats <- tibble(
  rho = cor( z_Y, z_Yhat ),
  sd_Y = sd( z_Y ),
  sd_Yhat = sd( z_Yhat ) )
stats
```

```
# A tibble: 1 x 3
    rho  sd_Y sd_Yhat
  <dbl> <dbl>   <dbl>
1 0.771 1.000   1.000
```

15.2.3 Step 3. Get some new Xs

So far we are just modeling relationship in our original data. We next set the stage for generating a new dataset. We start with making more X values. A simple trick is to simply bootstrap the data, keeping the X only. We might, for example, generate a new dataset of 500 observations like so:

```
nd = dat %>%
  dplyr::select( starts_with("X") ) %>%
  slice_sample( n = 500, replace = TRUE )
```

If we don't want exact duplicates of the vectors of X values in our data, we can use other methods such as the `synthpop` package, which generates data based on a series of conditional distributions.

So we can check our work later, we are instead going to cheat and use a dataset, `new_data` that is generated from the same DGP as our original fake data. It looks like so:

```
# A tibble: 3,000 x 5
      X1    X2    X3     X4 trueY
   <dbl> <dbl> <dbl>  <dbl> <dbl>
 1     1     1 0.870  0.387  15.4
 2     4     1 0.336  2.30  169.
 3     5     0 2.36   2.19   57.7
 4     1     0 1.83  -0.630  20.2
 5     0     1 4.00  -0.426  50.8
 6     3     1 1.58   2.00  133.
 7     5     1 4.63   3.53  550
 8     4     0 3.65   1.01   44.5
 9     0     1 4.90   0.784  38.6
10     5     1 1.40   3.31  550
# i 2,990 more rows
```

To be very specific, `trueY` would not normally be in our data at this point; we have it here so we can compare our generated Y to the true Y to see how well our DGP is working. In practice, we would not be able to do this: we would need to generate new X like shown above.

15.2.4 Step 4. Generate a predicted $\hat{Y}_i$ for each X_i using the machine learner.

This step is quite easy:

```
new_data$Yhat = predict( mod, newdata = new_data )
```

Done!

15.2.5 Step 5. Generate the final, observed, Y_i using a copula that ties $\hat{Y}_i$ to Y_i.

Now we generate our new Y by converting our predictions (the Yhat) to the new Y via the copula. We are bascially using the same pieces that are in the z-score function, above.

We first generate the percentiles the Y-hats represent:

```
K = nrow(new_data)
new_data$r_Yhat = rank( new_data$Yhat ) / K - 1/(2*K)
summary( new_data$r_Yhat )
```

```
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
```

```
0.0001667 0.2500833 0.5000000 0.5000000 0.7499167 0.9998333
```

We then convert the percentiles to the z-scores they represent:

```
new_data$z_Yhat = qnorm( new_data$r_Yhat )
summary( new_data$z_Yhat )
```

```
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-3.587915 -0.674228  0.000000 -0.000001  0.674228  3.587915
```

We next use a formula for the conditional distribution of a normal variable given the other variable, namely, if X_1, X_2 both have unit variance and mean zero, we use

$$X_2|X_1 = a \sim N(\rho a, 1 - \rho^2).$$

With this formula, we generate Y as so:

```
rho = cor( z_Y, z_Yhat ) # from our empirical
new_data$z_Y = rnorm( K,
                      mean = rho * new_data$z_Yhat,
                      sd = sqrt( 1-rho^2 ) )
```

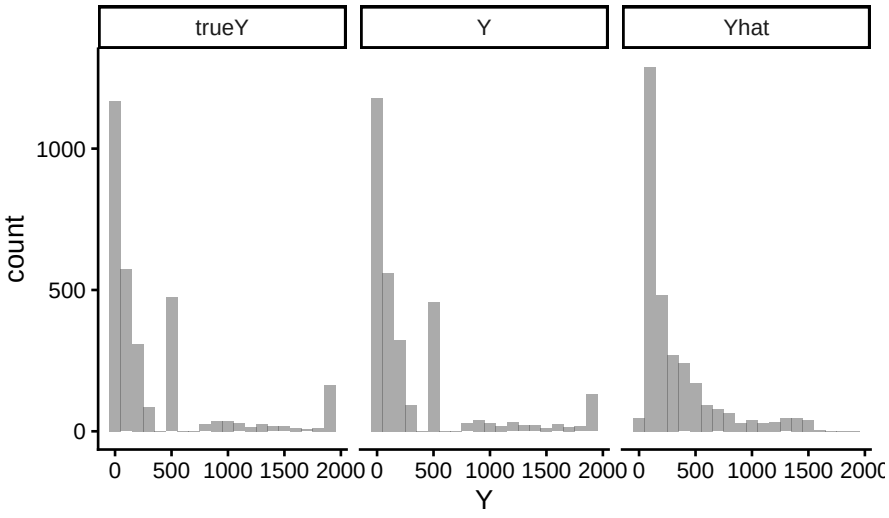
Now, finally, for each observation in z-space, we figure out what rank it would correspond to in our empirical distribution, and then take that empirical data point as the value:

```
empDist = sort(dat$Y)
K = length(empDist)
new_data$r_Y = ceiling( pnorm( new_data$z_Y ) * K )
new_data$Y = empDist[ new_data$r_Y ]
```

And we are done!

15.2.6 Assessing the copula's success

Let us next see how the distribution of our generated Y to the true Y from the real DGP:



`trueY` and Y are very close! Note the Yhat is quite different, by contrast.

We further check our process by looking at how closely the correlations of our new Y with the Xs match the correlations of the true Y and the Xs:

Variable	Y	trueY
X1	0.45	0.53
X2	0.20	0.22
X3	0.50	0.67
X4	0.42	0.45

Our new Y roughly looks like the true Y, with respect to the correlation with the Xs. We gave our copula approach a hard test here–simpler covariate relationships will be better captured. That said, this doesn't seem that bad.

15.2.7 Building a multifactor simulation from our calibrated DGP

In the above, we made a DGP that mimics, as closely as we can, the empirical data we are targeting. In an actual simulation, however, we would generally want to manipulate different aspects of our DGP to explore how different methods might function.

We offer some examples of how to do this for this context.

First, we can directly set `rho` to control how tightly coupled the outcome is to the covariates. If `rho` is 0, then our outcome distribution will be exactly the same, but there would be no connection between it and the covariates. For

`rho` of 1, the outcome would be perfectly predicted by the predicted outcome, which is a function of the covariates. The strength of the connection between X and Y is maximized.

Second, we can easily control sample size by generating larger or smaller datasets of X and then generating Y from those X.

Third, we can drop some of our Xs by simply not including them in the prediction model, thus making them noise variables. We may still end up with relationships between a dropped X and Y due to the dropped X's correlation structure with other Xs that are tied to Y.

15.3 Simulation Sections in Methods Papers

Simulations, as we have argued, have many uses, and appear in many places. One of the most prevalent is the simulation section of a statistical methods paper. In this section, we make a case for what kinds of simulation evidence a methods paper should have, and outline several principles to consider if you are writing up, or reviewing, such a section.

In what follows, we discuss several related goals that simulations in methods papers should serve, along with practical guidance on how to accomplish each of them. We then give some advice on navigating the many design decisions of a simulation, and conclude with some guiding principles for achieving these ends.

15.3.1 The simulation's purpose

Ideally, a simulation study in a methods paper is not decoration. In the best case, it provides evidence for an argument. These arguments can be pointed (the researcher has an agenda to prove) or more exploratory (the researcher is trying to learn something). Which kind of argument correlates with what kind of methods paper is being written.

Broadly speaking, there are two kinds of methods papers that involve simulation. The first is a methods development paper. These usually propose some novel approach, have some (mathematical) theory arguing for its utility, have a simulation to justify or illustrate the argument, and finally an applied example to demonstrate the method's use. For these papers, the argument of the simulation is generally something along the lines of "our proposed method is an improvement over whatever else is out there." The simulation evidence is then the empirical case for whatever claim is being made.

The second type of methods paper is a "simulation paper," where the re-

searchers are specifically using simulation to evaluate some set of research questions. Here there is not necessarily a specific case being made, but instead a research question such as, "which of a set of methods is most appropriate for our given context of interest?" Now the simulation results are evidence to help adjudicate which options are most promising, and when.

In either case, the simulation evidence is only good insofar as the choice of data generating process, factors and levels, and performance metrics are relevant.

15.3.1.1 Increase belief in the theory

We generally hope that a mathematical theorem in a paper is correct, but few of us will spend the time carefully reading any accompanying proof or deriviation to ensure this is so. In fact, many papers have these derivations in a supplement, and often they are cut down, without many of the intermediate steps.

Simulation can provide a sanity check and reassurance in this case. One might even say that the most basic job of a simulation is to verify that the theory behind a method actually works. Such a "sanity check" simulation is often a simple one. A clean, stylized data generating process with the key structural features the theory assumes—perhaps independent errors, a specific parametric form for the outcome, or a particular covariate distribution—is entirely appropriate here. The goal is not to show that the method works in a realistic context; it is to show that the theory does what it says it does. For example, if your theory says the estimator has a bias that diminishes at rate $1/n$, you should see this as a linear relationship of bias vs. n on a log scale.

For this type of simulation, one useful practice is to verify not just that the point estimates look right, but also that the uncertainty quantification is correct. Coverage of nominal 95% confidence intervals is a particularly demanding test: it requires both that the point estimate be close to unbiased and the standard error be well estimated. Showing close-to-nominal coverage in a clean setting can be a quite meaningful form of theoretical verification.

15.3.1.2 Increase the readibility of your paper

Methods papers can be extremely technical, and the theory can be difficult to understand. If the notation is not well laid out (which in many cases it is not), even deciphering what a theorem is saying, or what an estimator actually is, can be difficult. The simulation section often has to get down to brass tacks, laying out the problem cleanly. Thus simulations can be an accessible entry point into your paper, providing some concrete examples of what the context for the estimators given are, what assumptions the estimators depend on, and so forth.

While we advocate making the rest of your paper readable, we also note that a good simulation can help add clarity to your overall project. Clean simulations

of this type are also useful for illustrating *how* the method works.

15.3.1.3 Demonstrate finite-sample relevance

Most statistical theory is asymptotic: as $n \to \infty$, the estimator is consistent; as the number of studies grows, the coverage of a confidence interval approaches 95%; as the sample size grows, some test statistic converges to a chi-squared distribution. Such asymptotic results might say little about whether the method works at the sample sizes anyone would actually encounter. Simulations, on the other hand, can show that at some reasonable n, things are in fact close to what theory promises (for at least the contexts explored). For an alarming example of the perils of "asymptopia," Kang et al. (2018) examines weak instruments for Instrumental Variable analysis. Even though asymptotic IV estimators guarantee 95% coverage *eventually*, Kang et al. find coverage below 50%!

If you have derived a new estimator and proved it is consistent and asymptotically normal, a simulation showing that at $n = 500$ the bias is small, the sampling distribution is approximately normal, and the confidence intervals achieve close to nominal coverage could be reassuring to readers in a way that the asymptotic proof alone might not be. Readers are (rightly) skeptical of novel methods, and seeing the theory confirmed in a clean, interpretable simulation builds trust.

A second major purpose of simulation is therefore to show that the method works— or to characterize how well it works—in the finite-sample conditions that are realistic for the problem. This is where the multi-factor simulation framework developed earlier becomes essential. Finite-sample performance depends on more than just n. It depends on signal-to-noise ratios, covariate distributions, the complexity of the true outcome model, the degree of model misspecification, and a host of other features. A convincing finite-sample simulation should vary these features systematically, using levels that span the realistic range of what practitioners in the domain are likely to encounter.

This requirement does put a burden on the researcher to understand their domain. What are typical sample sizes in your field? What kinds of covariate structures are common? How much model misspecification should one worry about? The simulation factors you choose, and the levels you assign to them, are an implicit argument about what a domain looks like.

15.3.1.4 Motivate and contextualize the subsequent applied analysis

A methods paper often includes an applied data analysis, usually to show that the methods work on real data and to make an appeal that the methods designed are not completely useless or irrelevant. Simulations can play a supporting role here. If your applied analysis shows that methods A and B gave

different answers, then you might need your simulation section to make the case that we should believe A and not B.

Such a case can be best built via what one might call a *domain-calibrated simulation*, distinct from the multifactor simulation. Where the multifactor simulation explores many conditions to establish (ideally) general properties, a domain-calibrated simulation is specifically designed to mimic the applied dataset in the paper. It is built using the methods discussed above: the covariate distribution migth be drawn from the actual data, the outcome model might be fit to the actual data, and key features of the real setting (sample size, proportion treated, etc.) would all be matched.

The purpose of this calibrated simulation is not to add statistical generality—it may add very little—but to argue that, in a context that looks a lot like the problem we care about, we should prefer one method over another. This is especially true if the multifactor simulation shows that in some cases we might prefer the other method, or if in some cases there is no real difference between the two.

A natural structure for a methods paper, then, is to present the multifactor simulation first to establish a general story, and then follow it with a calibrated simulation tied to the applied data.

15.3.2 Navigating the many design decisions

We next discuss several decisions one needs to make as they design their simulation.

15.3.2.1 What performance metrics?

The choice of what to measure in a simulation is not neutral. Different metrics emphasize different aspects of performance, and a method that looks good on one metric can look bad on another. Bias measures systematic error; variance measures precision; mean squared error combines them. Coverage of confidence intervals assesses the full inferential pipeline. Type I error rate assesses validity of tests under the null. Power assesses sensitivity. Computational time is sometimes also relevant.

For a given paper, some of these matter more than others. If you are proposing a new estimator meant to correct a bias problem, bias is obviously central. If you are proposing a new test for a hypothesis where existing tests are known to be anti-conservative, Type I error rate is central. Think carefully about which metrics are directly relevant to the specific claim your paper is making, and report those prominently. Do not, of course, report only favorable metrics. Do report a more full range of metrics in your appendix, if you are only presenting a few in the main paper.

Do not focus overly narrowly on your primary metric, however. For example,

methods that controls bias tightly can often have much larger variances. If you are introducing a new method that controls bias, you should first show bias reduction, but also show that your method achieves a favorable bias-variance trade-off.

15.3.2.2 How many replications?

Too few replications and your simulation results will be noisy—a reviewer may reasonably wonder whether your conclusions hold up, or whether you got lucky. Too many and you waste computational resources that could have been spent on more scenarios or factors.

In your paper, you should compute and report *Monte Carlo standard errors* for your key performance metrics. You can even use the MCSE formula to calculate target iterations. For example, the Monte Carlo standard error for an estimated coverage probability $\hat{p}$ based on B replications is approximately $\sqrt{\hat{p}(1-\hat{p})/B}$. For a nominal 95% confidence interval, this means at $B = 1000$ replications, the Monte Carlo SE is about 0.7 percentage points—precise enough to distinguish coverage of 93% from 95% but not to distinguish 94% from 95%. If that level of precision is important for your argument, you need more replications. Reporting these Monte Carlo SEs, or at minimum thinking carefully about them when choosing B, signals methodological care to readers.

All of this said, as a rule of thumb, $B = 1000$ is often good enough.

15.3.2.3 How many factors?

Start with a few factors, run $R = 10$ or so iterations, and time your code to see how long your simulation runs (see Section A.3 for how). If your simulation runs quickly, you can add more factors in addition to scaling up the number of replications, and you probably should.

15.3.3 Principles of design and presentation

We next dive into some suggested principles.

15.3.3.1 Don't make your DGP stupid

All too often, the simulation section of a methods paper uses a DGP that is so extreme it is hard to image how it can be useful for anything other than a quick check of theory. In reading papers, watch for exponential functions, extremely small amounts of noise relative to the variation of the covariates, or other features that make the DGP unrealistic.

Do not do this, if you do not have good reason.

15.3.3.2 Figures, not Tables

As we discussed extensively in prior chapters, use figures, not tables. Figures allow you to show trends (as something goes up, something else goes down, and so forth). Tables do not make trends apparent.

That said, including all your raw numbers in an excel table or csv file as supplemental material is good for transparency. You might also have the raw values in a supplement, if you think people may be interested in exactly what those numbers are.

15.3.3.3 Avoid selective reporting

Simulations are particularly susceptible to selective reporting. The combination of degrees of freedom available to the researcher and the pressure to present results favorably creates a real risk of findings that do not paint an accurate picture.

There are three ways to defend against selective reporting.

The first is advanced planning. Before running your main set of simulations, think through the full set of factors you will vary and commit to a range of levels for each. If you deviate from this plan—because a factor turned out not to matter, or because a computational constraint forced a change—be transparent about why. Full factorial designs, with all combinations of all levels, are preferable to hand-selected sets of scenarios precisely because they leave less room for post-hoc selection.

The second is code sharing. Code sharing is increasingly expected in many fields, and for simulation studies it is particularly valuable. Sharing the simulation code allows readers to verify results, extend the simulation to additional conditions, or reproduce your findings if they are skeptical. Beyond serving the field, code sharing also disciplines the researcher: knowing that others could read and run the code tends to make one more careful and more honest. In practice, this means keeping simulation code clean and well-documented, something that the software engineering practices discussed earlier in this book will help facilitate.

The third is to find contexts where your methods fail. If you present simulations that include boundaries where your methods fail, then you are giving a more complex picture of the landscape of performance. You also gain credibility: by showing areas where things fall apart, you are signaling that you are not just cherry picking favorable results.

15.3.3.4 Tell a coherent story

The best simulation sections should not be a list of very specific findings. This, for example, is poor:

> In scenario A, method X had an RMSE of 0.34, and method Y had an RMSE of 0.23, so Y was preferred, while in scenario B, method X's RMSE rose to 0.44 and Y rose to 0.52, which was higher. In scenario C, X's RMSE rose again to 0.55, higher than Y's 0.45, and in scenario D, X's RMSE was 0.40, lower than Y's 0.42.

This is better:

> Results were mixed: Method X outperformed Y in scenarios B and D, while Y outperformed X in scenarios A and C.

And even better:

> In Scenarios B and D, which had a nonlinear trend, method X outperformed Y, but Y was superior when the trend was linear.

In the discussion you then might have:

> Earlier we saw that method X outperformed Y when the trend was nonlinear, but Y outperformed X when the trend was linear. This is consistent with the fact that method X is designed to capture nonlinear trends: it has a stability cost (note the higher SEs on Table Q), but that cost is worth it if there is something to correct for. We generated our nonlinear trend based on our fit model to the applied example discussed in the next section; we therfore view the risk of increased bias in practical settings as quite real, and thus recommend using X over Y in practice, without strong reason to believe a linear trend.

Some people try to report all the results, without interpretation, and then have the discussion after. Under this model, the reporting part can be extremely difficult to write, and extremely painful and boring to read. Keep it short and refer to the tables and figures for the details. Do not just list all those numbers without any contextualization.

In fact, if you can get away with it, just present the figures and tables, and dive right into interpretation as you explain how to read those tables.

E.g., continuing the running example from above:

> Table Q1 shows method X, with the bias correction, outperforms Y in the "nonlinear" scenarios, but Y outperforms when the trend is in fact linear. The higher RMSE in the linear scenarios is a stability cost (also note the higher SEs on Table Q2), but that cost is worth it if there is something to correct for. We generated our nonlinear trend based on our fit model to the applied example discussed in the next section; we therfore view the risk of increased bias in practical settings as quite real, and thus recommend using X over Y in practice, without strong reason to believe a linear trend.

Overall, make an argument. Give the reader your best understanding of why the proposed method works, when it works, how much it matters, and when

one should or should not worry.

Your visualizations should be selected to support your argument. In the above, connect your linear scenarios with a line, and the non-linear ones with a different line, to make it easy to see the pattern. Curate your visualizations to show the findings that matter, and put the remainder in the supplement for transparency.

15.3.3.5 Give summaries, not details, in the main text

Extending the above, in the main text you might have summaries of your performance. For example, in a large multifactor simulation of hundreds of scenarios, you might say

> Across the scenarios considered, the RMSE of method X ranged from 0.24 to 0.64. The other methods generally had higher RMSEs, ranging from 0.32 to 0.81. That said, in 15% of the scenarios explored, method X's RMSEs were more than 20% larger than the overall average, well beyond what could be explained by the Monte Carlo uncertainty.

15.3.3.6 Keep the main text simple

Not everything should be in the main text. It is entirely appropriate to put secondary results—additional factor levels, supplementary metrics, robustness checks—in an appendix or supplementary materials. What belongs in the main text are the figures that most directly supports the central claim of the paper, presented as clearly and compactly as possible.

15.3.3.7 Report MCSEs, but do it concisely

Including error bars from MCSEs on your plots can increase clutter substantially. Instead consider putting the range of MCSEs in the caption of your plot.

In your text, do not report differences between methods that are smaller than the MCSEs you are calculating. If arguing equivalence, say that your simulations show the methods are about the same, but could differ by about twice your MCSE (getting a true MCSE on the difference can be surprisingly tricky as the estimated performance measures of different methods will be correlated; see Section 14.4 for further discussion).

15.4 Concluding Thoughts

For a methods paper, simulations should generally serve at least three purposes: verifying that the theory is correct in clean conditions, demonstrating

that the method works in finite samples at realistic scales for the domain, and motivating the applied analysis by showing how the method performs in a context that resembles the real application. It should use performance metrics that are directly relevant to the paper's central claims, report enough replications to make the results trustworthy, and be presented in a way that tells a coherent story rather than merely exhausting the reader with numbers and details. Getting all of this right is not easy, but it is what separates a simulation study that genuinely advances knowledge from one that merely fills space between the theory section and the data analysis.

Simulations generally have a bad reputation, and that is unfortunately deserved. When reading methods papers, keep an eye out for extreme simulations that have no connection to reality, simulations that only report on one metric, leaving other ones unstated, and reported measures where it is impossible to tell if a difference is substantively meaningful or not. These examples are all too easy to find. Please do not increase the pool, in this regard.

Part IV

Computational Considerations

16

Organizing a simulation project

Multi-factor simulations can easily grow into quite complex projects with many moving pieces. As a project grows in complexity, the steady accretion of functions, scripts, and stored results can become overwhelming—to the point that you would dread sharing these work products with anyone. Fortunately, a bit of forethought and advance planning can make these types of projects much more tractable and manageable.

Before we get started on the content, we strongly recommend using an integrated development environment such as RStudio, Positron, or VS Code to manage projects and develop simulations. These Integrated Development Environments (IDEs) not only provide editors that understand the code you are writing (and therefore provide various enhancements such as syntax formatting and checking), but also help manage files and other aspects of a more compelx project. As these are now nearly ubiquitous, we do not discuss them further; we do, however, assume that you are using one as we discuss the techniques that follow.

In the next several chapters, we describe some organizing principles and programming practices that will make it easier to handle complex simulation projects. In this chapter, we discuss project organization, file management, and storing simulation results. In Chapter 17, we introduce some basic debugging and testing techniques that we have found useful for keeping on top of complex code bases. Finally, in Chapter 18, we demonstrate parallel processing methods that will help you to speed up the computation involved in multifactor simulations.

16.1 Simulation project structure

In a very simple simulation that does not take long to execute, you might keep all of the code, output, and analysis in a single file. However, if any of the computations take substantial time to execute, keeping everything in one file will become unwieldy. For more complex or more computationally demanding simulations, there are big benefits to organizing your code across more than

one file.

Multifactor simulation studies usually involve three distinct phases of work, each involving a different set of processes and different types of code. The first phase involves developing functions for generating data, applying estimators or data analysis procedures to the data, calculating performance measures, and running the full simulation for a single scenario. The end-product of this phase is a set of functions or methods for executing a simulation. The second phase involves running the simulations across multiple scenarios, as discussed in Chapter 10. The end-product of this phase is a set of results, containing estimates of performance measures for one or more methods under each condition. The third phase involves analyzing the simulation results and drawing conclusions about the performance of the method or methods under evaluation. The end-product of this phase is often a set of figures, tables, and text (perhaps taking the form of a memo, blog post, slide deck, or manuscript) that summarizes what you find.

To stay organized when conducting a larger, multifactor simulation, we find it generally useful to keep a clear separation between phases. In practice, this means keeping the code for each phase in a separate file (or set of files) and storing the results from each phase so that code for subsequent phases can be re-run or revised without re-executing the entire code base. In particular, dividing your simulations in this way allows for easily changing how one *analyzes* an experiment without re-running the entire thing. The modular approach we propose is in keeping with a general principle for organizing large computational projects: keep the code for distinct tasks in different files.

In this chapter, we describe strategies for such organization. We start by offering guidance and recommendations for how to structure individual files and how to make use of code that is organized across more than one file. We then describe a directory structure that encourages a clear separation between the phases of a project. Finally, we discuss strategies for storing and organizing results when executing a simulation. The advice and recommendations we offer are drawn from our own experience working on large simulation projects (including the many mistakes we have made and lessons we have learned through trial-and-error). Of course, we readily acknowledge that the approach we describe here is not the only way to do things, and we make no claims of optimality. As you gain experience with your own work, you will develop your own systems, habits, and strategies. We offer this guidance as a starting point from which to build.

16.2 Well structured code files

Conducting a simulation will require writing, testing, and revising code (potentially quite a bit of code!). There are two main types of files for holding R code; plain `.R` scripts and R markdown (`.Rmd`) or Quarto (`.qmd`) notebooks. The former can only contain R syntax and comments, whereas the latter can hold a mix of R code chunks and written text. The latter work as the source code for creating reproducible reports. Compiling a notebook will automatically run the embedded code and interweave the results with the text, producing a formatted document that can display figures, tables, and other forms of output.

For very small projects, it may be possible to store all of your functions and code for creating, executing and analyzing your simulation. An advantage of using a notebook here is that you can easily mix in plain text explanations and descriptions of the component pieces with the actual source code, making all rationales and design decisions fully documented. Further, you can re-run the entire simulation at the click of a button, simply by re-compiling the notebook. However, it is difficult to work this way if the simulation involves more intensive computation.

Larger or more computationally demanding projects, by contrast, will make more use of the first type of file, the humble `.R` script. Ideally, these scripts should follow the principles of modular programming, with each `.R` file holding a collection of code relevant to a single task. Accordingly, we distinguish between two types of `.R` scripts: "function library" scripts and scripts that carry out numerical calculations. The "function library" scripts just hold functions. The only consequence of running them from top to bottom will be to load some new functions into your workspace. This makes them easy to use in a modular form. The latter type are traditional scripts that do calculations, store or load data files, and create graphs and other summaries of data. Sourcing them will actually do work.

16.2.1 Putting headers in your .R file

When you write an `.R` script, it is a good idea to put a header at the top of the file that describes the file's purpose. Within the file, you can also describe the contents with shorter section headers. For instance, a very simple section header might look like this:

```
#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#
# Data generating functions ----
#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#
```

Note the `----` at the end of the middle line. The four trailing dashes trailing

dashes tells RStudio's parser that the given line is a heading line that organizes the file. If you click on the dropdown at the bottom of the RStudio source pane, you will see a pop-up table of contents listing all of these heading lines along with all defined functions. You can use it to quickly navigate to different parts of your source file. You can also see the same table of contents by clicking the `Outline` button in the top right corner of the source pane.

16.2.2 The source command

If your code is organized across more than one file, you will need a way to call one file from another. This is the purpose of the `source()` command, which effectively cuts-and-pastes the contents of the named file into your R session. For example, the following code runs three separate R scripts in turn:

```
source( here::here( "R/data_generators.R" ) )
source( here::here( "R/estimators.R" ) )
source( here::here( "R/simulation_support.R" ) )
```

If the named file has code to run, it will run it. If the named file contains a set of functions, those functions will now be available for use. The `here::here()` command is a convenience function that allows you to specify a file path relative to your R project root directory, so you can easily find your files.[1]

The `source()` command is simple and literal—it runs a specified script from top to bottom, regardless of the contents. In particular, if the sourced script includes further calls of `source()`, the referenced code will be run. For example, the `simulation_support.R` script could include calls to source the other two files. In this case, you would only need to source the single `simulation_support.R` file to load the contents of all three files. Although this might seem convenient, it does create some constraints as well. With this structure, there would be no way to source the contents of `simulation_support.R` without also sourcing `data_generators.R` and `estimators.R`. For this reason, we tend to avoid using `source()` in scripts that might themselves be sourced.

One reason for storing code in individual files and using `source()` is that you can also then include testing code in each of your files to both demonstrate the syntax and check the correctness of your functions (see below). Then, when you are not focused on that particular function or component, you do

[1] If you are used to using RStudio projects, you might wonder why you should use `here::here()`. If you always open your project from the `.Rproj` file, then the working directory will always start out set to the root directory, but it will not necessarily remain so (if a script calls `setwd()`, for instance). Further, when `.Rmd` or `.qmd` notebooks are compiled, the working directory is temporarily changed to the location of the notebook, so relative file references will work differently than in plain `.R` scripts. Wrapping file names in `here::here()` ensures that they will always be evaluated relative to the root directory of the project.

not have to look at that testing code. Another good reason for this type of modular organization is that is supports developing a broader simulation universe, potentially with a variety of data generating functions. With a whole library of options, you can then readily run multiple simulations that involve some different components and some shared features.

We followed this approach in one recent simulation project examining estimators for an instrumental variable analysis, a common approach to handling non-compliance in randomized experiments. In this project, we wrote several different data generating functions that implemented different types of non-compliance patterns. Our `data_generators.R` code file then had several methods. When we sourced it, we end up with the following methods available to call:

```
> 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 main methods in the file are for a set of different data generating methods, each developed over the course of the project as we chased down errors in our estimators and worked to understand strange behavior. The `describe()` and `summarize()` methods print summary statistics for a given dataset; we used these to explore and debug our different data generating functions.

We also had a separate file, `estimators.R`, that contained our different estimators we were comparing. Having our estimators in a separate file made it easy to use them for other purposes, such as our empirical data analysis. We could simply source the `estimators.R` file and then apply them to a real dataset. Having a single file ensured that our simulation and applied analysis were exactly aligned in terms of the estimators we were using. After we debugged or tweaked our estimators, we could immediately re-run our applied analysis to update the results.

16.2.3 Storing testing code in your scripts

If you have an extended `.R` file with a collection of functions, you might also want to store some 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:

```
# My testing code ----
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
}
```

When you open and look at this script, you can then paste the code inside the block into the console when you want to run it. However, if you `source()` the script, the `FALSE` block will not run at all, so you can avoid extraneous output or computations that are not actually needed. This is a good way to keep simple demo code or testing code in close proximity to the function it is testing. When you want to work on just the part of the project captured by your script, you can work inside the single file very easily, ignoring the other parts of your project.

In previous chapters, we have emphasized the importance of writing code to validate that your functions work as intended. Sometimes, such validation code will involve more than just a quick call to the function. In such cases, it can be beneficial to organize validation code in its own file, separate from the function or functions that it tests. You would then need to `source()` the files containing the function to be tested. This approach does have the drawback that the testing code is not in close proximity to the function to be tested, so you would need to work across multiple files when developing and running the test code. However, unlike with the `FALSE` blocks, organizing your test code in a separate file makes it easier to re-run a collection of several tests. This approach to organizing becomes more appealing when the validation code involves calls to functions that are themselves stored in separate files or when it involves invoking packages that are not needed for the main calculations involved in the simulation. It is also closer to how you might work when developing code for an R package, with unit tests that check the correctness of critical functions. We discuss unit testing further in Section 17.5.

16.3 Principled directory structures

Tidy home, tidy mind, as the saying goes. With any computational project, our home is usually a directory on our computer (or in a cloud). For multi-factor simulation studies, we *strongly* advocate using an organized and clearly labelled directory structure, which will facilitate more intentional—and easier to follow—coding practices. We recommend building a directory structure as follows:

```
my_project/
  proj.Rproj
```

```
README.md
R/
test/
data/
scripts/
results/
```

If you have ever looked at the source code for or developed your own R package, this structure will look familiar.

- The `R/` directory is where to put the core R code for your simulation. It should contain one or more `.R` scripts that hold the main functions for implementing a tidy simulation, including data-generating functions, analysis functions, performance calculation functions, and a simulation driver that pulls all the pieces together. With all of these functions saved in the same directory, you can then `source()` the scripts as needed to gain access to your functions. You *should not* put the scripts that actually execute the simulation or analyze the results in this folder. Instead, the `R/` folder is reserved for the building blocks of the simulation.
- The `test/` directory is where to put any code for testing the functions and methods you have developed. You could even write formal unit tests for your methods using a testing framework such as the `testthat` package, and put those here.
- The `data/` directory is for any data files used in executing the simulation. For instance, a simulation of a cluster-randomized experiment might make use of an empirical dataset containing features such as cluster sizes or covariate values, which are sampled when generating artificial datasets. This directory is reserved for *source* data, which may be read in as part of the simulation. It should not contain any files that are *created* as part of executing the simulation.
- The `scripts/` directory is where to put scripts that actually run the simulations and analyze simulation results. Some of these scripts will make use of functions saved in the `R/` directory. You will likely have at least one script that runs the simulation and at least one script for analyzing the results.
- The `results/` directory is where to save any generated results of your simulation. This directory should *only* contain files that have been created by running code in the `scripts/` directory. Sometimes it might be worth having separate directories `raw_results` and `results`, where `raw_results` holds output created from running the simulation driver, and `results` holds final results produced from merging and summarizing the raw results.

Using this structure will help you to maintain reproducibility of the full project. If any files in `data/`, `R/`, or `scripts/` are changed, then some or all of the files in `results/` may need to be updated. In principle, it should be possible

to delete all of the files in `results/` and recreate them in full by re-running the files in the `scripts/` directory.

16.4 Saving simulation results

Multifactor simulations can be time-consuming to run, and deciding how to analyze raw simulation results often involves several rounds of iteration and refinement. Because of this, it pays to save raw simulation results so that they can be accessed and analyzed without having to re-run the code that generated them. As a bare minimum for any multifactor simulation, we recommend saving the complete set of results to a file. In many instances, it may make sense to go further by saving results for each unique condition as soon as it is complete. The latter approach provides an additional level of protection in the event that you need to re-run parts of the simulation or if the code crashes out in rare circumstances or for only a subset of the conditions. By saving as you go, you can keep the results as they come, and thus not have to restart if something goes wrong.

16.4.1 File formats

There are two primary candidates for saving simulation results: a comma-separated value file (a `.csv` file) or an R-specific `.rds` file.

A comma-seperated value file is a plain text file where each row of data is printed out as a series of values separated by commas. To make one, after your simulation has completed, you would save as so:

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

This will make (or overwrite) a file, `simulation_CRT.csv`, in the `results` folder of your project that holds all the information in `res`. The `write_csv()` function comes from the `readr` package, which is part of the `tidyverse` suite. It takes a dataset and a file name as input and creates a comma-separated value file containing the contents of the dataset.[2]

Once stored, load the dataset back into R for analysis using `read_csv()`:

```
res <- read_csv( "results/simulation_CRT.csv" )
```

A key advantage of storing results as `.csv` is that it is a general format, so the file can be read by others even if they are not familiar with R. The format

[2] See Section 11.5 of the *R for Data Science* textbook (Wickham et al., 2023) for further details.

also makes for convenient viewing in a spreadsheet program. However, storing results in this format will only work for plain, rectangular datasets with no special features (such as list-columns). Further, saving a dataset as `.csv` and then reading it back into R results in a loss of meta-data, such as variable labels and column data-types.[3]

A second option is to use the `saveRDS()` and `readRDS()` methods. For instance, you could save a dataset containing simulation results in the file `results/simulation_CRT.rds` like so:

```
dir.create("results", showWarnings = FALSE)
saveRDS( res, "results/simulation_CRT.rds" )
```

The `saveRDS()` command is a base R function that takes an R object and a file name as input and creates a file that stores the object in a serialized (usually binary) format. Once stored, you can load the object back into R for analysis using `readRDS()`:

```
res <- readRDS( "results/simulation_CRT.csv" )
```

One major advantage of storing results in `.rds` format is that they can be reloaded in the exact same form in which they were saved. There is no conversion between file formats, and so no loss of meta-data about variable formats, labels, or the like. Another advantage is that `.rds` can hold *any* type of R object—whether it be a dataset, a function, a fitted model. Because of this, it can be used to save tibbles even if they included nested listed-columns or other more exotic structures. The main disadvantage is that the contents of the file cannot be interpreted without reading it back into R.

16.4.2 Saving simulations as you go

For some projects involving multifactor simulations with multiple data-generating conditions, it can be useful to save results from each condition in its own file. This is a prudent step to take if you are not sure you will have time to run the entire simulation, if you are worried that your R session might crash half way through because of an error, or if you are trying to debug an error that only occurs under certain conditions. This approach to saving results is also effective when running simulations in parallel, as we will discuss in Chapter 18. With a bit of further machinery, it also makes it possible to selectively delete files and rerun only parts of a larger simulation.[4]

We will illustrate this approach to storing results by revisiting the simula-

[3] For example, a variable that was initially created with `factor()` will be converted into a plain character vector, and the ordering of the factor levels will be lost.

[4] Of course this requires some care to ensure everything is still reproducible. Before publishing final results from the simulation, always rerun everything from scratch to ensure everything is up to date and to avoid potentially embarrassing errors.

tion study on confidence intervals for Pearson's correlation coefficient under a bivariate Poisson distribution, building on our development in Chapter 10. This simulation involved four different factors, each with multiple levels, with data-generating conditions created as follows:

```
params <- expand_grid(
  N = c(10, 20, 30),
  mu1 = c(4, 8, 12),
  lambda = c(0.5, 1.0),
  rho = seq(0.0, 0.7, 0.1)
) %>%
  mutate( mu2 = mu1 * lambda )
```

In Chapter 10, we used `bundle_sim()` to create a simulation driver for executing the simulation for a given scenario and then used `evaluate_by_row()` to call the simulation driver for every condition listed in `params`:

```
library(simhelpers)

Pearson_sim <- bundle_sim( f_generate = r_bivariate_Poisson,
                           f_analyze = r_and_z,
                           f_summarize = evaluate_CIs )

sim_results <- evaluate_by_row( params, Pearson_sim, reps = 100 )
```

To store the results from each condition in its own file, we need to modify the simulation driver to save its result to file rather than just returning it. One way to do so is to wrap `Pearson_sim()` inside another function:

```
Pearson_sim_save <- function(..., filename) {
  res <- safely(Pearson_sim)(...)
  if (is.null(res$error)) saveRDS(res$result, file = filename)
  return(res$error)
}
```

This function calls the simulation driver, `Pearson_sim()` using any arguments in the `...`. To prevent an error from stopping the entire simulation, we have wrapped the simulation driver in `safely()` to trap errors (see Section 7.4). As long as `Pearson_sim()` does not return an error, it saves the results from the simulation in a file with the user-specified `filename`. The function returns any errors as output so that it they can examined to figure out what went wrong.

To use `Pearson_sim_save()`, we first need to create a file name for every condition. We also create a unique seed for each condition, which will get passed to the simulation driver to ensure computational reproducibility. The unique seed also makes it easy to generate unique file names, so each simulation run will be kept separate. Once we have our scenarios, we call `evaluate_by_row()`

with `Pearson_sim_save()` as the simulation driver. Here is the full block of code:

```
params <-
  params %>%
  mutate(
    seed = 20260129 + row_number(),
    filename = paste0("results/pearson-sim/pearson-row",
                      seed, ".rds")
  ) %>%
  select(-lambda)

saveRDS( params, "results/pearson-sim/pearson-sim-params.rds" )

sim_errors <- evaluate_by_row(params, Pearson_sim_save, reps = 100 )
```

The above code creates a set of 144 files containing the simulation results from every condition. The results are saved inside our code, rather than after the simulation completes, as we saw before.

The object `sim_errors` will include a column containing the error messages from any conditions that did not complete. In addition to the simulation results, we save `params` as our "table of contents" that tracks our parameter settings for each condition and the corresponding file name for the results. In the event that some conditions produced errors or if we had to stop execution of the simulations before every condition was run, some of the results will still be saved, and we would only need to re-run the remaining conditions.

Rerunning a subset of conditions. To illustrate how to re-run a subset of results, suppose that we had only been able to run and save results for the first three conditions. The directory containing the simulation results will then only contain three files:

```
completed_rows <- list.files("results/pearson-sim", full.names = TRUE)
completed_rows
```

```
[1] "results/pearson-sim/pearson-row20260130.rds"
[2] "results/pearson-sim/pearson-row20260131.rds"
[3] "results/pearson-sim/pearson-row20260132.rds"
```

To avoid repeating the simulations for these conditions, we screen them out of `params` before running `evaluate_by_row()` again, as in:

```
params <- readRDS( "results/pearson-sim/pearson-sim-params.rds" )
sim_errors <-
  params %>%
  filter(!(filename %in% completed_rows)) %>%
  evaluate_by_row(Pearson_sim_save, reps = 100 )
```

This will run our simulation on anything *not* found in the directory.

Loading all the results. Once all the conditions are run, `results/pearson-sim` will have a set of files containing all the simulation results. The only thing left to do is read these files in and combine them into a single dataset. To do so, we will first get the names of the files with `list.files()` and check that everything is complete

```
completed_rows <- list.files("results/pearson-sim", full.names = TRUE)
setdiff(params$filename, completed_rows)
```

```
character(0)
```

We can then read in the files using `map()` to call `readRDS()` on each file:

```
sim_results <-
  params %>%
  # ensure we only read in completed conditions
  filter(filename %in% completed_rows) %>%
  # read in results
  mutate(
    res = map(filename, readRDS)
  ) %>%
  unnest(res) %>%
  select(-filename)
```

The dataset `sim_results` now has all the results from every condition evaluated, organized by stacking the fragments from each of the individual conditions. We use the `filename` variable stored in the list of simulation conditions so that the parameter settings of each condition are available in the same dataset as the results.

To avoid having to repeat the process of reading in all the files, we could finally save the full set of results in a separate `.rds` file:

```
saveRDS(sim_results, file = "results/pearson-simulation-all-results.rds")
```

Now we can simply load that file as we have for other simulations previously.

16.5 Summary

Few particularly love the project-management aspect of doing simulations, but it is an important part of the process. Keeping things organized, even if it feels like extra time along the way, can save a lot of time later on, especially for complex or collaborative projects.

17

Debugging and Testing

Writing code is usually not too difficult. Writing *correct* code, on the other hand, is harder. Over the course of developing a simulation, you will surely write some code that does not do what you expect—or even that does not work at all. Figuring out why code produces errors or incorrect results can be enormously frustrating and time-consuming. If you've written code that follows a modular design, you will often have functions that call other functions you wrote, which might call even more functions you wrote. When you get an error, it may not be immediately clear what went wrong or even where to look for an error. In this chapter, we introduce some basic techniques for diagnosing and preventing such errors.

17.1 Debugging with `print()`

A simple debugging method (often reviled by professional programmers, but still useful for more ordinary folks) is to make your code more verbose, so that it provides information about what was happening just before an error occurred. One approach to doing so is to add `print()` or `cat()` statements inside your functions. For example, consider the following code:

```
    if ( any( is.na( rs$estimate ) ) ) {
        cat( "There are NAs in the estimates!\n" )
    }
```

Here, we test whether there are `NA` values in the results stored in `rs$estimate`; if so, we print out a message to flag this condition.

There are a few methods for printing to the console. The first is `print()`, which takes any object and prints it out. You can print a variable, a string of text, or even a `tibble`:

```
my_var <- 5.3e7
print( paste("My var is ", my_var, "\n" ) )
print( my_tibble )
```

A second is `cat()`, which is designed to print strings:

```
cat( "My var is ", my_var, "\n" )
```

A further alternative is to use functions from the `cli` package, which produce a nicer printout and allow for easier formatting:

```
cli::cli_alert("My var is {my_var}")
```

A major problem with using printing to debug it that it is easy to go overboard, so that running your code produces a huge wall of text. Writing text to the console also takes computing time. With simulation code, it is easy to print so much that it will meaningfully slow your simulation down! Thus, if you use `print()` or `cat()` statements to help figure out what is going on with your code, it is best to be sparing and to remove most such statements from your code once you are satisfied that it works correctly.

17.2 Debugging with `debug()`

The great benefit of writing your own custom function for something is that it *automates* a sequence of steps, so that you do not have to write and run code for each step every time you want to execute the sequence. The drawback of doing so is that running the function *hides* the intermediate steps in the calculations, making it harder to tell if or where an error occurs. For instance, consider the data-generating function for the bivariate Poisson distribution, initially developed in Section 6.3. We run the function and get the following nonsense output:

```
r_bivariate_Poisson(N = 4, mu1 = 6, mu2 = 4, rho = -0.3)
```

```
  C1 C2
1 NA NA
2 NA NA
3 NA NA
4 NA NA
```

You might recall that this function does not work for negative correlations, so the problem is caused by setting `rho = -0.3`. Suppose, though, that we had forgotten this and need to figure out exactly what is going wrong.

R has several commands that allow for interactive debugging of functions. One useful technique is to use the `debug()` function, which tells R to interactively step through a particular function whenever it is called. Calling it on `r_bivariate_Poisson` and then re-running our line of code will let us step through the body of the function line by line:

```
debug(r_bivariate_Poisson)
r_bivariate_Poisson(N = 4, mu1 = 6, mu2 = 4, rho = -0.3)
```

This puts you into an interactive debugging console (indicated by `Browse[1]>` instead of the usual `>` console symbol) that allows you to inspect the current state of all of the objects and variables that R can access when running the function. This is just like a normal R workspace, but you will only see the objects that are passed to the function and the variables created inside the function. In RStudio, the source pane will usually jump to the part of your script that defines the function, so you can see the code that will be run in each step.

Once in this debugging console, you can type `n` to go to the next line of code.[1] Using `n` makes it possible to walk through the code step by step, observing what happens with each line of code. You can also type regular R commands in the debugging console to do calculations with the objects in the environment. Type `c` to exit from debugging mode and continue regular execution of the code or `Q` to exit debugging and stop execution.

The `debug()` command changes how `r_bivariate_Poisson()` is evaluated every time it is run. If your code calls this function multiple times, then it will pause and open the debugging console *every time* the function is called—even if the function is called by a different function. For example, in Section 8.4 we created a simulation driver function called `sim_r_Poisson()`, which calls `r_bivariate_Poisson()` to generate data and then calls an estimation function to compute a Pearson correlation. Try running the simulation driver as follows:

```
sim_r_Poisson( 3, N = 4, rho = -0.3, mu1 = 6, mu2 = 4 )
```

The code should pause and open the debugging console for `r_bivariate_Poisson()` for each replication of the simulation process. To stop this behavior, run `undebug()` on `r_bivariate_Poisson`:

```
undebug(r_bivariate_Poisson)
sim_r_Poisson( 3, N = 4, rho = -0.3, mu1 = 6, mu2 = 4 )
```

If you would like to pause and debug a function only the first time it is invoked, you can use `debugonce()`:

```
debugonce(r_bivariate_Poisson)
r_bivariate_Poisson(N = 4, mu1 = 6, mu2 = 4, rho = -0.3)
sim_r_Poisson( 3, N = 4, rho = -0.3, mu1 = 6, mu2 = 4 )
```

[1] In RStudio, the debugging interface also includes clickable buttons for this and other commands.

17.3 Debugging with `browser()`

Another useful debugging tool is the `browser()` command. Like `debug()`, it will open up an interactive debugging console so that you can inspect what is happening inside of a function. However, instead of pausing whenever a function is called, `browser()` pauses at the point that it is encountered in the code. This is helpful for longer functions, so that you do not need to step through every line of code to get to the point where problems start to occur. It also makes it possible to pause and debug *contingently*, only if certain conditions are met. For instance, we might add a `browser()` call to `r_bivariate_Poisson()` that will be executed only if there are missing values in the generated data:

```
r_bivariate_Poisson <- function(N, mu1, mu2, rho = 0) {

  # covariance term, equal to E(Z_3)
  EZ3 <- rho * sqrt(mu1 * mu2)

  # Generate independent components
  Z1 <- rpois(N, lambda = mu1 - EZ3)
  Z2 <- rpois(N, lambda = mu2 - EZ3)
  Z3 <- rpois(N, lambda = EZ3)

  # Assemble components
  dat <- data.frame(
    C1 = Z1 + Z3,
    C2 = Z2 + Z3
  )

  if (any(is.na(dat))) {
    browser()
  }

  return(dat)
}
```

Now when we call the function, we only enter the debugging console if there are missing values in the generated data. Here we would just get some data:

```
a <- r_bivariate_Poisson(N = 4, mu1 = 6, mu2 = 4, rho = 0.3)
```

Here it would enter the debugger:

```
b <- r_bivariate_Poisson(N = 4, mu1 = 6, mu2 = 4, rho = -0.3)
```

```
Called from: r_bivariate_Poisson(N = 4, mu1 = 6, mu2 = 4, rho = -0.3)
Browse[1]>
```

Once we hit a scenario with missing values, we can inspect all of the objects visible to the function and the intermediate variables created by the function. In this case, we could see that `Z3` contains missing values and that `EZ3` was negative, from which we can infer that the problem is the negative value for `rho`.

Using `browser()` is a useful way to make sure you understand what values are passed to a function. Many bugs arise due to the wrong thing getting passed to some code that would otherwise work. Triggering `browser()` when something bad happens (such as when a set of estimates includes unexpected missing values) can often help you to untangle what is driving an error — even one that occurs only rarely. Because of its versatility, adding `browser()` calls is often our first move when debugging our own simulation code.

17.4 Protecting functions with assertions

When writing functions, especially those that take a lot of parameters, it is often wise to include *assertions* to verify that the function's input arguments are as expected. For instance, in our `r_bivariate_Poisson()` data generating-function, we should check that `mu1` and `mu2` are both positive and that `rho` is non-negative. Adding such assertions is useful because it will make errors apparent and stop your function as early as possible.

In R, a simple way to write assertions is with the `stopifnot()` command. For example, we could add the following lines to the top of `r_bivariate_Poisson()`

```
r_bivariate_Poisson <- function(N, mu1, mu2, rho = 0) {

  stopifnot( N > 0 )
  stopifnot( mu1 > 0, mu2 > 0 )
  stopifnot( rho >= 0 )

  # remainder of function...
}
```

Now if we call the function with a negative correlation, we immediately get a relatively informative error:

```
r_bivariate_Poisson(N = 4, mu1 = 6, mu2 = 4, rho = -0.3)
```

```
Error in `r_bivariate_Poisson()`:
```

```
! rho >= 0 is not TRUE
```

Assertions can also protect us from silent errors that might arise from the wrong thing getting passed to a function. Consider the following function for 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 `make_groups()` using input vectors of different lengths for `means` and `sds`, the code executes without error because R recycles the standard deviation parameter:

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

```
[1]  101  259 8293  401
```

This sort of behavior could produce errors, and it is pernicious because there is no indication that something is wrong. Imagine building an entire simulation involving this function, not realizing that your fourth group has the variance of your first. You might spend a lot of time and computing power, only to be left scratching your head over results that make no sense. Even if you know something is wrong with your simulation, it might take some time and effort to track down the origin of problem.

Adding some simple assertions to the top of our function might let us avoid such hardship and suffering. The assertions should verify that the arguments to our function are as they should be, and, if they are not, stop the function in its tracks. In this case, we will check that the input arguments are vectors of the same length:

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

Now, if we call our function incorrectly, we get this:

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

```
Error in `make_groups()`:
! length(means) == length(sds) is not TRUE
```

The `stopifnot()` command throws an error to signal that the inputs to the function are not specified correctly.

Assertions can also serve as a type of documentation for valid inputs to a function. 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. The assertion serves as a good reminder of what the input parameter is for.

Assertions can also provide a layer of protection if you mis-specify the order of the input parameters when you call a function. Consider:

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

```
Error in `make_xy()`:
! -1 <= rho && rho <= 1 is not TRUE
```

Of course, it is good practice to use named inputs. Better to be safe and call the function as follows:

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

17.5 Unit testing

Throughout this book, we have emphasized the utility of writing separate functions for the different components of a simulation study, and checking those functions to ensure that they work properly and as expected. In previous chapters, we have seen several examples of how to check functions, such as when we made plots of data produced by a data-generating function to see if the distributions looked like we intended. This sort of code could be stored in the script alongside the functions being tested,[2] or in a separate file. In either case, it is useful to retain the code so that you can run it again later to verify that a function still works as expected.

These sorts of informal checks are very useful, but in our experience it is easy to let them fall by the wayside or fall apart as you continue to develop the code for a simulation. It can be hard to find motivation to go and re-run checks after making apparently trivial changes to your core code. Likewise, it can also be challenging to track down the ripple effects of changing a low-level function that is used by many other pieces of your code.

[2] To prevent execution of the code, the tests can be cordoned off using the `FALSE` trick discussed in Section 16.2.3.

Unit testing is a set of practices, commonly used in software development, that formalize the idea of writing code to verify that a function works. The spirit of unit testing is to write code to test your functions *thoroughly* and to run those tests *frequently* to verify that everything works as intended, even as you make changes to component functions. Unit testing frameworks allow you to write test code that, whenever you want, can be used to generate a report of which tests pass and which fail so you can zero in on what problems you might have with your functions.

The `testthat` package is a popular unit testing framework for R.[3] Although originally designed for unit testing of functions included as part of R packages, it is nonetheless quite useful for more general coding projects, such as developing the code base for a simulation study. There are two general parts to `testthat`: the `expect_*()` methods and the `test_that()` function.

Consider the following simple DGP to generate an X and Y variable with a specified association:

```
my_DGP <- function( N, mu, beta ) {
  stopifnot( N > 0, abs(beta) <= 1 )
  dat = tibble( X = rnorm( N, mean = 0, sd = 1 ),
                Y = mu + beta * X + rnorm( N, sd = 1-beta^2 ) )
}
```

Here we run an initial set of unit tests for the `my_DGP()` function. We get a nice happy message for each test:

```
library(testthat)
set.seed(44343)
test_that("my_DGP produces output of correct form and dimension.", {

  dta <- my_DGP(10, 0, 0.5)

  # Check that the output is a tibble
  expect_s3_class( dta, "tbl_df" )

  # Check that the output has the right number of rows
  expect_equal( nrow(dta), 10 )

  # Check that the output has the right columns
  expect_true( all(c("X", "Y") %in% colnames(dta)) )
})
```

```
Test passed with 3 successes.
```

[3] See Chapters 13 through 15 of Wickham and Bryan (2023) for an in-depth discussion of unit testing in the context of R package development

```
test_that("my_DGP returns errors with incorrect inputs.", {

  # Check we get an error when we should
  expect_error( my_DGP(-10, 0, 0.5) )
  expect_error( my_DGP(10, 1, 1.5) )
  expect_error( my_DGP(10, 0.4, 0.5, 0.3) )

})
```

```
Test passed with 3 successes.
```

If one or more of our tests fail, we will get a set of error messages that tells us what went wrong, and where things broke:

```
test_that("my_DGP produces data with expected features.", {

  # Check that the mean of Y is close to mu
  dta2 = my_DGP(1000, 2, 0.5)
  expect_equal(mean(dta2$Y), 2, tolerance = 0.1)


  # Check that the SDs of X and Y are close to 1
  dta <- my_DGP(10000, 2, 0.2)
  expect_equal( sd(dta$X), 1, tolerance = 0.02 )
  expect_equal( sd(dta$Y), 1, tolerance = 0.02 )

  # Check that the regression slope is close to beta
  M <- lm( Y ~ X, data=dta )
  expect_equal( coef(M)[[2]], 0.5, tolerance = 0.02 )
} )
```

```
-- Failure: my_DGP produces data with expected features. -------
Expected `coef(M)[[2]]` to equal 0.5.
Differences:
1/1 mismatches
[1] 0.196 - 0.5 == -0.304

Error:
! Test failed with 1 failure and 3 successes.
```

A thorough set of unit tests will target multiple aspects of a function's behavior—not just that it produces output of the correct form, but also that the output is consistent with the input specifications, that it produces errors when given incorrect inputs, and that it works under a range of scenarios or contingencies that might arise in your simulation study. In principle, if you write any code to figure out why something is not working as expected, you should turn that code into a unit test so that you can run it again later,

ensuring that any bug you fixed will stay fixed moving forward.

If you have written a set of tests and collected them in a file, you can run them all at once:

```
test_file(here::here( "code/demo_test_file.R" ) )
```

```
== Testing demo_test_file.R ================================

[ FAIL 0 | WARN 0 | SKIP 0 | PASS 0 ]
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 1 ]
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 2 ]
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 3 ]
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 4 ]
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 5 ]
[ FAIL 0 | WARN 1 | SKIP 0 | PASS 5 ]
[ FAIL 0 | WARN 1 | SKIP 0 | PASS 6 ]
[ FAIL 1 | WARN 1 | SKIP 0 | PASS 6 ]
[ FAIL 1 | WARN 1 | SKIP 0 | PASS 7 ]

-- Warning ('demo_test_file.R:35:3'): my_DGP works as expected (test 2) --
NAs produced
Backtrace:
    x
 1. +-my_DGP(10000, 2, -2) at demo_test_file.R:35:3
 2. | \-tibble::tibble(...) at demo_test_file.R:7:3
 3. |   \-tibble:::tibble_quos(xs, .rows, .name_repair)
 4. |     \-rlang::eval_tidy(xs[[j]], mask)
 5. \-stats::rnorm(N, sd = 1 - beta^2)

-- Failure ('demo_test_file.R:39:3'): my_DGP works as expected (test 2) --
Expected `var(dta$Y)` to equal 1.
Differences:
1/1 mismatches
[1] 0.792 - 1 == -0.208

[ FAIL 1 | WARN 1 | SKIP 0 | PASS 7 ]
```

The `test_file()` method will produce a printout reporting which of the tests passed, which gave warnings, and which were skipped.[4] Tests are run "from scratch" in a fresh R session. Because `test_file()` runs the tests in a separate session, you will have to make sure that the testing script loads any needed

[4] You can also run the tests from inside RStudio. When in a unit test file, you should see "Run Tests" at the top-right of the script pane. If you click on it, RStudio will start an entirely new work session, and source your file to test it.

libraries and creates any objects needed for the unit tests to execute. If your unit tests are testing your core (stored in the `R/` folder of your project, see Chapter 16 for further discussion), you need to source the relevant files at the top of your test file.

If you have written tests for several different functions, you might store the results in several files. You can run all of your test files in sequence using `test_dir()`. For instance, if you store all of your testing files in `tests/`, then run

```
test_dir("tests")
```

Once you have created a set of unit tests, you can continue to work on your project and tweak your functions. If you make changes to any function, you can re-run *all* the unit tests to see if you broke anything. Perhaps even more important, if you are working with a collaborator, you can both run unit tests to ensure you have not broken something that someone else was counting on! Furthermore, you can use the test code as a quick reference for how your functions should be used and what their expected output should look like. For any reasonably complex project, having test code can be of enormous benefit.

18

Parallel Processing

Especially if you take our advice of "when in doubt, go more general" and if are running enough replicates to get nice and samll Monte Carlo errors, you will quickly come up against the computational limits of your computer. Simulations can be incredibly computationally intensive, and there are a few means for dealing with that. The first is to optimize code by removing extraneous calculation (e.g., by writing methods from scratch rather than using the safety-checking and thus sometimes slower methods in R, or by saving calculations that are shared across different estimation approaches) to make it run faster. This approach is usually quite hard, and the benefits often minimal; see Appendix Section A.4 for further discussion and examples. The second is to use more computing power by making the simulation *parallel*. This latter approach is the topic of this chapter.

Parallel computation is where you have multiple computers working on a problem at the same time. Simulations are very easy to do in parallel. With a multifactor experiment it is very easy to break apart the overall into pieces. For example, each computer could do one of your scenarios in a multi-factor simulation. Even without a multifactor experiment, due to the cycle of "generate data, then analyze," it is easy to have a bunch of computers doing the same thing, as long as they are generating different datasets, with a final collection step where all the individual iterations are combined into a single set at the end. In either case, the *total* work is the same, but since each computer is doing a piece at the same time, the overall simulation will be completed much sooner.

There are three general ways to do parallel calculation. The first is to take advantage of the fact that most modern computers come with multiple cores (i.e., computers) built in. For this simplest approach, we simply tell R to use more of this processing power. 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, putting your simulation on a virtual computer that might have 50 or more cores. Other than having to handle moving files back and forth from your machine to the cloud, this is not much different from the first approach: you are still just telling R to use more cores, but now you have more cores to use. Even if the speed-up itself is not massive,

this approach also 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.

The third way is to use a computing cluster. A cluster is a network of hundreds or thousands of computers, coupled with commands for breaking apart a simulation into pieces and sending the pieces to that vast army. Conceptually, this approach 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 a cluster up and running, it can be a very powerful tool. Not only is it off your personal computer, a cluster can potentially give you hundreds of cores, which means a speed-up of hundreds rather than four or eight.

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] 4
```

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 **?@sec-exp_design** (see **?@sec-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 do not do some handholding regarding how it should set seeds for pseudoranom number generation. In particular, if you do not 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 on a virtual machine

Your laptop probably has around 8 cores, meaning you can have an 8 fold speed-up. But wouldn't it be nice to have a computer with 50 cores? Or even more? You can get one!

Cloud services, such as Amazon Web Services (AWS), can give you a *virtual machine* that can have many cores (where many is usually 50 or so). Some

of these services give you what is effectively an R Studio session, and so you can run your scripts and everything just like we have discussed above, with no change.

The Data Colada blog advocates for an alternative to AWS, which is [Kamatera] (https://www.kamatera.com). The Data Colada folks say "it is much easier to use, way faster to set up, and flexible (e.g., you can easily update R and R Studio in it)." See Data Colada's post for more information.

This kind of cloud computing is relatively straightforward, once you understand parallel on your own computer. But you can go further, where you dispatch a very large number of computers on your task. We discuss this last option next.

18.3 Parallel on a cluster

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 are the tasks you want the computers to do for you.

The 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 do not need all the individual computers working on your project communicating with each other (which is the hard part of distributed computing, in general).

18.3.1 Command-line interfaces and CMD mode in R

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 such as find files with a specific name, or copy entire directories, or, importantly, start different programs. One place you may have used a command line interface 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" (which are specific versions of this interface). You can get access to a shell from within RStudio by clicking on the "Terminal" tab. Try it, if you have never done anything like this before, and

type (the ">" is not part of what you type):

```
> ls
```

You should see a list of file names. 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". This will list the directory holding all the files you just saw via the `ls` command.

For a shell, commands do *not* have 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 or some generative AI 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 non-interactive: you say "go do this thing," and R starts up, goes and does it, and then quits. Any output R generates on the way will be saved in a file, and any files your script saves along the way will also be at your disposal once R has completed.

To see command-line R 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" )
```

After you save, open the terminal and type `ls` to list the files, as above. Do you see your `dumb_job.R` file? If not, your terminal session is in the wrong directory. Look at the `pwd` command to find out where R thinks it is, and then save your dumb job file to wherever that is.

Alternatively, you can 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 ..` (change 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, and save all console output in the file `R_output.txt` (the file will be saved in the current directory where you run the program). The `--no-save` means do not save the R workspace when finished.

Your 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`.

While the job is running, if you open up your Finder or Microsoft equivalent, you can actually see the `R_output.txt` file appear half-way through. If you open it, you will see the usual header of R loading up, the "Making numbers" comment, and so forth. R is saving all the output 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 launch a bunch of R programs that are each running a script like this on different computers in the cluster. They will all save their results to files (they will have files of different names so you do not overwrite your work) 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?

18.3.2 Task dispatchers and command line scripts

In the above, when you run a command on the command-line, it pauses until the command is complete.

When you properly run a big job (program) on a cluster, it does not 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 it in the background"). This is good because you get your command-line prompt back, and can use it to tell your computer to do other things, all while the program is running.

There are various methods for starting a job, but they usually boil down to a request from you to some sort of managerial process (a dispatcher) that takes requests and assigns some computer, somewhere, to do them. For a sense of this process, imagine a dispatcher at a taxi company. You call up, ask for a ride, and the dispatcher sends you a taxi to do it. The dispatcher is just fielding requests, not doing them.

One dispatcher on some clusters is called `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).

For slurm, you first set up a script that describes the job to be run. It is like a work request, and then you can request a worker to do the request via a slurm command called `sbatch`. These scripts are plain text files, such as this example (`sbatch_runScript.txt`):

```
#!/bin/bash

# Number of cores requested
#SBATCH -n 32

# Ensure that all cores are on one machine
#SBATCH -N 1

# Upper bound on runtime in minutes
#SBATCH -t 480

# Partition to submit to
#SBATCH -p stats

# Memory per cpu in MB
#SBATCH --mem-per-cpu=1000

# Append to output file, don't truncate
#SBATCH --open-mode=append

# Standard out goes to this file
#SBATCH -o /output/directory/out/%j.out

# Standard err goes to this file
#SBATCH -e /output/directory/out/%j.err

# Email notification options : BEGIN, END, FAIL, ALL
#SBATCH --mail-type=ALL
#SBATCH --mail-user=email@gmail.com

# Load modules for the computing environment
source new-modules.sh
module load gcc/7.1.0-fasrc01
module load R

# Set user R library path
export R_LIBS_USER=$HOME/apps/R:$R_LIBS_USER

# Run R script; write output to indexed log file
```

```
R CMD BATCH simulation_script.R \
                 logs/R_output_${INDEX_VAR}.txt \
                 --no-save --no-restore
```

This file starts by setting a bunch of variables that describe how `sbatch` should handle the request. It then has a series of commands that gets 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 a working example file for the system you are on, 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 the background.

Interestingly, your R script `simulation_script.R` could, at this point, do the "one computer" parallel type code listed above and use `future_pmap()` or similar. Note the sbatch script above asks for 32 cores; your single job could then have 32 cores all working away on their individual pieces of the simulation, as before. You would have about a 32-fold speedup, in this case.

Using this `sbatch` command to request a computer to do a task 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.

18.3.3 Running lots of jobs on a cluster

We have seen how to fire off a job (possibly a big job) that could 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 could not even allocate it until the needed number of workers are 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 program in the shell that makes a whole series of sbatch requests, with each request responsible for one part of the overall simulation. We write this program in the shell, just like we write R scripts in R.

For example, the following `run_full_simulation.sh` is a script that fires off a bunch of jobs for a simulation:

```
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
```

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 loop: it will run the indented code once for each number from 1 to 500. In the loop we export the `INDEX_VAR` variable, so each task can know what part of the simulation it is responsible for. The script also specifies where to put log files and names each job with the index so you can know which job generated which file.

One question is then how to have the different processes read `INDEX_VAR` so they know what part of the simulation they should be working on. E.g., each worker needs to have its own seed so it does not 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. Then, inside your R script, the index is read like so:

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

The script can then use `index` to make unique filenames and seeds. For example:

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

and

```
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).

So far we have not fully specified how to have each worker know what to work on. Consider the case of a multifactor experiment, where we have a large set of simulation scenarios we want to run.

There are two approaches one might use here. One simple approach is the following: generate all the factors with `expand_grid()` as usual, and then 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.) Now if you want 1000 total replicates for each scenario, you would have 100 workers, each doing 10 replicates of each scenario.

For saving files and then loading and combining them for analysis, see Section 16.4.

18.3.4 Moving files to and from a cluster

One annoying issue with working on a cluster is you will need to get all your code *to* the cluster and get all of your results *back*. One approach for getting the code to the cluster is to use GitHub. Make a project, and then check out the project on the cluster. This has the advantage of making it easier to develop your code on your local computer, where you can easily run RStudio or another IDE. GitHub also has the advantage of allowing you to record which version of the code you ran, so you can know exactly which code produced which results.

For getting your results off the cluster it is *not* recommended to use GitHub as your full results may take many gigabytes of space. They are also not really in the spirit of a version control system. Instead, use a scp (secure copy) client such as FileZilla to transfer files back to your personal computer.

Overall, keeping your code in Git and downloading your simulation results outside of Git is a good rule of thumb.

18.3.5 Checking on a job

Once your jobs are dispatched, the cluster will keep at it until they finish (or crash, or are terminated for taking up too much memory or time). As your jobs chug away, there will be different ways to check on them. 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. The `sacct` command lists if your file is running, pending, timed out, etc. If your job is pending, your job request is in a queue waiting to be assigned because the dispatcher has not found a free worker of sufficient size to do it yet.

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
```

Another way to check on jobs is to look at the log files to see if the job is logging information as it is working:

```
tail my_log_file.log
```

Finally, in the original sbatch script, you can set up email notifications to be sent to you when the job starts, ends, and/or fails. The email notifications you can choose are `BEGIN`, `END`, `FAIL`, and `ALL`; `ALL` is generally good. What is a few more emails?

18.4 Concluding Thoughts

Simulation lends itself naturally to parallel processing, because a simulation inherently is doing the same thing over and over. You can do parallel processing on your laptop, which can give very easy speed ups with minimal effort. If you want even more speed up, you need to turn to cloud computing or cluster computing, where you launch a simulation across many different computers. There are a variety of ways to do this, but the core workflow is going to be the same: have the different processes save their piece of the simulation results, and then assemble all of the results at the end.

When getting used to a new system for doing parallel processing, we recommend finding existing code and modifying it to suit your needs. With modular programming, this should be straightforward.

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.

Part V

Complex Data Structures

19

Using simulation as a power calculator

We can use simulation as a power calculator. In particular, to estimate power, generate data according to a best guess as to what we might find in a planned evaluation, and then analyze the data to see if the effect built into the DGP is detectable. The average chance of detection across multiple trials of simulated data is power.

Now, if we are generally right about our DGP, and the associated parameters we plugged into it, being an accurate model of the world, then our power will be right on. Estimating the chance of detection given a working model of the world is all a power analysis is, using simulation or otherwise. A power analysis simulation should, in other words, be a calibrated simulation.

Simulation has benefits over using power calculators for two reasons. First, it is easy to take into account special circumstances we might want to account for such as missing values or non-normal distributions. Second, it is easy to assess non-standard approaches to evaluation that we might not be able to find in an off-the-shelf power calculator. In particular, we can compare different proposals for how we would analyze our data, to see which ones are most powerful (and also verify that they are not biased or otherwise broken).

We illustrate these ideas with a case study. In this example, we are planning a school-level intervention to reduce rates of discipline via an intervention on both teachers and students that targets socio-emotional skills.

We are planning a planned randomized controlled trial (RCT), where we will treat entire schools (so this is a cluster-randomized study). Power is a concern because treating each school is very expensive (we have to run a large training program and also provide coaching of the staff), so treating each unit is a major decision. Ideally, we want something like 4, 5, or maybe 6 treated schools. Our diving question is: Can we get away with this? And does it help at all if our control group is larger, even if the treatment group is tiny?

Our outcome is the discipline rate at the school level for one and two years after treatment. We know we will have three years of baseline rates of discipline that we can use as a school-level covariate. After some initial brainstorming,

we generated a list of different possible regression models we might use, such as using prior years discipline rates as a control in a simple linear regression, or using a two-way fixed effects model. Our second question is then: Are these models valid in this context, and which one is best?

19.1 Getting design parameters from pilot data

Power analyses, via simulation or otherwise, depend on our assessment of what the data will look like when we finally do our analysis. In our case, we have pilot data from school administrative records (in particular discipline rates for each school and year for a series of five years), and we will 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 those schools most likely be in our study. Our assumption is that the data we are using for our simulation look like the data we would see (for the control group) when we actually run our experiment.

Here are our data, with each row a potential school in the district:

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

```
# A tibble: 27 x 6
  Code  `2015` `2016` `2017` `2018` `2019`
  <chr>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 S1     0.057  0.06   0.054  0.03   0.007
2 S2     0.027  0.059  0.077  0.063  0.036
3 S3     0.05   0.056  0.06   0.034  0.007
4 S4     0.02   0.041  0.08   0.026  0.013
# i 23 more rows
```

19.2 The data generating process

Our first step is to write a data generator that, given a desired number of control and treatment schools, and a hypothesized treatment effect, makes a synthetic dataset by generating time series of discipline rates, and then imposes a "treatment effect" of scaling the discipline rate by the treatment coefficient for the last two years.

We first summarize our five years of pilot data to estimate a mean and variance-covariance matrix on the log-transformed rates (we used the log transform to put things on a multiplicative scale, and to make the distribution of rates

more normal, given a heavy skew in the original):

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

```
     2015      2016      2017      2018      2019
-3.074399 -2.870688 -2.928910 -3.334486 -4.014886
```

```
lpd_cov = cov( datWl )
round( lpd_cov, digits=2 )
```

```
     2015 2016 2017 2018 2019
2015 0.42 0.30 0.23 0.37 0.18
2016 0.30 0.37 0.20 0.25 0.16
2017 0.23 0.20 0.31 0.29 0.25
2018 0.37 0.25 0.29 0.57 0.29
2019 0.18 0.16 0.25 0.29 0.54
```

To generate fake schools, we then sample from a multivariate normal distribution with the estimated mean and covariance structure. Here is our DGP function, `make_dat_param()`, that does this:

```
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 = dplyr::as_tibble( lpdisc ) %>%
        dplyr::mutate( ID = 1:(dplyr::n()),
                Z = 0 + ( sample( n ) <= n_t ) )

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

    lpdisc %>%
      dplyr::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 power, realism, in terms of the aspects impacting uncertainty, is key.

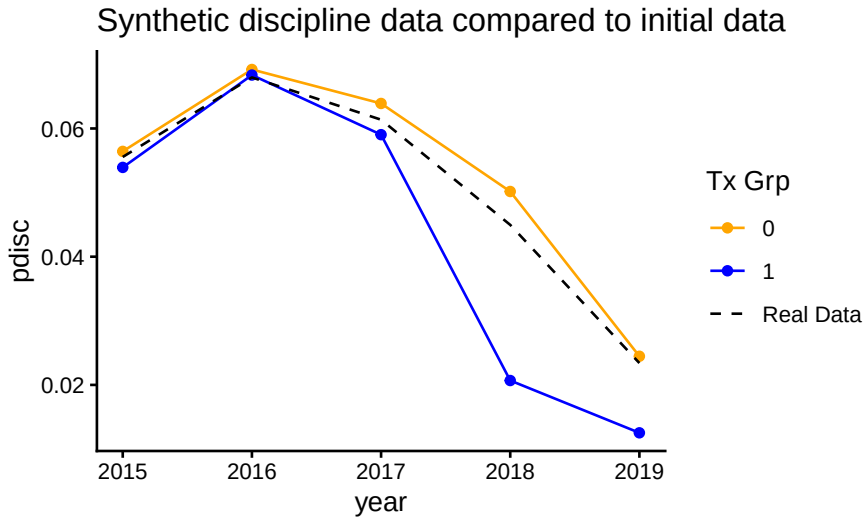
For our impact model, the treatment kicks in for the final two years, multiplying discipline rate by `tx` (so `tx = 1` means no treatment effect).

We test our function, hypothesizing the treatment reduces discipline by 50% in the two follow-up years:

```
set.seed( 594585 )
a = make_dat_param( 100, 100, 0.5 )
print( a, n = 4 )
```

```
# A tibble: 200 x 7
     ID     Z pdisc_2015 pdisc_2016 pdisc_2017 pdisc_2018
  <int> <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
1     1     0     0.0227     0.0610     0.0440     0.0489
2     2     0     0.0586     0.0626     0.0299     0.0293
3     3     0     0.0803     0.0752     0.0903     0.0567
4     4     1     0.0185     0.0279     0.0599     0.0154
# i 196 more rows
# i 1 more variable: pdisc_2019 <dbl>
```

We can group each treatment arm and look at discipline over the years for our synthetic dataset:



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

We next implement all of our candidate treatment effect estimators. This should all feel very familiar: we are just doing our simulation framework, as usual.

```
eval_dat = function( sdat, include_FE = TRUE ) {

    # 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 = dplyr::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_2015 ),
                    data=sdat )

    # Time and unit fixed effects
    if ( include_FE ) {
      sdatL = tidyr::pivot_longer( sdat, cols = pdisc_2015:pdisc_2019,
                                   names_to = "year",
                                   values_to = "pdisc" ) %>%
        dplyr::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 )
    } else {
      M_2wfe = NULL
    }

    # 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 <- purrr::map_df( models, broom::tidy, .id="model" ) %>%
      dplyr::filter( term=="Z" ) %>%
      dplyr::select( -term ) %>%
      dplyr::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. Here is the result of our evaluation function on a single dataset:

```
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             -0.652     0.307     -2.12   0.0429
2 log            -0.329     0.401     -0.821  0.472
3 log_twopost    -0.576     0.383     -1.50   0.230
4 ratio          -0.145     0.140     -1.03   0.341
5 ratio_twopost  -0.185     0.146     -1.27   0.252
6 raw            -0.718     0.634     -1.13   0.301
7 simple         -0.0118    0.0128    -0.927  0.422
```

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 = simhelpers::bundle_sim( make_dat_param, eval_dat )
```

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$reps = 1000
res$seed = 1010203 + 1:nrow(res)
```

We then use `evaluate_by_row()` to run our simulation across all of our scenarios, and save the results for later analysis:

```
plan(multisession, workers = parallel::detectCores() - 1 )
fopts = furrr_options(seed = NULL,
                      globals = c("lpd_mns", "lpd_cov") )

res_u <- simhelpers::evaluate_by_row( params = res,
                                      sim_run,
                                      .options = fopts,
                                      .progress = TRUE )

plan( sequential )

saveRDS( res_u, file="data/discipline_simulation.rds" )
```

Note that we pass our estimated mean and covariance structure from our pilot data to the parallel environment. For parallel processing, you sometimes need to explicitly hand over parts of the workspace so each worker can have access to them.

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   pow
   <dbl> <dbl> <dbl> <chr>           <dbl>  <dbl>    <dbl> <dbl>
 1     4     4   0.5  FE            -0.684  0.318    0.278  0.739
 2     4     4   0.5  log           -0.664  0.467    0.438  0.356
 3     4     4   0.5  log_twopost   -0.659  0.433    0.390  0.38
 4     4     4   0.5  ratio         -0.363  0.214    0.198  0.46
 5     4     4   0.5  ratio_twopost -0.285  0.152    0.138  0.552
```

```
 6     4     4  0.5  raw           -0.660  0.527    0.505  0.295
 7     4     4  0.5  simple        -0.0189 0.0207   0.0155 0.268
 8     4     4  0.75 FE            -0.294  0.319    0.276  0.299
 9     4     4  0.75 log           -0.276  0.481    0.432  0.15
10     4     4  0.75 log_twopost   -0.279  0.425    0.377  0.153
# i 368 more rows
```

We now have rejection rates across a range of scenarios. Before we look at power, we need to check on whether our different models are valid. Once we determine which methods are valid, we will then check to see which are most powerful, and also assess when we would have sufficient power to detect plausible effects.

19.4 Evaluating power

Before we start examining power, note that our different models are estimating different quantities. Here we look at the average estimated treatment effect across our simulation scenarios, for each method:

```
sres %>%
  group_by( model, tx ) %>%
  summarise( E_est = mean( E_est ) ) %>%
  pivot_wider( names_from="tx", values_from="E_est" ) %>%
  arrange( `0.5` ) %>%
  knitr::kable( digits = 2 )
```

model	0.5	0.75	1
FE	-0.69	-0.29	0.00
log_twopost	-0.69	-0.29	0.00
log	-0.69	-0.29	0.00
raw	-0.69	-0.28	-0.01
ratio	-0.37	-0.19	0.00
ratio_twopost	-0.29	-0.15	0.00
simple	-0.02	-0.01	0.00

For no treatment effect, we are generally seeing no estimated effect—this is good, suggesting that they are unbiased under the null of no effect whatsoever. Note, however, that when there is a treatment effect, the different methods are giving different answers. Furthermore, in the 0.5 case, none of these answers are close to 0.5!

What is happening, is our models are estimating different things, none of

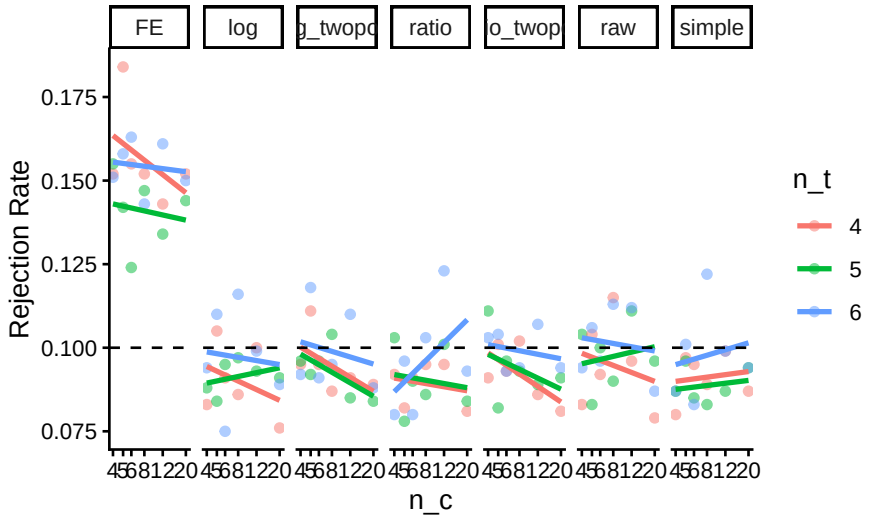
which are the treatment effect as we parameterized it. In particular, "FE," "log," "raw," and "log_twopost" are all estimating the impact on the log scale. Note that $log(0.5) \approx -0.69$ and $log(0.75) \approx -0.29$, which corresponds to what we see on the above. Our "simple" estimator is estimating the impact on the absolute scale; it turns out that 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.

Estimating different things, from a power point of view, is ok. Power is the chance of detecting an effect, and is dependent on the p-values, not scale of the estimate. We next show how to assess methods for validity, and then show how to evaluate power.

19.4.1 Checking the validity of our models

Validity is especially 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` (no treatment effect), and look at the rejection rates by plotting rejection rates for all models across all our simulation factors. We use a linear smoother to reduce Monte Carlo noise, and mark the nominal 10% rejection rate with a dashed line.

```
sres %>% filter( tx == 1 ) %>%
ggplot( aes( n_c, pow, col=as.factor(n_t) ) ) +
  facet_wrap( ~ model, nrow=1 ) +
  geom_point( alpha=0.5 ) +
  geom_smooth( method="lm", se=FALSE ) +
  geom_hline( yintercept = 0.10, lty = 2 ) +
  scale_x_log10(breaks=unique(sres$n_c) ) +
  labs( col="n_t", y="Rejection Rate" )
```

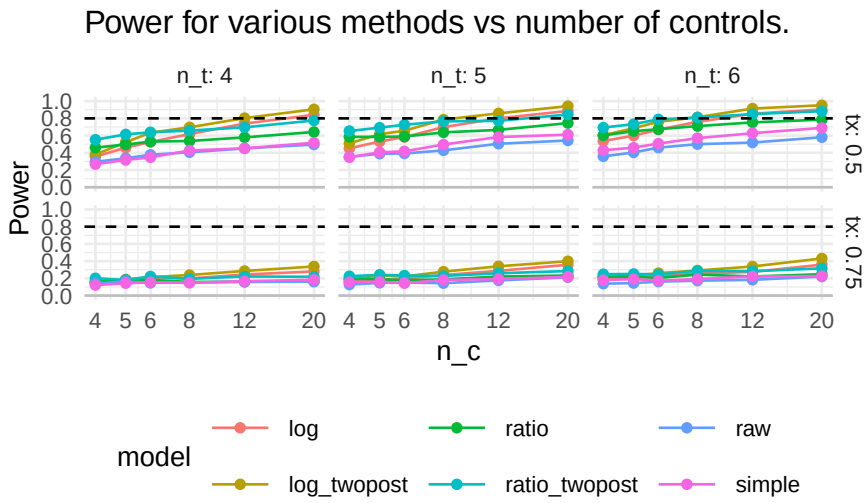


We see the fixed effect models have notably elevated rates of rejection under all scenarios considered. The other models look fine. This means our fixed effect approach is an invalid choice for analysis; even if it is more powerful, it is not giving us a fair evaluation. We therefore drop it in our subsequent power analysis.

19.4.2 Assessing power

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

```
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( 0.80 ), lty=2 ) +
  scale_x_log10( breaks=unique( sres$n_c ))  +
  scale_y_continuous( breaks=c(0, 0.2, 0.4, 0.6, 0.8, 1) ) +
  expand_limits( y = c(0,1) ) +
  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. The "raw" estimator gives a baseline of no covariate adjustment; everything is substantially more precise than it. The covariates matter a lot. Not working with the log scale is costly: the simple estimator, which adjusts for covariates but does not use a log transform, is also low power. Averaging two years of outcomes post-treatment also seems beneficial: the "twopost" methods have a distinct power bump over their counterparts. For a single year of outcome data, the log model seems our best bet.

19.4.3 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%. If working with standard errors, a MDE is usually the SE times a constant, such as 2.8 for 80% power, 0.05 alpha, and a normal approximation.

With simulation, we can calculate Minimal Detectable Effects by generating a series of scenarios with gradually increasing treatment effects, and then finding out how large they have to get to obtain 80% power.

Using the same code as above, we have:

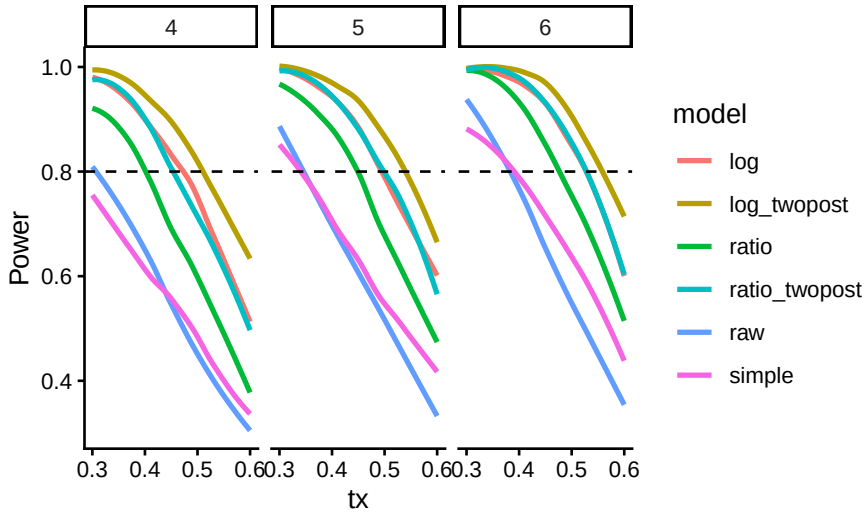
```
res = expand_grid( tx = seq( 0.3, 0.6, by = 0.01 ),
                   n_c = c( 12 ),
                   n_t = c( 4, 5, 6 ) )
```

```
res$reps = 200
res$seed = 101440203 + 1:nrow(res)
```

We will end up with a lot of simulation scenarios to run (in this case we have 93). For each scenario, we do not need to run many replicates since we will be smoothing across our continuous treatment effect values. We run these using the same template as above.

Here we plot our results:

```
res <- readRDS( file="data/discipline_simulation_MDE.rds" )
sresMDE <- res %>%
  group_by( model, tx, n_c, n_t ) %>%
  summarise( pow = mean( p.value <= 0.10 ) )

ggplot( sresMDE, aes( tx, pow, col=model) ) +
  facet_wrap( ~ n_t ) +
  geom_smooth( se=FALSE ) +
  geom_hline( yintercept = 0.8, lty = 2 ) +
  labs( y = "Power" )
```



In looking at the graph, the point where each line crosses the 0.80 dashed line is our MDE, in terms of our treatment effect as a reduction in disciplinary rates. Remember that smaller values are *larger* effects (since it is the percent of prior discipline). We can convert to percent reduction by subtracting these values from 1, and then estimate the MDEs for each method by solving where each line crosses 0.8:

```
mde_est <- sresMDE %>%
  mutate( tx = 1 - tx ) %>%
  group_by(model, n_c, n_t) %>%
  group_modify( \(.x, .y) {
    fit <- loess(pow ~ tx, data = .x)
    .x %>%
      mutate(pred = predict(fit)) %>%
      slice_min( abs(pred-0.8), n = 1) %>%
      select(mde = tx, pred)
  }) %>%
  ungroup() %>%
  dplyr::select( -pred ) %>%
  pivot_wider( names_from = n_t, values_from = mde ) %>%
  arrange( `6` )

mde_est
```

```
# A tibble: 6 x 5
  model           n_c   `4`   `5`   `6`
  <chr>         <dbl> <dbl> <dbl> <dbl>
1 log_twopost      12  0.49  0.46  0.44
2 log              12  0.53  0.51  0.47
3 ratio_twopost    12  0.55  0.5   0.47
4 ratio            12  0.6   0.55  0.52
5 simple           12  0.7   0.66  0.61
6 raw              12  0.69  0.65  0.62
```

With 6 treatment units and 12 control units, we can detect a 44% reduction in discipline rates with 80% power, using the twopost estimator working with log-transformed discipline.

19.5 Using and exploring packages for power calculations

Many randomized trials have distinct structure such as clustering. For example, researchers frequently want to calculate power for multisite randomized trials, where each of a series of sites has students randomized to treatment, or not. Our earlier cluster RCT case study (see, e.g., Section 6.6) is another example of this, where entire sites are randomized, but we have outcomes at the student level.

We can use the same simulation framework as above to calculate power for these types of models. We write a data generation function that generates our data given our target structure, and then repeatedly generate and analyze

data to assess power, just as we have done.

As we saw earlier, however, it can be sometimes tricky to write code that properly has covariates that relate to the different levels of our model, or that divides variance appropriately across levels. For example, in a multisite experiment, we might want a covariate that has a different mean value within each cluster, but also has variation within cluster.

Instead of immediately writing our own data generation function when faced with such a project, it might be worth looking to the literature to see what tools are available. If we looked in this case, for example, we might come across Enders et al. (2023), which showcases a package, `mlmpower`, designed to generate data according to a flexible range of multilevel models.

```
library( mlmpower )
model <- outcome('Y') +
  within_predictor('X') +
  between_predictor('W') +
  effect_size(icc = 0.1,
              within = 0.2 )

set.seed(4232345)

dat <- generate( model,
                 n_within = 5,
                 n_between = 12 ) %>%
  as_tibble()
print( dat, n=4 )
```

```
# A tibble: 60 x 4
  `_id`     Y      X        W
  <int> <dbl>  <dbl>    <dbl>
1     1 20.6   0.705 -0.00235
2     1  5.32 -0.146 -0.00235
3     1 15.6   0.265 -0.00235
4     1  4.69  1.32  -0.00235
# i 56 more rows
```

Due to our coefficient specification, our covariate X is correlated with Y:

```
cor( dat$X, dat$Y )
```

```
[1] 0.5310738
```

The package even provides methods for calculating power, assuming multilevel modeling (via the `lme4` package) is used for the estimator. The power analysis method of this package in fact runs a multifactor simulation:

```
                    value     mc_moe
```

```
fixed.(Intercept)  1.00 0.00000000
fixed.cgm(X)       0.95 0.04293162
fixed.cgm(W)       0.05 0.04293162
```

We have estimated 95% power for detecting the effect of the within-level predictor, `X`, with 10 students per site and 5 sites, given the parameters we set. We refer the reader to the package documentation for how to use this specific package.

The broader point is it is sometimes worth digging into provided code to get material for generating data or even running simulations. That said, each package is designed for a specific purpose, and has its own language. Here, for example, the effect size, defined as `within`, is rooted in how the package centers and incorporates covariates into the model. Tying its parameterization to classic regression coefficients may be non-obvious. It is thus sometimes easy to use a package to get data that looks like data, but does not actually have the structure you intend. As always, do diagnostics and verification to ensure the tools are working as you expect.

Here, for example, we might generate a very large dataset to get estimated coefficients as a sanity check:

```
dat <- generate( model,
                 n_within = 100,
                 n_between = 100 ) %>%
  as_tibble()
M = lme4::lmer( Y ~ X + W + (1|`_id`), data=dat )
arm::display(M)
```

```
lme4::lmer(formula = Y ~ X + W + (1 | `_id`), data = dat)
            coef.est coef.se
(Intercept) 10.05     0.15
X            2.44     0.04
W            0.03     0.14

Error terms:
 Groups   Name        Std.Dev.
 _id      (Intercept) 1.40
 Residual             4.22
---
number of obs: 10000, groups: _id, 100
AIC = 57462.4, DIC = 57435.4
deviance = 57443.9
```

The coefficient for X is 2.44, which is not the 0.2 we specified. Clearly some scaling is happening.

We can check our ICC:

```
sigma2_2 = lme4::VarCorr(M)$`_id`[1]
sigma2_1 = attr(lme4::VarCorr(M), "sc")^2
sigma2_2 / (sigma2_2 + sigma2_1)
```

```
[1] 0.09864158
```

At least this is close to the listed 0.10. More testing is needed.

19.6 Concluding Thoughts

Power analysis simulations are simply simulations with the chance of rejecting the null as the primary performance measure of interest. The key concern with a power analysis is to calibrate the simulation to ones best guess as to what the real data will look like. It is wise to run a series of simulations under related scenarios to try and triangulate what possible worlds one might face, and then see how power changes in the face of that uncertainty.

Power analysis can also be used for planning: by comparing different designs, we can see which ones are valid, and which are most powerful.

Power analysis simulations have some advantages over calculators. First, it is easier to test how things might change given possible features of the data. For example, we saw how non-normality impacted our estimators in the primary simulation on discipline, above. Second, once the simulation is written, the code for analyzing the future data is also written! By the end of a simulation power analysis, the researcher knows their methods deeply, has the framework to test their methods in the face of misspecification, and knows the chance of errors in their code is minimal (especially if they did a valididty check as well as power analysis).

20

Simulation under the Potential Outcomes Framework

In this chapter we give some general approaches for running simulations in the context of causal inference. The broader point of this chapter is to underscore that one can be very creative in how one designs a simulation, insofar as there is a well-defined way to make data with a well-defined "answer" that we are trying to recover by analyzing the synthetic data.

When evaluating causal inference methods, we would probably turn to the potential outcomes framework. The potential outcomes framework is a framework typically used in the causal inference literature to make explicit statements regarding the mechanics of causality and the associated estimands one might target. While we recommend reading, for a more thorough overview, either Rosenbaum (2017) or Gerber and Green (2017), we briefly outline this framework here to set out our notation.

Take a sample of n experimental units, indexed by i. For each unit, we can treat it or not, with $Z_i = 1$ if we do and $Z_i = 0$ if we do not. Now we imagine each unit has two *potential outcomes* we might see depending on whether we treated it ($Y_i(1)$) or not ($Y_i(0)$). We observe only one of these outcomes, depending on what we did:

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

The "Fundamental Problem of Causal Inference" is that we can never see both potential outcomes for a given unit. But with simulation, since we will be making our own data, we can—and that allows us to evaluate causal inference methods in many natural-seeming scenarios.

Each unit's treatment effect is then $\tau_i = Y_i(1) - Y_i(0)$; it is how much our outcome changes if we treat vs. not treat. Frustratingly, because we can only see one of a unit's two potential outcomes, we can never directly estimate an 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 most common estimand of 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 proportional increase, by contrast, would be

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

Importantly, under a potential outcomes framework, our sample is defined by the potential outcomes, and randomness comes from the random assignment of treatment, not the sample itself. We call the process of how units are assigned to treatment the *assignment mechanism*.

20.1 Finite vs. Superpopulation inference

In the above we consider the sample as fixed. That said, if we thought of the sample as being drawn from some larger population (often called a "superpopulation"), 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, and different units have different τ_i, then our estimand could be 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 across our samples.

Now if we were generating data with a constant treatment impact, then all the τ_i are the same and $ATE_{\mathcal{S}} = ATE$ always; this is typical for many simulations in the literature. That being said, treatment variation is what causes a lot of methods to fail, and so having simulations with treatment variation is usually important.

The catch is when we calculate our performance metrics, we now have two possible targets to pick from: do we compare our estimate to the sample's true treatment effect, or the population's? Furthermore, if we are targeting the population 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 Generating data under the potential outcome framework

When writing a simulation using the potential outcomes framework, we recommend first generating a complete set of potential outcomes, then a random assignment based on some assignment mechanism, and finally the observed outcomes as a function of assignment and original potential outcomes.

For the first step, we 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 says 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.

The result of the above is a *schedule of potential outcomes* for our n units. The schedule of potential outcomes encodes, for each unit, what would happen to it given the possible treatment assignements it might receive.

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 implementing the above model (we leave treatment assignment to later):

```
gen_data <- function( n = 100,
                      R2 = 0.5,
                      beta_0 = 0, beta_1 = 1,
                      tau_0 = 1, tau_1 = 1,
                      alpha = 1, df = 3 ) {
  stopifnot( R2 >= 0 && R2 < 1 )
  X_i = rexp( n, rate = 1 ) - 1
  beta_1 = sqrt( 1 - R2 )
  sigma_e = sqrt( R2 )
  Y0_i = beta_0 + beta_1 * X_i + rnorm( n, sd=sigma_e )
```

```
  tau_i = tau_0 + tau_1 * X_i + alpha * rt( n, df = df )
  Y1_i = Y0_i + tau_i

  tibble( X = X_i, Y0 = Y0_i, Y1 = Y1_i )
}
```

We have reparameterized our model so we can index our DGP by an R2 value rather than a coefficient 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.

Here is some sample data:

```
set.seed( 40454 )
d1 <- gen_data( 50 )
head( d1 )
```

```
# A tibble: 6 x 3
       X      Y0     Y1
   <dbl>   <dbl>  <dbl>
1 -0.769 -0.572   2.50
2  0.136 -0.549   1.95
3 -0.164 -1.12    2.21
4  0.417  0.572   0.623
5 -0.623  0.493  -2.77
6 -0.128  0.0174  0.277
```

We store latent 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.

Note how our estimand can change with each sample generated:

```
mean( d1$Y1 - d1$Y0 )
```

```
[1] 0.6374925
```

```
d2 <- gen_data( 50 )
mean( d2$Y1 - d2$Y0 )
```

```
[1] 0.5479788
```

In reviewing our model and 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.

Once we have our *schedule of potential outcomes*, we generate the *observed outcomes* by assigning our (synthetic, randomly generated) n units to treat-

ment or control. For example, say we wanted to simulate an observational context where chance of treatment was a function of our covariate. We could model each unit as flipping a weighted coin with some probability that was a function of X_i as so:

$$
\begin{aligned}
p_i &= logit^{-1}(\xi_0 + \xi_1 X_i) \\
Z_i &= Bern(p_i) \\
Y_i &= Z_i Y_i(1) + (1 - Z_i) Y_i(0)
\end{aligned}
$$

Here is code for assigning our data to treatment and control:

```
assign_data <- function( dat,
                         xi_0 = -1, xi_1 = 1 ) {
  n = nrow(dat)
  dat = mutate( dat,
                p = arm::invlogit( xi_0 + xi_1 * X ),
                Z = rbinom( n, 1, prob=p ),
                Yobs = ifelse( Z == 1, Y1, Y0 ) )
  dat
}
```

We can then add our assignment variable to our given data as so:

```
assign_data( d2 )
```

```
# A tibble: 50 x 6
          X     Y0      Y1     p     Z    Yobs
      <dbl>  <dbl>   <dbl> <dbl> <int>   <dbl>
 1  0.670    0.667   2.58  0.418     1   2.58
 2  0.371    0.314   4.57  0.348     1   4.57
 3  1.94     1.29    3.03  0.719     0   1.29
 4 -0.244    0.119 -10.0   0.224     1 -10.0
 5  0.00850  1.44    2.88  0.271     0   1.44
 6  1.41     1.14    5.02  0.600     1   5.02
 7 -0.864    0.461   0.802 0.134     1   0.802
 8 -0.00533 -0.914  -1.17  0.268     0  -0.914
 9 -0.907   -0.202   0.555 0.129     1   0.555
10 -0.363   -0.141   1.16  0.204     1   1.16
# 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) are randomly assigning these whole units to various conditions and observing the consequences.

We might 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

For a finite sample evaluation, we can 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:

```
tauS = mean( dat$Y1 - dat$Y0 )
tauS
```

```
[1] 1.154018
```

Our finite-population simulation would be:

```
rps <- simhelpers::repeat_and_stack( 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"]] )
  })
rps %>%
  summarise( EATE_hat = mean( ATE_hat ),
             bias = EATE_hat - tauS,
                   SE = sd( ATE_hat ),
                   ESE_hat = sqrt( mean( SE_hat^2 ) ) )
```

```
# A tibble: 1 x 4
  EATE_hat    bias    SE ESE_hat
     <dbl>   <dbl> <dbl>   <dbl>
```

```
1     1.16 0.00482 0.248   0.309
```

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. The finite sample simulation opens up some room for critique: what if our single dataset is non-standard? How can we claim our findings are general? For example, perhaps the difference in the true SE and average estimated one is a quirk of our particular dataset, and not a general phenomenon.

Our super-population simulation would be, by contrast:

```
rps_sup <- simhelpers::repeat_and_stack( 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"]] )
  })

rps_sup %>% summarise( EATE_hat = mean( ATE_hat ),
                       bias = EATE_hat - 1,
                   SE = sd( ATE_hat ),
                   ESE_hat = sqrt( mean( SE_hat^2 ) ) )
```

```
# A tibble: 1 x 4
  EATE_hat        bias    SE ESE_hat
     <dbl>       <dbl> <dbl>   <dbl>
1    1.000 -0.00000591 0.381   0.389
```

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 standard 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 finite 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 phenomenon 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. See Miratrix et al. (2021) for an example of this kind of simulation framework.

Borrowing from the simulation appendix of Miratrix et al. (2021), for a nested simulation 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. We store both potential outcomes for all generated units.
2. Record the true finite-sample ATE for our generated data.
3. Then, three times, do a finite simulation as follows:

a. Randomize units to treatment and control using the assignment mechanism.
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 (finite sample) performance, but the average performance across the R datasets in a given scenario will give a reasonable estimate of the *expected* performance 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 <- simhelpers::repeat_and_stack( R0, {
         dat = mutate( dat,
```

```
                    Z = 0 + (sample( n ) <= n/2),
                    Yobs = ifelse( Z == 1, Y1, Y0 ) )
        mod = lm( Yobs ~ Z, data=dat )
        tibble( ATE_hat = coef(mod)[["Z"]],
                SE_hat = arm::se.coef(mod)[["Z"]] )
    }) %>%
    bind_rows()
  rps$ATE = mean( dat$Y1 - dat$Y0 )
  rps
}
```

This driver also stores the finite sample ATE for future reference:

```
one_finite_run()
```

```
# A tibble: 3 x 3
  ATE_hat SE_hat   ATE
    <dbl>  <dbl> <dbl>
1   0.348  0.421 0.768
2   1.32   0.472 0.768
3   1.17   0.549 0.768
```

We then run a bunch of finite runs.

```
runs <- simhelpers::repeat_and_stack( 500, one_finite_run(),
                                      id = "runID" )
head( runs )
```

```
  runID    ATE_hat    SE_hat       ATE
1     1 0.61007830 0.4908961 0.7583748
2     1 0.04792879 0.3968365 0.7583748
3     1 0.76594448 0.4617217 0.7583748
4     2 0.62535954 0.3123307 0.8821758
5     2 0.59101011 0.3247807 0.8821758
6     2 1.04007072 0.3666993 0.8821758
```

We set an 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 finite-level 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 replicates within each finite sample run:

```
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.

20.5 Superpopulation performance measures

For a superpopulation simulation, the easy road is to simply generate a new dataset with each simulation trial. We can also 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 for assessing calibration):

```
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
```

We see our estimated SEs may be a bit too small, on average, as compared to the true SE.

20.6 Calibrating simulations within the potential outcomes framework

The potential outcomes framework provides a natural path for generating calibrated simulations for causal inference problems (See Section 15.1.6 for more discussion of calibration.)

Under the calibration framework, we would take an existing randomized experiment or observational study and then 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 fully known.

Data in hand, we 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 here is the imputation step: how do we build the full set of covariates and potential outcomes? There are a variety of options that have different pros and cons.

Regardless of how we generate the missing potential outcomes, once we have a "fully observed" sample with the full set of treatment and control potential outcomes for all of our units, we can calculate any target estimands we like on our population, and then compare our estimators to these ground truths (even if they have no parametric analog) as desired, just as we saw above.

20.6.1 Matching-based imputation

One approach for generating pairs of potential outcomes is to generate a matched-pairs dataset by, for each unit, finding a close match in terms of

the covariates. The opposite pair of each matched unit then gives the imputed potential outcome.
By doing this (with replacement) for all units we can generate a fully imputed dataset which we can then use as our population, with all outcomes being "real," as they are taken from actual data. Such matching can mostly preserve complex relationships between covariates and outcomes in a manner that is not model dependent. For example, if an outcome is coarsely defined (e.g., on an integer scale) or has specific clumps (such as from zero-inflation or rounding), this structure will be preserved.

One concern with the matching approach is the inexactness of the matching could dilute the structure of the treatment effect as the control- and treatment-side potential outcomes may be very unrelated, creating a lot of so-called idiosyncratic treatment variation (unit-to-unit variation in the treatment effects that is not explained by the covariates). This is akin to measurement error diluting found relationships in linear models.

We could try to reduce such variation by first imputing missing outcomes using some model (e.g., a random forest) fit to the original data, and then matching units including the imputed potential outcomes as additional "covariates." 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.

20.6.2 Model-based imputation

A second approach for imputing the missing potential outcomes is to specify a treatment effect model, predict treatment effects for all units and use the predicted effects to impute the treatment potential outcome for all control units and control potential outcomes for all treated units. For example, if we fit $\widehat{\tau}(X)$ to the original data, where $\widehat{tau}(X)$ estimates the treatment effect for a unit with covariate value X, we can then impute $Y_i(1) = Y_i(0) + \widehat{\tau}(X_i)$ for all control units and $Y_i(0) = Y_i(1) - \widehat{f}(X_i)$ for all treated units. This imputation perfectly preserves the complex structure between the covariates and the $Y_i(0)$s for the original controls and covariates and $Y_i(1)$ for the original treated.

One drawback is the imputed outcomes may not be values we actually see on the counterfactual side. For example, if the original outcomes are binary, but if $\widehat{\tau}(X)$ estimates fractional effects, then the imputed outcomes will not be binary, and could even be negative or above 1.

A second (probably minor) drawback is this form of imputation does not allow for any idiosyncratic treatment variation. To add in idiosyncratic variation we would need to generate a distribution of perturbations and add these to the imputed outcomes just as we would an error term in a regression model.

One advantage of the modeling approach is it allows for varying the amount of treatment effect. We can, for example, scale $\hat{\tau}(X)$ by some factor to increase or decrease the size of the treatment effects in our imputed dataset.

20.7 Concluding Thoughts

Potential outcomes are a way of thinking about causal inference where we fully specify characteristics of the data, and then define estimands as summary measures across the full data (including aspects we would not normally see in the real world). This idea is a general one: in any simulation context, it is important to remember that the model for *generating* the data need not have any relation to how we *estimate* or *analyze* the data. By keeping these pieces separate in a simulation, it is much easier to generate data that is relevant to a specific context (e.g., by calibrating to an existing dataset) than when trying to use the same model used for estimation.

Keeping these parts separate has another important quality of helpling keep the researcher honest: it is usually the case that an estimator will work best on data generated from the estimator's own working model. By generating using a different process, then the data does not inherently align with the estimator, meaning the estimator performance is less likely to be entirely a special case.

References

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.

Antonakis, J., Bastardoz, N., and Rönkkö, M. (2021). On ignoring the random effects assumption in multilevel models: Review, critique, and recommendations. *Organizational Research Methods*, 24(2):443–483.

Benjamin, D. J., Berger, J. O., Johannesson, M., Nosek, B. A., Wagenmakers, E.-J., Berk, R., Bollen, K. A., Brembs, B., Brown, L., Camerer, C., Cesarini, D., Chambers, C. D., Clyde, M., Cook, T. D., De Boeck, P., Dienes, Z., Dreber, A., Easwaran, K., Efferson, C., Fehr, E., Fidler, F., Field, A. P., Forster, M., George, E. I., Gonzalez, R., Goodman, S., Green, E., Green, D. P., Greenwald, A. G., Hadfield, J. D., Hedges, L. V., Held, L., Hua Ho, T., Hoijtink, H., Hruschka, D. J., Imai, K., Imbens, G., Ioannidis, J. P. A., Jeon, M., Jones, J. H., Kirchler, M., Laibson, D., List, J., Little, R., Lupia, A., Machery, E., Maxwell, S. E., McCarthy, M., Moore, D. A., Morgan, S. L., Munafó, M., Nakagawa, S., Nyhan, B., Parker, T. H., Pericchi, L., Perugini, M., Rouder, J., Rousseau, J., Savalei, V., Schönbrodt, F. D., Sellke, T., Sinclair, B., Tingley, D., Van Zandt, T., Vazire, S., Watts, D. J., Winship, C., Wolpert, R. L., Xie, Y., Young, C., Zinman, J., and Johnson, V. E. (2017). Redefine statistical significance. *Nature Human Behaviour*, 2(1):6–10.

Bloom, H. S., Raudenbush, S. W., Weiss, M. J., and Porter, K. (2016). Using Multisite Experiments to Study Cross-Site Variation in Treatment Effects: A Hybrid Approach With Fixed Intercepts and a Random Treatment Coefficient. *Journal of Research on Educational Effectiveness*, 10(4):0–0.

Boos, D. D. and Osborne, J. A. (2015). Assessing variability of complex descriptive statistics in Monte Carlo studies using resampling methods. *International Statistical Review*, 83(2):228–238.

Borenstein, M., Hedges, L. V., Higgins, J. P., and Rothstein, H. R. (2021). *Introduction to Meta-Analysis*. John Wiley & Sons, Chichester, UK, 3 edition.

Boulesteix, A.-L., Hoffmann, S., Charlton, A., and Seibold, H. (2020). A replication crisis in methodological research? *Significance*, 17(5):18–21.

Boulesteix, A.-L., Lauer, S., and Eugster, M. J. A. (2013). A plea for neutral comparison studies in computational sciences. *PLOS ONE*, 8(4):e61562.

Boulesteix, A.-L., Wilson, R., and Hapfelmeier, A. (2017). Towards evidence-based computational statistics: Lessons from clinical research on the role and design of real-data benchmark studies. *BMC Medical Research Methodology*, 17(1):1–12.

Brown, M., Solomon, H., and Stephens, M. A. (1977). Estimation of parameters of zero-one processes by interval sampling. *Operations Research*, 25(3):493–505.

Brown, M. B. and Forsythe, A. B. (1974). The Small Sample Behavior of Some Statistics Which Test the Equality of Several Means. *Technometrics*, 16(1):129–132.

Cameron, A. C. and Miller, D. L. (2015). A practitioner's guide to cluster-robust inference. *Journal of Human Resources*, 50(2):317–372.

Chen, M. and Pustejovsky, J. E. (2024). Multi-level meta-analysis of single-case experimental designs using robust variance estimation. *Psychological Methods*, 29(3):537–560.

Chen, M. and Pustejovsky, J. E. (2025). Adapting methods for correcting selective reporting bias in meta-analysis of dependent effect sizes. *Psychological Methods*.

Cho, H., Liu, C., Preisser, J. S., and Wu, D. (2023). A bivariate zero-inflated negative binomial model and its applications to biomedical settings. *Statistical Methods in Medical Research*, 32(7):1300–1317.

Davison, A. C. and Hinkley, D. V. (1997). *Bootstrap Methods and Their Applications*. Cambridge University Press, Cambridge.

Dong, N. and Maynard, R. (2013). *PowerUp!* : A Tool for Calculating Minimum Detectable Effect Sizes and Minimum Required Sample Sizes for Experimental and Quasi-Experimental Design Studies. *Journal of Research on Educational Effectiveness*, 6(1):24–67.

Dorie, V., Hill, J., Shalit, U., Scott, M., and Cervone, D. (2019). Automated versus do-it-yourself methods for causal inference: Lessons learned from a data analysis competition. *Statistical Science*, 34(1):43–68.

Efron, B. (2000). The bootstrap and modern statistics. *Journal of the American Statistical Association*, 95(452):1293–1296.

Efron, B. and Tibshirani, R. J. (1994). *An introduction to the bootstrap*. Chapman and Hall/CRC.

Enders, C. K., Keller, B. T., and Woller, M. P. (2023). A simple monte carlo method for estimating power in multilevel designs. *Psychological Methods*.

Faul, F., Erdfelder, E., Buchner, A., and Lang, A.-G. (2009). Statistical power analyses using G*Power 3.1: Tests for correlation and regression analyses. *Behavior Research Methods*, 41(4):1149–1160.

Franklin, J. M., Schneeweiss, S., Polinski, J. M., and Rassen, J. A. (2014). Plasmode simulation for the evaluation of pharmacoepidemiologic methods in complex healthcare databases. *Computational Statistics & Data Analysis*, 72:219–226.

Freedman, D. A. (2008). On regression adjustments to experimental data. *Advances in Applied Mathematics*, 40(2):180–193.

Friedman, L. W. and Pressman, I. (1988). The metamodel in simulation analysis: Can it be trusted? *Journal of the Operational Research Society*, 39(10):939–948.

Fryda, T., LeDell, E., Gill, N., Aiello, S., Fu, A., Candel, A., Click, C., Kraljevic, T., Nykodym, T., Aboyoun, P., Kurka, M., Malohlava, M., Poirier, S., and Wong, W. (2014). H2o: R Interface for the 'H2O' Scalable Machine Learning Platform.

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., and Rubin, D. B. (2013). *Bayesian Data Analysis*. Chapman and Hall/CRC.

Gelman, A., Pasarica, C., and Dodhia, R. (2002). Let's practice what we preach: turning tables into graphs. *The American Statistician*, 56(2):121–130.

Gentzel, A. M., Pruthi, P., and Jensen, D. (2021). How and why to use experimental data to evaluate methods for observational causal inference. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 3660–3671. PMLR.

Gerber, A. S. and Green, D. P. (2017). Field experiments on voter mobilization: An overview of a burgeoning literature. *Handbook of economic field experiments*, 1:395–438.

Gilbert, J. and Miratrix, L. (2024). Multilevel metamodels: A novel approach to enhance efficiency and generalizability in monte carlo simulation studies. *arXiv preprint arXiv:2401.07294*.

Good, P. (2013). *Permutation tests: a practical guide to resampling methods for testing hypotheses*. Springer Science & Business Media.

Hettmansperger, T. P. and McKean, J. W. (2010). *Robust Nonparametric Statistical Methods*. Chapman & Hall / CRC Monographs on Statistics & Applied Probability. Taylor and Francis, 2nd ed edition.

Hill, J. L. (2011). Bayesian nonparametric modeling for causal inference. *Journal of Computational and Graphical Statistics*, 20(1):217–240.

Hoogland, J. J. and Boomsa, A. (1998). Robustness Studies in Covariance Structure Modeling: An Overview and a Meta-Analysis. *Sociological Methods & Research*, 26(3):329–367.

Hunter, K. B., Miratrix, L., and Porter, K. (2024). Pump: Estimating power, minimum detectable effect size, and sample size when adjusting for multiple outcomes in multi-level experiments. *Journal of Statistical Software*, 108(6):1–43.

James, G. S. (1951). The comparison of several groups of observations when the ratios of the population variances are unknown. *Biometrika*, 38(3/4):324.

Jones, O., Maillardet, R., and Robinson, A. (2012). *Introduction to Scientific Programming and Simulation Using R*. Chapman and Hall/CRC, New York.

Kang, H., Peck, L., and Keele, L. (2018). Inference for instrumental variables: a randomization inference approach. *Journal of the Royal Statistical Society Series A: Statistics in Society*, 181(4):1231–1254.

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.

Kleijnen, J. P. (1981). Regression analysis for simulation practitioners. *Journal of the Operational Research Society*, 32(1):35–43.

Lakens, D., Adolfi, F. G., Albers, C. J., Anvari, F., Apps, M. A. J., Argamon, S. E., Baguley, T., Becker, R. B., Benning, S. D., Bradford, D. E., Buchanan, E. M., Caldwell, A. R., Van Calster, B., Carlsson, R., Chen, S.-C., Chung, B., Colling, L. J., Collins, G. S., Crook, Z., Cross, E. S., Daniels, S., Danielsson, H., DeBruine, L., Dunleavy, D. J., Earp, B. D., Feist, M. I., Ferrell, J. D., Field, J. G., Fox, N. W., Friesen, A., Gomes, C., Gonzalez-Marquez, M., Grange, J. A., Grieve, A. P., Guggenberger, R., Grist, J., Van Harmelen, A.-L., Hasselman, F., Hochard, K. D., Hoffarth, M. R., Holmes, N. P., Ingre, M., Isager, P. M., Isotalus, H. K., Johansson, C., Juszczyk, K., Kenny, D. A., Khalil, A. A., Konat, B., Lao, J., Larsen, E. G., Lodder, G. M. A., Lukavský, J., Madan, C. R., Manheim, D., Martin, S. R., Martin, A. E., Mayo, D. G., McCarthy, R. J., McConway, K., McFarland, C., Nio, A. Q. X., Nilsonne, G., De Oliveira, C. L., De Xivry, J.-J. O., Parsons, S., Pfuhl, G., Quinn, K. A., Sakon, J. J., Saribay, S. A., Schneider, I. K., Selvaraju, M., Sjoerds, Z., Smith, S. G., Smits, T., Spies, J. R., Sreekumar, V., Steltenpohl, C. N., Stenhouse, N., Świątkowski, W., Vadillo, M. A., Van Assen, M. A. L. M., Williams, M. N., Williams, S. E., Williams, D. R., Yarkoni, T., Ziano, I., and Zwaan, R. A. (2018). Justify your alpha. *Nature Human Behaviour*, 2(3):168–171.

Lee, Y. R. and Pustejovsky, J. E. (2023). Comparing random effects models,

ordinary least squares, or fixed effects with cluster robust standard errors for cross-classified data. *Psychological Methods*.

Lehmann, E. L. et al. (1975). Statistical methods based on ranks. *Nonparametrics. San Francisco, CA, Holden-Day*, 2.

Lin, W. (2013). Agnostic notes on regression adjustments to experimental data: Reexamining freedman's critique. *The Annals of Applied Statistics*, pages 295–318.

Little, R. J. (2013). In praise of simplicity not mathematistry! ten simple powerful ideas for the statistical scientist. *Journal of the American Statistical Association*, 108(502):359–369.

Long, J. S. and Ervin, L. H. (2000). Using heteroscedasticity consistent standard errors in the linear regression model. *The American Statistician*, 54(3):217–224.

Maechler, M., Rousseeuw, P., Croux, C., Todorov, V., Ruckstuhl, A., Salibian-Barrera, M., Verbeke, T., Koller, M., Conceicao, E. L. T., and Anna di Palma, M. (2024). *robustbase: Basic Robust Statistics*. R package version 0.99-4-1.

Maronna, R. A., Martin, R. D., and Yohai, V. J. (2006). *Robust Statistics: Theory and Methods*. Wiley Series in Probability and Statistics. J. Wiley, Chichester (GB).

McKean, J. W. and Schrader, R. M. (1984). A comparison of methods for studentizing the sample median. *Communications in Statistics - Simulation and Computation*, 13(6):751–773.

Mehrotra, D. V. (1997). Improving the brown-forsythe solution to the generalized behrens-fisher problem. *Communications in Statistics - Simulation and Computation*, 26(3):1139–1145.

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.

Morris, T. P., White, I. R., and Crowther, M. J. (2019). Using simulation studies to evaluate statistical methods. *Statistics in Medicine*.

Oehlert, G. W. (1992). A Note on the Delta Method. *The American Statistician*, 46(1):27–29.

Pashley, N. E., Keele, L., and Miratrix, L. W. (2024). Improving instrumental variable estimators with poststratification. *Journal of the Royal Statistical Society Series A: Statistics in Society*, page qnae073.

Pedersen, T. L. (2024). *patchwork: The Composer of Plots*. R package version 1.3.0.

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.

Pustejovsky, J. E. (2018). The multivariate delta method.

Robert, C. and Casella, G. (2010). *Introducing Monte Carlo Methods with R*. Springer, New York, NY.

Rosenbaum, P. R. (2017). *Observation and experiment: An introduction to causal inference*. Harvard University Press.

Rousseeuw, P. J. and Croux, C. (1993). Alternatives to the Median Absolute Deviation. *Journal of the American Statistical Association*, 88(424):1273–1283.

Satterthwaite, F. E. (1946). An approximate distribution of estimates of variance components. *Biometrics Bulletin*, 2(6):110.

Shaw, P. A., Gruber, S., Williamson, B. D., Desai, R., Krakauer, C., Nelson, J. C., van der Laan, M. J., and Shortreed, S. M. (2025). A cautionary note for plasmode simulation studies in the setting of causal inference.

Siepe, B. S., Bartoš, F., Morris, T. P., Boulesteix, A.-L., Heck, D. W., and Pawel, S. (2024). Simulation studies for methodological research in psychology: A standardized template for planning, preregistration, and reporting.

Staiger, D. O. and Rockoff, J. E. (2010). Searching for effective teachers with imperfect information. *Journal of Economic perspectives*, 24(3):97–118.

Tipton, E. (2013). Stratified sampling using cluster analysis: A sample selection strategy for improved generalizations from experiments. *Evaluation Review*, 37(2):109–139.

Tipton, E. and 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.

Tufte, E. R. and Graves-Morris, P. R. (1983). *The visual display of quantitative information*. Graphics press Cheshire, CT.

Vevea, J. L. and Hedges, L. V. (1995). A general linear model for estimating effect size in the presence of publication bias. *Psychometrika*, 60(3):419–435.

Welch, B. L. (1951). On the comparison of several mean values: An alternative approach. *Biometrika*, 38(3/4):330.

Westfall, P. H. and Henning, K. S. (2013). *Understanding advanced statistical methods*, volume 543. CRC Press Boca Raton, FL.

White, H. (1980). A heteroskedasticity-consistent covariance matrix estimator and a direct test for heteroskedasticity. *Econometrica*, 48(4):817–838.

Wickham, H. (2014). Tidy data. *Journal of Statistical Software*, 59(10):1–23.

Wickham, H. and Bryan, J. (2023). *R Packages*. "O'Reilly Media, Inc.".

Wickham, H., Çetinkaya-Rundel, M., and Grolemund, G. (2023). *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. "O'Reilly Media, Inc.".

Wilcox, R. R. (2022). *Introduction to Robust Estimation and Hypothesis Testing*. Academic Press, an imprint of Elsevier, London, United Kingdom San Diego, United States Cambridge, MA Oxford, United Kingdom, fifth edition edition.

Zhou, J., Zhang, Z., Li, Z., and Zhang, J. (2015). Coarsened propensity scores and hybrid estimators for missing data and causal inference. *International Statistical Review*, 83(3):449–471.

A

Coding Reference

In this appendix chapter we give a bit more detail on some core programming skills that we use throughout the book.

A.1 How to repeat yourself

At the heart of simulation is replication: we want to do the same task over and over. In this book we have showcased a variety of tools to replicate a random process. In this section we give a formal presentation of these tools.

A.1.1 Using `replicate()`

The `replicate( n, expr, simplify )` method is a base-R function, which takes two arguments: a number `n` and an expression `expr` to run repeatedly. You can set `simplify = FALSE` to get the output of the function as a list, and if you set `simplify = TRUE` then R will try to simplify your results into an array.

For simple tasks where your expression gives you a single number, replicate will produce a vector of numbers:

```
replicate( 5, mean( rpois( 3, lambda = 1 ) ) )
```

```
[1] 0.6666667 1.0000000 1.6666667 0.6666667 0.3333333
```

If you do not simplify, you then will need to massage your results:

```
one_run <- function() {
  dd = rpois( 3, lambda = 1 )
  tibble( mean = mean( dd ), sd = sd( dd ) )
}
rps <- replicate( 2, one_run(), simplify = FALSE )
rps
```

```
[[1]]
```

```
# A tibble: 1 x 2
   mean    sd
  <dbl> <dbl>
1  1.33 0.577

[[2]]
# A tibble: 1 x 2
   mean    sd
  <dbl> <dbl>
1  2.33 0.577
```

In particular, you will probably stack all your tibbles to make one large dataset:

```
rps <- bind_rows( rps )
rps
```

```
# A tibble: 2 x 2
   mean    sd
  <dbl> <dbl>
1  1.33 0.577
2  2.33 0.577
```

Note that you give replicate a full piece of code that would run on its own. You can even give a whole block of code in curly braces. This is exactly the same code as before:

```
rps <- replicate( 2, {
  dd = rpois( 3, lambda = 1 )
  tibble( mean = mean( dd ), sd = sd( dd ) )
}, simplify = FALSE ) %>%
  bind_rows()
```

The `replicate()` method is good for simple tasks, but for more general use, you will probably want to use `map()`.

A.1.2 Using `map()`

The tidyverse way of repeating oneself is the `map()` method. The nice thing about `map()` is you map over a list of values, and thus can call a function repeatedly, but with a shifting set of inputs.

```
one_run_v2 <- function( N ) {
  dd = rpois( N, lambda = 1 )
  tibble( mean = mean( dd ), sd = sd( dd ) )
}
n_list = c(2, 5)
rps <- map( n_list, one_run_v2 )
```

```
rps
```

```
[[1]]
# A tibble: 1 x 2
   mean    sd
  <dbl> <dbl>
1     2  1.41

[[2]]
# A tibble: 1 x 2
   mean    sd
  <dbl> <dbl>
1   0.4 0.548
```

You again would want to stack your results:

```
bind_rows(rps)
```

```
# A tibble: 2 x 2
   mean    sd
  <dbl> <dbl>
1   2   1.41
2   0.4 0.548
```

We have a small issue here, however, which is we lost what we *gave* `map()` for each call. If we know we only get one row back from each call, we can add the column directly:

```
rps$n = n_list
```

A better approach is to *name* your list of input parameters, and then your `map` function can add those names for you as a new column when you stack:

```
n_list %>%
  set_names() %>%
  map( one_run_v2 ) %>%
  bind_rows( .id = "n" )
```

```
# A tibble: 2 x 3
  n      mean    sd
  <chr> <dbl> <dbl>
1 2       1   1.41
2 5       0.6 0.894
```

An advantage here is if you are returning multiple rows (e.g., one row for each estimator tested in a more complex simulation), all the rows will get named correctly and automatically.

In older tidyverse worlds, you will see methods such as `map_dbl()` or

`map_dfr()`. These will automatically massage your output into the target type. `map_dfr()` will automatically bind rows, and `map_dbl()` will try to simplify the output into a list of doubles. Modern tidyverse no longer likes this, which we find somewhat sad.

To read more about `map()`, check out out Section 21.5 of R for Data Science (1st edition), which provides a more thorough introduction to mapping.

A.1.3 map with no inputs

If you do not have parameters, but still want to use `map()`, you can. E.g.,

```
map_dfr( 1:3, \(.) one_run() )
```

```
# A tibble: 3 x 2
   mean    sd
  <dbl> <dbl>
1 0.667 1.15
2 1.33  0.577
3 1.33  1.15
```

The weird "(.)" is a shorthand for a function that takes one argument and then calls `one_run()` with no arguments. We are using the 1:3 notation to just make a list of the right length (3 replicates, in this case) to map over. A lot of fuss! Just use `replicate()`

To make all of this more clear, consider passing arguments that you manipulate on the fly:

```
map_dfr( n_list, \(x) one_run_v2( x*x ) )
```

```
# A tibble: 2 x 2
   mean    sd
  <dbl> <dbl>
1  0.75 0.957
2  1.24 1.05
```

Anonymous functions, as these are called, can be useful to connect your pieces of simulation together.

A.1.4 Other approaches for repetition

In the past, there was a tidyverse method called `rerun()`, but it is currently out of favor. Originally, `rerun()` did exactly that: you gave it a number and a block of code, and it would rerun the block of code that many times, giving you the results as a list. `rerun()` and `replicate()` are near equivalents. As we saw, `replicate()` does what its name suggests—it replicates the result of an expression a specified number of times. Setting `simplify = FALSE` returns

the output as a list (just like `rerun()` did).

A.2 Default arguments for functions

To write functions that are both easy to use and configurable, set default arguments. For example,

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

my_function()
```

```
[1] 1020
```

```
my_function( 5 )
```

```
[1] 520
```

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

```
[1] 1005
```

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

```
[1] 105
```

We can 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:

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

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

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

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

```
   B          Y0       Y1
1 B1  0.4715667 5.795944
2 B1 -1.5834016 4.362639
3 B1 -0.2669267 4.896446
4 B2 -1.5431692 2.681150
5 B2  0.7432187 5.191804
```

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

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

```
args( generate_blocked_data )
```

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

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

A.3 Profiling Code

Simulations can be extremely time intensive. With a large simulation it can also be hard to determine *why*, exactly, the simulation is as long as it is. Is it one of the methods? Is it just that as sample size grows, the time grows far more rapidly than one might expect? Knowing the answer to these questions can allow you to plan out your simulation and, sometimes, make some hard choices as to what things you want to include.

There are a variety of tools for timing code. We are going to go through a few useful ones here.

A.3.1 Using `Sys.time()` and `system.time()`

The simplest way to time code is to use the `system.time()` function. This function takes an expression and returns the time it took to run that expression. For example:

```
time <- system.time( rnorm( 1000000 ) )
time
```

```
   user  system elapsed
  0.036   0.001   0.037
```

Elapsed time is how much time actually passed. The user time is how much time your computer spent on the task at hand, not including if it paused to do something else (like deal with a mouse click or pop-up message). With current computers with multiple cores, you would expect user and elapsed time to be very similar.

You can also start and stop a clock by checking the system time:

```
start_time <- Sys.time()
A = rnorm( 1000000 )
mean(A)
```

```
[1] 1.699463e-05
```

```
sd(A)
```

```
[1] 0.9993256
```

```
end_time <- Sys.time()
tot_time <- end_time - start_time
tot_time
```

```
Time difference of 0.05163932 secs
```

This can be useful, but be careful, as the time is stored along with the units. If your simulation takes a long time, it will flip from a lot of seconds to a few minutes, and you can end up thinking something that took much, much longer was actually fast.

A.3.2 The `tictoc` package

The `tictoc` package is a very simple way to time code. It has two functions, `tic()` and `toc()`, which you can use to mark the start and end of a block of code.

```
library( tictoc )
tic("Generating data")
A = rnorm( 1000000 )
toc()
```

```
Generating data: 0.041 sec elapsed
```

It is basically like `Sys.time()` but it has a few more features, such as being able to label the time you are measuring.

A.3.3 The `bench` package

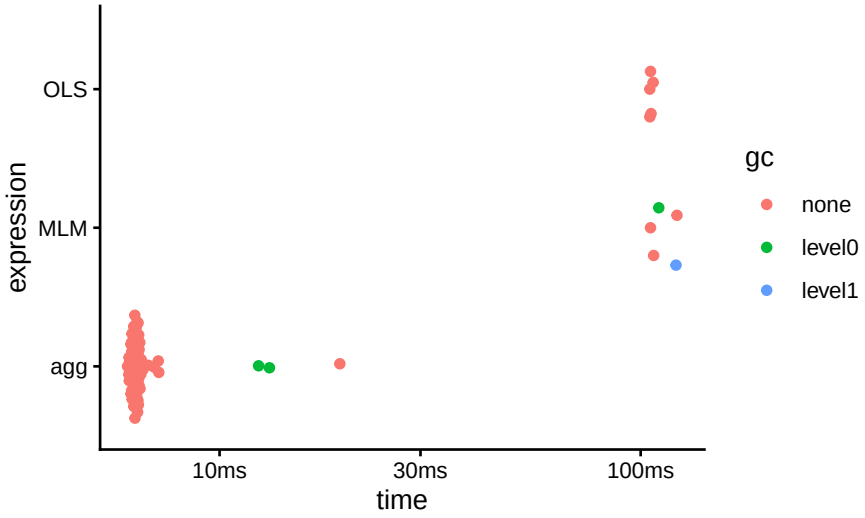
The `bench` package provides some powerful tools for timing code, and is in particular good for comparing different ways of doing the same (or similar) thing. `bench::mark()` runs each expression 10 times (by default) and tracks how long the computations take. It then summarizes the distribution of timings. For example, we can time how long it takes to analyze some data from our cluster RCT DGP from Section 6.6:

```
library( bench )
dat <- gen_cluster_RCT(n_bar=100, J = 100)
timings <- mark(
      MLM = analysis_MLM_safe(dat),
      OLS = analysis_OLS(dat),
      agg = analysis_agg(dat),
      check = FALSE
)
timings
```

```
# A tibble: 3 x 6
  expression      min   median `itr/sec` mem_alloc `gc/sec`
  <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
1 MLM        105.53ms 107.36ms      8.96   28.48MB     5.97
2 OLS        105.11ms 105.53ms      9.46    3.27MB     0
3 agg          6.05ms   6.31ms    153.      2.46MB     4.20
```

You can even get a viz of how long everything took:

```
plot( timings )
```



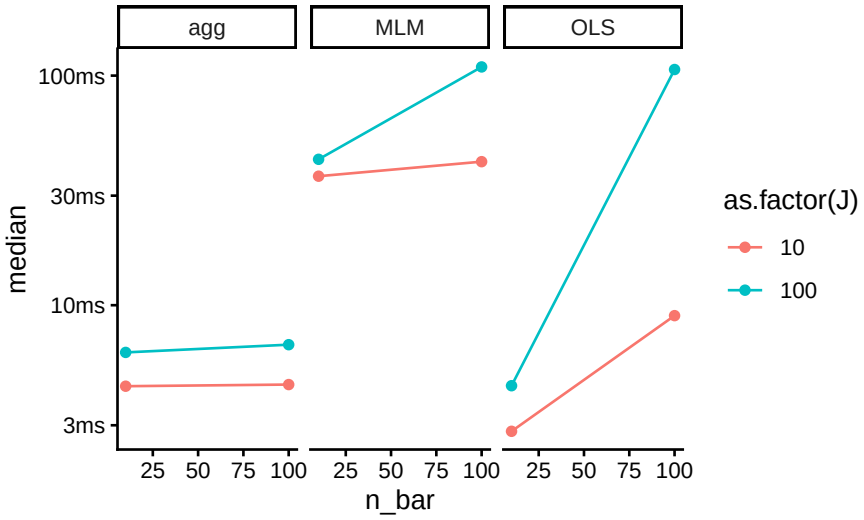
The "gc" coloring in the above indicates runs where "garbage collection" took place, meaning R paused to empty out some used memory.

You can also use `bench::press()` to run a variety of configurations to explore how timing works under changed parameters. To illustrate, let's compare how long each method for the cluster RCT running example takes:

```
source( here::here("case_study_code/clustered_data_simulation.R" ) )
timings <- bench::press(
  n_bar = c( 10, 100 ),
  J = c( 10, 100 ),
  {
    dat <- gen_cluster_RCT(n_bar=n_bar, J = J)
    bench::mark(
      MLM = analysis_MLM_safe(dat),
      OLS = analysis_OLS(dat),
      agg = analysis_agg(dat),
      check = FALSE
    )
  }
)
```

And we can make our custom plot of time:

```
timings$name = attr( timings$expression, "description" )
ggplot( timings, aes( n_bar, median, color = as.factor(J) ) ) +
  facet_wrap( ~ name ) +
  geom_point() + geom_line()
```



Note the `timings` object is not quite a classic tibble, and the expression at start captures the code run. The "name" line grabs the names given in the initial evaluation so we can make the plot the way we want.

A.3.4 Profiling with `profvis`

Sometimes you don't have a head to head comparison in mind, but are instead trying to find out *where* in your full simulation the time is being spent. The `profvis` package allows for exploring this sort of question. You "profile" a block of code and then you can explore the results in a browser inside of RStudio.

For example, you might have the following:

```
profvis( {
  replicate( 10, {
    data <- gen_cluster_RCT( n_bar = 100, J = 100 )
    res1 <- analysis_MLM_safe(data)
    res2 <- analysis_OLS(data)
    res3 <- analysis_agg(data)
  } )
} )
```

In the browser you will get the code you ran, and you can see how long each line took to run. You will get little "time" bars—the bigger the bar, the greater fraction of time that line took versus the other lines.

You can also click on the "Data" tab and it will give you a series of cascades of function calls, so you can see how long each function took to run. You click

on a function to expand it, and it will show you how long each part inside took.

A.4 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, generally optimize code only after discovering 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 will not help shorten the overall run time of a simulation 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.

Overall, computational efficiency should usually be a secondary consideration when you are starting to design a simulation study. It is better to produce accurate code, even if it is a bit slow, than to write code that is speedy but hard to follow (or even worse, that produces incorrect results).

That warning made, in the next sections we will look at a few optimization efforts applied to the ANOVA example from Chapter 5 to illustrate some principles of optimization that come up a lot in simulation projects.

A.4.1 Hand-building functions

In our initial ANOVA simulation (see Chapter 5) 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 write our own function as so:

```
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)
}
```

We are using data as generated in Chapter 6. To see the difference between our version and R's version, we benchmark:

```
timings <- bench::mark(Rfunction = ANOVA_F_aov(sim_data),
                       direct    = ANOVA_F(sim_data))
summary( timings )[1:4] %>%
  mutate( speed_up = as.numeric( max(median)/median ) ) %>%
  knitr::kable( digits = 2 )
```

expression	min	median	itr/sec	speed_up
Rfunction	495us	534us	1807.74	1.00
direct	224us	240us	3840.55	2.23

The direct function is 2.2 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!

A.4.2 Computational efficiency versus simplicity

On the data generation side, 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,] 1.9454613 -0.8216704 -0.760806  1.4374452
 [2,] 1.8228633  0.2817601  1.627944  1.4093370
 [3,] 1.2994653  0.2945038  2.103286  2.2914832
 [4,] 2.1675659  2.8035855  1.870356  3.5454638
 [5,] 1.6408437  3.3318438  4.434477 -0.5136008
 [6,] 0.9718801  4.0079770  2.803974  3.1613009
 [7,] 3.6294955  3.0036280  2.860671  0.7946165
 [8,] 1.9145808  3.4543692  2.426922  2.9275181
 [9,] 1.7609879  2.4777368  1.379899  1.6096698
[10,] 9.2491482  1.8205699  7.091377  2.7410474
[11,] 4.0784544  3.7242153  4.112892  2.8034151
[12,] 4.5819584  6.5380988  4.835205  4.9001549
[13,] 5.3154103  5.7915971  5.841061  5.8426595
```

```
[14,] 5.8789499  7.3502369  6.743593  6.0738525
[15,] 5.9919054  6.0309925  7.199642  4.4479573
```

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). It is usually the case that most time spent in the simulation is the *analyzing* of the data, not the generating it, so these savings are usually not worth the bother.

A.4.3 Reusing code to speed up computation

Once we have our own ANOVA method to go with our own Welch method, we see some glaring redundancies. In particular, both `ANOVA_F` and `Welch_F` start by taking the simulated data and calculating summary statistics for each group, using the following code:

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

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

```
summarize_data <- function(sim_data) {
  x_bar <- with(sim_data, tapply(x, group, mean))
  s_sq <- with(sim_data, tapply(x, group, var))
  n <- table(sim_data$group)

  list(
    x_bar = as.numeric( x_bar ),
    s_sq = as.numeric( s_sq ),
    n = as.numeric( n )
  )
}
```

We just packaged the code from above:

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

```
$x_bar
[1] 0.5778353 1.6654075 4.7351691 6.0655680

$s_sq
[1] 1.011576990 2.188110429 0.003579253 1.115150538

$n
[1] 3 6 2 4
```

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

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

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

  return(pval)
}

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

```
[1] 0.0002412255
```

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

```
[1] 0.002926864
```

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 0.0000740 0.0113
```

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

```
timings <- bench::mark(original = one_run(mu = mu, sigma_sq = sigma_sq,
                                          sample_size = sample_size),
                       one_agg = one_run_fast(mu = mu, sigma_sq = sigma_sq,
                                              sample_size = sample_size),
                       check=FALSE)
timings[1:4] %>%
  mutate( speed_up = as.numeric( max(median) / median ) ) %>%
  knitr::kable( digits = 2 )
```

expression	min	median	itr/sec	speed_up
original	1.56ms	1.61ms	610.85	1.00
one_agg	1.28ms	1.35ms	733.01	1.19

Our improvement is fairly modest.

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.

That said, optimizing code is dangerous (it is an easy way to introduce bugs into your simulation workflow) and time consuming for you (thinking through and writing all the fancy code is time you are not doing something else). But it sure can be satisfying to spend an extra weekend hacking away at this sort of thing!

Index

assignment mechanism, 400

Bias-SE-RMSE plot, 286
browser(), 360
bundle_sim(), 62, 150, 211

calc_absolute(), 195
calc_coverage(), 195
calc_rejection(), 195
calc_relative_var(), 195
calibrated simulation, 383
calibrated simulations, 409
case
    cluster RCT, 304
cluster-randomized trial, 81, 116, 156
copula, 326
create_skeleton(), 59
CRT, *see* cluster-randomized trial

data-generating process, 16, 22, 38, 49, 67, 68, 70, 79, 90, 113, 116, 119, 145, 155, 159, 160, 224, 298, 321, 323, 364, 400
debug(), 358
debugging code, 149, 357
DGP, *see* data-generating process

errors and warnings, 127, 318
evaluate_by_row(), 219
example
    Biserial correlation, 299
    bivariate Poisson, 80, 85, 86, 105, 107, 114, 121, 145, 146, 151, 152, 156, 201, 215, 227, 298
    cluster randomized trial, 90, 116, 140, 147, 156, 164, 172, 175, 179, 184, 234, 269, 310, 428
    heteroskedastic ANOVA (Welch), 65, 80, 104, 120, 138, 156, 200, 208, 431
    meta-regression, 111, 142, 243, 256
    t-test simulation, 41, 64

functions
    custom written, 30
future package, 148, 178

GitHub, 379

HLM, *see* multilevel model

IRT, *see* item response theory
item response theory, 82, 141

list columns, 213

MCSE, *see* Monte Carlo standard error
meta-regression, 315
Monte Carlo simulation, 15, 31, 41, 55, 76, 153, 161, 246, 280, 312, 338, 369
Monte Carlo standard error, 44, 161, 190, 273, 294, 312, 338
multilevel model, 81

package
    furrr, 370
    future, 370

simhelpers, 59, 62, 74, 150, 195, 211
parallel processing, 237, 345, 370
parametric bootstrap, 26
performance measures
bias, 159
calibration, 169
coverage, 160, 270
median bias, 167
RMSE, 240, 270, 340
RMSE, 160, 226
trimmed mean bias, 167
Type I error, 67, 160, 337
winsorized bias, 168
winsorized RMSE, 168
winsorized variance, 168
plasmode simulation, 324
potential outcomes framework, 26, 399, 426
power analysis, 19, 178, 383
print(), 357
pseudo-random number generator, 152
random seed, 145
randomized controlled trial, 383
RCT, *see* randomized controlled trial
reparameterization, 96, 149, 156
repeat_and_stack(), 74

sbatch, 375
schedule of potential outcomes, 401
SE, *see* standard error
semi-synthetic simulation, 324
simulation driver, 150, 207, 354
simulation study, 22, 49, 65, 90, 153, 334, 431
single factor simulation, 208
skeleton, 36
slurm, 374
stopifnot(), 108, 149, 361

tidy design, 26, 63, 224, 350

validity, 390