Best coding practices to ensure reproducibility[*]

Gonzalo Rivero   Jiating (Kristin) Chen
Statistics and Evaluation Sciences Unit, Westat

August 6, 2020 (Version 0.9.8)

1 Introduction

Practical work in statistics—more often than not— involves coding. As a result, both statisticians and data scientists not only need to write code but are expected to write *good* code. By "good code" we mean code that, needless to say, makes the computer perform the operations that we intended but also code that is easily understood by humans—code that other people can reason about, edit, and reuse if necessary. If we adopt this view, our goal should be to write statistical programs that strike a balance between effectiveness and readability, an aim that is not always easy to achieve given the lack of formal training in software development of most practitioners, the sensible inclination to prioritize deadlines over potential future readers, and the subjective and social nature of the problem itself. In this document, guided by our own experience, we discuss what we believe are some of the core challenges that data scientists and statisticians face when we need to share our code and offer suggestions of possible tools and workflows.

We make these recommendations with an eye towards *reproducibility*. The concept is key in modern scientific research and has been at the center of the debate around the *credibility revolution* in Academia, although it affects applied researchers, too (see, for instance, Stodden, Seiler, and Ma 2018). At the end of the day, the extent to which consistent results can be obtained in repetitions of an analytical task is a minimum standard for judging a claim (Peng 2011) which is a way of saying that reproducibility is a warranty that we extend on the output of our statistical operations. When we share our results with others we are making a implicit commitment that, if asked to, we would be able to reproduce the same output starting from the same input and that others can verify the process. With that idea in mind, we should consider that the code that produces the results—even more than the results themselves—is the actual

[*]We are indebted to Andreea Erciulescu, Tom Krenzke, Jeri Mulrow, Benjamin Schneider, and Marcelo Simas for their comments to earlier versions of this manuscript comments and to the participants in the seminar 'Reproducibility in Quantitative Research' held at Westat.

product of research activity (McElreath 2018, 443). Our goal in preparing this document is to make the process of writing reproducible code easier so that our colleagues, team mates, clients, and the scientific community at large can use our results and methods (Barnes 2010; Martinez et al. 2018). In this, we acknowledge that *practical* reproducibility, by which we mean "enabling others to reproduce results *without difficulty*" (Project Jupyter et al. 2018, 113, the emphasis is ours), is still the main barrier to the adoption of a reproducibility framework.

We have divided our recommendations in two parts. In the first one, we talk about good coding practices for statistics and data science. We make suggestions that are fundamental enough to cover all types of statistical products. In the case of analysis and modeling, the final output may be a report that perhaps will need to be revisited as data changes or as we try different modeling approaches. In this case, goodf code is code that makes it possible for other users—and ourselves in the future—, not only to verify what has been done but also make contributions. When we produce survey estimates, good code means that clients can inspect and reproduce which will imply also the ability to run our code themselves. When the goal is to produce statistical tools, we are squarely in the terrain of software development and our standards should align with those that are common among software engineers. In all these cases, our code will benefit from following some of the standard practices we list.

The second part of the document focuses on reproducibility and the tools that we can use to make our code more stable, portable, and easier to share (Boettiger 2015). Here our target is to facilitate a workflow that allows us to provide our users with a way to replicate the computation environment (Tatman, VanderPlas, and Dane 2018). In this discussion, it is important to keep in mind that there is more than one way to make our code reproducible and that the standard of reproducibility we choose to apply should be a negotiation between us and our users. To put it in a different way, the recommendations in this second section are not intended to be taken as a monolith but rather as suggestions that can be combined modularly depending on the specific needs of the project.

In both parts, our recommendations emphasize tools from the ecosystem of the `R` language. However, we believe that the overarching ideas apply more generally to other open source tools.

A final note is in order. While we should certainly use whatever tool can help us, it is our view that reproducibility is first and foremost a problem of *workflow and process*—of how we use the tools, of how we structure and develop a project, of adopting habits that push us to think about others when writing code. From this point of view, we are persuaded that reproducibility is not a burden but rather a beneficial concept that can help us increase our productivity and confidence in what we do (Sandve et al. 2013) by adopting a mode of thinking that prompt us to put ourselves in the shoes of others.

1.1 Good coding practices improve reproducibility

It does not matter if we are writing a one-off analysis or a module that will be used by others, we can benefit from having good tools that help us simplify the operations that are involved in making our code and environment reproducible by others, as we will see in the next section. However, the challenge we face is often not only technical but rather one of technique. What separates good code from bad code is, to a large extent, how the information is organized and conveyed. Our position is that, given a choice among alternative ways of writing a program, we should pursue the one that others will find easier to interpret. Thus, the idea of "good" code will revolve around an idea of *style,* which depends on our individual skills and creativity, but also on an opportunistic adherence to conventions, which we see here as an agreement between all users about how to coordinate in situations that admit several solutions. In that regard, good coding practices are those that aim at reducing the cognitive effort required to understand the intention of the code and how it operates (Wilson et al. 2014).

1.1.1 Use idioms and conventions

A consistent coding style is like orthography or punctuation: it makes code easier for others to read. Simple things like conventions on how to name functions or objects, what the maximum length of a line should be, or where to align the brackets that enclose the body of a function go a long way to make our code look like familiar territory to others. If possible, you should favor idiomatic alternatives (see, for instance, Johnson 2019) to consistently adopt a style guide, such as the Tidyverse Style Guide. To make the process easier, you should use code checkers such as `lintr` that help you stay on track with the guide and to spot potential errors.

1.1.2 Avoid assumptions about the execution environment

It is convenient to work in a project assuming that it will be executed by others in a fresh session which means that they will not have access to artifacts or customizations from your personal environment (Blischak et al. 2016). At the same time, be mindful of the changes that your code will make in someone else's session (Bryan 2017): it is tempting to force the user to comply with our expectations about what resources are available and where but there are often better solutions. For instance, if you want to make sure that the user has a package installed, instead of stating `install.packages()` in your script, provide them with an `renv` file. Similarly, rather than setting a working directory in your script, which is a way of making potentially unrealistic assumptions about the folder structure of other people's machines (Wickham and Grolemund 2017), use paths relative to the home of the project. Finally, avoid making unnecessary changes to the environment of the user by calling `rm(list=ls())`. It is hard to predict what you will be unnecessarily removing from your machine or from the user's machine.

1.1.3 Structure your code in predictable way

Structure your code in a standard and predictable way so that it is easy for any reader to know where to find things. For instance, have separate folders for code, data, or images with easily recognizable and standard names and split your code at different logical points as opposed to sourcing everything from a single script. Does your script clean data, run analysis, and create figures? As a general rule, we think it is cleaner and easier to understand to have separate scripts performing specific tasks. The same idea applies to functionality. Ensure that your code abstracts and automates functionality in a way in which each piece does one thing well and one thing only (Salus 1994). Does your function pull data from a data base and reshapes it? Would it be easier to understand if you split it so that data extraction and data transformation are two separate (even if connected) steps?

1.1.4 Document everything

Documentation is the entry point to our code for a new person but also for our future selves. Make sure that each logical unit (scripts, functions, packages) includes a short description of what they do, leaving a trace of your thinking process if some decisions are not obvious from the code itself. Better than describing what the code does, document why the code does it in a particular way. Keep in mind that whatever seems trivial to you now may not still be obvious one year from now.

In packages, document classes, methods, and functions using documentation generators like `roxygen2` making sure you are explicit about what comes in and what comes out. In the case of scripts, if you have a reason to not maintain your code under version control (see next section), include the author and date of the current version and archive older versions with a consistent and searchable file name.

We recommend using `README` files to help your reader understand what your code does, how it works, and what to expect from it. Think about a new user and give them a high level explanation about what the code does, how one uses it or the expectations about the resources that are available. We have found that the most useful `README` files are explicit, use examples, and a walk-through to help users understand how to use the code. Because credentials and personal information should never be hardcoded (Csefalvay 2018), a `README` file is also the best place to document how passwords are expected to be passed to the code whenever they are needed.

1.1.5 Be defensive against errors

Be stingy with dependencies. If you only need one function from a package, avoid importing everything from it. Similarly, if what you need to do can easily be expressed with the base distribution of `R`, do not import additional packages, especially if they are not commonly used or regularly maintained.

Errors creep in during manual operations like copying data or results from one tool to another. It is too easy to copy the wrong input or to forget to update a report with the most recent numbers. If your code is part of an analysis that is intended to be read with comments, tables, or figures, use a notebook like R Markdown.

Tested code is more reliable code. In thinking about testing, we should consider whether each individual component is fit for use (unit testing), whether different components work correctly together (integration testing), and whether the system as a whole behaves as expected (system testing). Each level follows a similar logic: for every piece of code we write, we commonly are able to anticipate what must happen for a given input. For instance, if we write a function to perform addition and the inputs are 1 and 1, we can manually work out what a valid implementation must return (unit testing). In some other cases, however, all we can do is express high level expectations. If we implement a function to make extractions out of a uniform distribution in $[0, 1]$, we cannot possible predict the exact return for each call but we will know that the implementation is wrong if it is not a float in the $[0, 1]$ interval. Making these expectations as explicit as possible and incorporating them into the code using `testthat` or `assertr` as part of the design can allow others to verify that the code behaves as intended (Wilson et al. 2014).

A complete strategy to ensure that the code meets the requirements and that it correctly does what we want often requires more than the adoption of good individual workflows and tools. Instead, it often depends on incorporating a formal process for development that favors transparency, introduces periodic reviews by peers, and builds upon tests as the foundation of the development strategy. The different approaches to manage the life-cycle of the code—including assessment, testing, and quality assurance—are outside the scope of this document but we encourage the reader to consult a high-level overview like Bourque and Fairley (2014).

2 A toolkit for reproducibility

Reproducibility speaks to the computational dimension of statistical research and practice. In the context of statistics and data science, reproducibility means that our code—a map from data to estimates or predictions—should not depend on the specific computational environment in which data processing and data analysis originally took place.

The challenge that we need to address is the following. The packages we use undergo changes through different iterations. Sometimes these changes are not backwards compatible which means that the same snippet of code can change its meaning in the future because we, as package developers, may alter what a function does in order to better suit our own needs. In addition, packages have dependencies themselves which may introduce changes in our code indirectly.

Without being a call to remove *all* dependencies, it is helpful to see things from the perspective that "[d]ependencies are invitations for other people to break your package" (Eddelbuettel 2018). Thus, our first goal is to ensure that we will be able to replicate in the future the exact network of dependencies that we used today to run our analysis.

The same concept can be rolled up to the level of the statistical environment itself. `R` changes and sometimes the changes affect the original intention of our code. The `R` environment may introduce effects through a subtle mechanism: computations can be affected by *how* R was installed—some routines may not work or work unexpectedly depending on system-level dependencies. Consequently, we will want to ensure that our user is aware that our code is tied to a particular version of `R` and provide them with a way of running the code under conditions as similar as possible as the ones we originally used.

Finally, even with the same computational environment, results may be different depending on the version of the code and the order in which it was executed. Say that, for instance, our project uses data that is processed through three scripts `A`, `B`, and `C` in sequence. Is it possible that running `B` before `A` will change the output that `C` needs? How can we make sure that the user knows what needs to be run and in which order to reproduce our results? What is more, knowing that those three scripts can change over time, how can we make sure that the user is indeed looking at the correct version of the code *and* data?

The recommendations below address these issues. Although they favor specific tools, it is important to be aware that the toolkit that is available to us is constantly evolving and that, as a result, the best tools are likely to change over time. Keeping an eye on the literature and on what our scientific community is doing is thus as important as mastering specific tools. Also, common sense is the best guiding principle: not all projects face the same demands and, rather than blindly applying the same solution to all possible scenarios, we recommend being thoughtful about the combination of tools that best serve the project at hand.

2.1 Embrace version control

Our code evolves throughout the life of the project. It is expected—and even trivial to say—that specific results are tied to specific versions of the code and the data. Being able to go back and forth between these different versions helps us declare the provenance of a given result. Have we changed the way a variable is recoded? Did we use a different model? Was the data edited to correct an error? Having the code in version control system such as `git` will help us keep track of different versions of the code without having to rely on *ad hoc* solutions like renaming files with dates or the author's initials. Suites like Github or GitLab offer integration of version control systems with other tools that can be used for issue tracking, which we have found useful to simplify the process of discussion and documentation. While not as standard yet, we want to mention that are tools available to maintain the data itself under version control such as `dvc` even although, with a goal of data sharing in mind, the challenge is more likely on the side of standards (Tierney and Ram 2020).

2.2 Document code dependencies

We recommend the use of a dependency manager to ensure that you document the exact versions of the packages that were installed when you ran the code for the last time. For `R`, `renv` is a good choice. `renv` will create a fresh library for your project and then incrementally add the packages that you need to a `.lock` file (a JSON file) that lists not only the version that you installed but also the source from which it was installed (Did you install version 2.2.1 of `devtools` from CRAN or did you install it from commit `00bc51e` from GitHub?). Sharing the environment you used for your analysis can then be as simple as distributing the `.lock` file along with your code so that any user can install the exact same dependencies you used.

2.3 Simplify code execution

If your analysis spans more than one script you will want to have a tool that helps you declare:

1. the order in which scripts are supposed to be run,
2. the input that each script expects, and
3. the output that each output produces

By describing the steps that go from the raw data to the deliverable as a concatenation of inputs and outputs connected by different scripts, tools like `make` or `drake` help you ensure that user that repeat the exact same process that you used. Do you first need to run script `A` and then script `B` before the input that `C` uses is ready?

As an additional advantage, because pipeline managers know whether the input to a script has changed, they can save time by deciding if something needs to be re-run or not. If the input to script `B` has not changed and `C` only uses that output from `B` then there is no need to execute `B` or `C` again. This behavior is exceptionally useful whenever the code in `B` takes a very long time to execute.

2.4 Declare your infrastructure as code

Sometimes a project will depend on a specific version of `R`. Perhaps the same code will produce different results in a different version of `R` because the behavior of some function has changed. Or maybe it will not work because some dependencies are only available for a particular version of `R`. In those cases, it is helpful to have an isolated environment in which you can install and configure the specific version of `R` that you need and ship to the user. Using Docker is a good idea if you need guarantee the portability of the analysis so that the user can stand up the same computational environment that you used—all the way down to the operating system. More importantly, you can declare the computational environment *as code* so that replicating it does not need detailed instructions for a human to execute.

The `rocker` team generously provides access to a repository of Dockerfiles to build images using different versions of `R` and, even more helpful, different standard configurations of `R` so that you do not need to install all the dependencies that are required for the `tidyverse` or LaTeX/`pandoc` to work.

3 Conclusions

In this document, we have tried to make the case that reproducibility is an evolving, collaborative enterprise between the members of the research community at large. The recommendations we have made in the previous pages take the view that reproducibility is largely a combination of automation (to reduce the number of manual tasks that are required to execute the code correctly) and thoughtfulness (to ease the burden that is required from users to understand what the code does). In that regard, reproducibility in the context of statistics and data science can take advantadge of the practices that are common in software engineering even if we need to understand that, in our role as professionals in the collection, analysis, and interpretation of data, we face distinct challenges due to the nature of the artifacts we interact with (i.e., data), the type of output we produce (i.e., estimates), and our own technical background and priorities. In that regard, we think it is important to adopt a perspective in which good tools that favor computational reproducibility *complement* good practices and processes that make research more transparent and, consequently, more accessible to other people.

References

Barnes, Nick. 2010. "Publish Your Computer Code: It Is Good Enough." *Nature* 467 (7317): 753–53.

Blischak, John, Daniel Chen, Harriet Dashnow, and Denis Haine. 2016. "Software Carpentry: Programming with R." http://swcarpentry.github.io/r-novice-inflammation.

Boettiger, Carl. 2015. "An Introduction to Docker for Reproducible Research, with Examples from the R Environment." *ACM SIGOPS Operating Systems Review, Special Issue on Repeatability and Sharing of Experimental Artifacts.* 49 (1): 71–79.

Bourque, Pierre, and Richard E Fairley. 2014. *Guide to the Software Engineering Body of Knowledge (SWEBOK: Version 3.0).* IEEE Computer Society Press.

Bryan, Jenny. 2017. "Project-Oriented Workflow." https://www.tidyverse.org/blog/2017/12/workflow-vs-script/.

Csefalvay, Chris von. 2018. "Structuring R Projects." https://chrisvoncsefalvay.com/2018/08/09/structuring-r-projects/.

Eddelbuettel, Dirk. 2018. "#17: Dependencies." http://dirk.eddelbuettel.com/blog/2018/02/28/#017_dependencies.

Johnson, Paul E. 2019. "Rchaeology: Idioms of R Programming," March. https://pj.freefaculty.org/R/Rchaeology.pdf.

Martinez, C., J. Hollister, B. Marwick, E. Szocs, S. Zeitlin, B. Kinoshita, S. Wykstra, J. Leek, N. Reich, and B. Meinke. 2018. "Reproducibility in Science: A Guide to Enhancing Reproducibility in Scientific Results and Writing." http://ropensci.github.io/reproducibility-guide.

McElreath, Richard. 2018. *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*. CRC press.

Peng, Roger D. 2011. "Reproducible Research in Computational Science." *Science* 334 (6060): 1226–7.

Project Jupyter, M Bussonnier, J Forde, J Freeman, B Granger, T Head, C Holdgraf, et al. 2018. "Binder 2.0-Reproducible, Interactive, Sharable Environments for Science at Scale." In *Proceedings of the 17th Python in Science Conference*, 113:120.

Salus, Peter H. 1994. *A Quarter Century of Unix*. ACM Press/Addison-Wesley Publishing Co.

Sandve, Geir, Anton Nekrutenko, James Taylor, and Eivind Hovig. 2013. "Ten Simple Rules for Reproducible Computational Research." *PLoS Computational Biology* 9 (October): e1003285.

Stodden, Victoria, Jennifer Seiler, and Zhaokun Ma. 2018. "An Empirical Analysis of Journal Policy Effectiveness for Computational Reproducibility." *Proceedings of the National Academy of Sciences* 115 (11): 2584–9.

Tatman, R., J. VanderPlas, and S. Dane. 2018. "A Practical Taxonomy of Reproducibility for Machine Learning Research." https://openreview.net/forum?id=B1eYYK5QgX.

Tierney, Nicholas J, and Karthik Ram. 2020. "A Realistic Guide to Making Data Available Alongside Code to Improve Reproducibility." *arXiv Preprint arXiv:2002.11626*.

Wickham, H., and G. Grolemund. 2017. *R for Data Science*. O'Reilly Media.

Wilson, Greg, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, et al. 2014. "Best Practices for Scientific Computing." Edited by Jonathan A. Eisen. *PLoS Biology* 12 (1): e1001745.