Writing reproducible code

Tad Dallas

Contents

What do we mean by 'reproducible'?

Reproducibility is obtaining consistent results using the same input data; computational steps, methods, and code; and conditions of analysis. This definition is synonymous with "computational reproducibility," and the terms are used interchangeably in this report.

Replicability is obtaining consistent results across studies aimed at answering the same scientific question, each of which has obtained its own data.

What are some bad things that can happen when we fail to focus on reproducibility?

Science is evidence-based by definition, and findings build on one another. Being able to reproduce the findings of other researchers is therefore nearly as important as contributing 'new' evidence. Researchers are fallable in their ability to write bug-free code, and by releasing code and data, these errors can be detected and corrected. Apart from this, reproducibiltiy and open-science sort of go hand-in-hand, in that code that is reproducible is also ideally open access, meaning that any researcher has the ability to take the code and data and reproduce the analyses.

What are the major barriers to reproducibility?

- **Changes to libraries that are *breaking changes***. This occurs when the people maintaining the code make changes to add a new feature etc., but do so in a way that breaks the functionality of old code based on their codebase.
- **Different operating systems**. The code and data could rely on few external libraries and still fail to run due to differences in installed backend libraries across different operating systems. For instance, to do spatial data analyses, many libraries outside of R need to be installed in order for the R packages to do spatial data analysis to be installed.
- **Link rot**. Even if the data and code can run, it does not mean that they will be accessible to everyone, as platforms for hosting code may change over time, resulting in some hyperlinks not properly redirecting to the necessary files.
- **Resistance to data and code sharing**. Data ownership and provenance are often listed as concerns among researchers who do not necessarily want to share their data and code. Recent NSF and NIH mandates start to force some researchers hands in terms of data and code sharing, but there will likely always be this feeling of ownership among some fraction of researchers.
- **The code itself is not reproducible**. Last one and perhaps the most obvious. Sharing code doesn't necessarily mean sharing good code, such that the actual code could not reproduce the analyses that the authors claim it does in their manuscript. This is actually not all that uncommon. Some authors code and data files that they share will only have summary data that we must trust is correct, and the code they supply is largely just for visualization.

How can we improve reproducibility in our code?

There are a number of ways we can improve reproducibility in the code we write.

- **Setting seed**. Some code that you may write might have an element of randomness in it. For instance, if I wrote a paper in which I generate data from a stochastic model of population dynamics, the user may not be able to reproduce that exact same trajectory of events. If everyone typed in the following code into an R instance, how likely do you think it is that any two students will have matching entries? Maybe but not likely for `rbinom(10, 1, 0.25)`, and vanishingly unlikely for something like `runif(10,0,1)`. 'Setting the seed' refers to setting the parameter which controls the resulting probabilistic output. So now, if users entered in `set.seed(1234)` before running the code above, everyone should have the same string of 10 numbers.

```
set.seed(1234); rbinom(10, 1, 0.25)
```

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

```
set.seed(1234); rbinom(10, 1, 0.25)
```

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

```
set.seed(1234);rbinom(10, 1, 0.5); rbinom(10,1,0.5); rbinom(10, 1, 0.25)
```

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

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

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

```
set.seed(123)
rbinom(10, 1, 0.25)
```

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

```
rbinom(10, 1, 0.25)
```

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

- **Documenting functions**. Sharing code is great, but often a lot of the burden of reading and interpreting the code is left to the user. By documenting functions – and our code more generally – some of this burden is removed, and can enhance code re-use, help identify potential errors earlier, and is just good practice.

Let's explore an example of this. I wrote a function that performs a "min-max standardization" on a given vector of data. This takes every entry of a vector and subtracts the minimum value, then divides by the maximum minus the minimum. The effect of this is that the data are now scaled between 0 and 1. Proper documentation following the format designed by the `roxygen2` package developers and incorporated into the workhorse library `devtools`, both of which are maintained by Wickham and folks at RStudio/posit. The nice part of this form of documentation is that the documentation and the function are in the same file. Long ago, the documentation of functions in for distribution as an R package would require the developer to edit a separate document with all the details of each function.

```
#' Min-max standardization
#'
#' @param x a vector of numeric data
#'
#' @return the same vector x, but standardized between 0 and 1
#'
#' @examples
#' getMinMax(x=1:10)
```

```
getMinMax <- function(x){
  (x-min(x)) / (max(x)-min(x))
}
```

Here we include information on what the function does, what arguments it takes, what the output will be, and provide a use case. This is most important for package developers, but it is good practice to provide some documentation of your code.

- **Handling errors**. Writing code defensively is a great skill to practice. One of the most important ways to write defensively is to incorporate catches into the code that check for odd inputs and provide informative `warnings` and `errors`.

Let's take the example of the function we wrote above. It's well-documented, but let's say I want to give it a vector that contains an `NA` value? What is going to happen?

```
test <- c(1:10, NA)
getMinMax(test)
```

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

That's not great. Ideally, it would keep the `NA` values where they are, but still standardize the vector. Maybe even provide the user a `warning` about `NA` values being present? Let's implement that now.

```
#' Min-max standardization
#'
#' @param x a vector of numeric data
#'
#' @return the same vector x, but standardized between 0 and 1
#'
#' @examples
#' getMinMax(x=1:10)

getMinMax <- function(x){
  if(any(is.na(x))){
    warning('The vector x contains NA values. These will be ignored.')
  }
  (x-min(x, na.rm=TRUE)) / (max(x, na.rm=TRUE)-min(x, na.rm=TRUE))
}
```

So now, we warn the user about the input data having the `NA` values and have programmed our function in a way to account for those `NA` values. What other ways should we think about modifying this function for clarity and usability?

- **`sessionInfo` at end of document**. Putting `sessionInfo` at the end the document will clearly list the R version, OS, and package versions that you are using. It's a couple steps short of full reproducibility, but it at least shows the user (if you compile the code to html or pdf) exactly what set of conditions allowed the code to run all the way through.

sessionInfo

```
sessionInfo()
```

```
## R version 4.3.1 (2023-06-16)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 22.04.2 LTS
##
## Matrix products: default
```

```
## BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.10.0
## LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.10.0
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## time zone: America/New_York
## tzcode source: system (glibc)
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] ROCR_1.0-11    geodata_0.5-8  terra_1.7-39   maps_3.4.1     rgbif_3.7.7
## [6] gbm_2.1.8.1    rmarkdown_2.23
##
## loaded via a namespace (and not attached):
##  [1] viridis_0.6.3     utf8_1.2.3        generics_0.1.3    xml2_1.3.5
##  [5] stringi_1.7.12    lattice_0.21-8    httpcode_0.3.0    digest_0.6.33
##  [9] magrittr_2.0.3    evaluate_0.21     grid_4.3.1        fastmap_1.1.1
## [13] plyr_1.8.8        jsonlite_1.8.7    Matrix_1.6-0      whisker_0.4.1
## [17] crul_1.4.0        tinytex_0.45      survival_3.5-5    urltools_1.7.3
## [21] gridExtra_2.3     httr_1.4.6        fansi_1.0.4       viridisLite_0.4.2
## [25] scales_1.2.1      oai_0.4.0         codetools_0.2-19  lazyeval_0.2.2
## [29] cli_3.6.1         rlang_1.1.1       triebeard_0.4.1   munsell_0.5.0
## [33] splines_4.3.1     yaml_2.3.7        parallel_4.3.1    tools_4.3.1
## [37] dplyr_1.1.2       colorspace_2.1-0  ggplot2_3.4.2     curl_5.0.1
## [41] vctrs_0.6.3       R6_2.5.1          lifecycle_1.0.3   stringr_1.5.0
## [45] pkgconfig_2.0.3   pillar_1.9.0      gtable_0.3.3      data.table_1.14.8
## [49] glue_1.6.2        Rcpp_1.0.11       xfun_0.39         tibble_3.2.1
## [53] tidyselect_1.2.0  highr_0.10        knitr_1.43        htmltools_0.5.5
## [57] compiler_4.3.1
```