Rclean: A Tool for Writing Cleaner, More Transparent Code

Introduction

The growth of programming in the sciences has been explosive in the last decade. This has facilitated the rapid advancement of science through the agile development of computational tools. However, concerns have begun to surface about the reproducibility of scientific research in general (R. D. Peng et al. 2011 Baker (2016)) and the potential issues stemming from issues with analytical software (Pasquier et al. 2017 Stodden, Seiler, and Ma (2018)). Specifically, there is a growing recognition across disciplines that simply making data and software "available" is not enough and that there is a need to improve the transparency and stability of scientific software (Pasquier et al. 2018).

At the core of the growth of scientific computation, the `R` statistical programming language has grown exponentially to become one of the top ten programming languages in use today. At it's root R is a *statistical* programming language. That is, it was designed for use in analytical workflows; and the majority of the R community is focused on producing code for idiosyncratic projects that are *results* oriented. Also, R's design is intentionally at a level that abstracts many aspects of programming that would otherwise act as a barrier to entry for many users. This is good in that there are many people who use R to their benefit with little to no formal training in computer science or software engineering, but these same users can also be frequently frustrated by code that is fragile, buggy and complicated enough to quickly become obtuse even to the authors. The stability, reproducibility and re-use of scientific analyses in R would be improved by refactoring, which is a common practice in software engineering (**???**). From this perspective, tools that can lower the time and energy required to refactor analytical scripts and otherwise help to "clean" code, but abstracted enough to be easily accessible, could have a significant impact on scientific reproducibility across all disciplines (Visser et al. 2015).

To provide support for easier refactoring in R, we have created `Rclean`. The `Rclean` package provides tools to automatically reduce a script to the parts that are specifically relevant to a research product (e.g. a scientific report, academic talk, research article, etc.). Although potentially useful to all R coders, it was designed to ease refactoring for scientists that use R but do not have formal training in software engineering. Here, we detail the structure of the package's API, describe the general workflow illustrated by an example use case and provide some background on how data provenance enables the underlying functionality of the package. We then end with a discussion of future applications of data provenance in the context of "code cleaning" and the potential integration with other software engineering tools for the R community.

Methods

More often then not, when someone is writing an R script, the intent to produce a set of results, such as a statistical analysis, figure, table, etc. This set of results is always a subset of a much larger set of possible ways to explore a dataset, as there are many statistical approaches and tests, let alone ways to create visualizations and other representations of patterns in data. This commonly leads to lengthy, complicated scripts from which researchers manually subset results, but never refactor, i.e. refine code so that it is shorter and focused on a desired product.

The goal of `Rclean` is to provide a set of tools that help someone reduce and organize code based on results. The package uses an automated technique based on data provenance (details below) to analyze existing scripts and provide ways to identify and extract code to produce a desired output. To keep the process simple

and straight-forward much of this process has been abstracted for the user, and the API has been kept to a minimum set of functions that enable a user to conduct the following basic workflow:

1. Obtain the "cleaned" code for a result(s).
2. Transfer the code to a new context (e.g. a new script, function, reproducible example, web-app, etc.).
3. Get information about the possible results and repeat as needed.

The API

The package's main functions are `clean` and `keep`. When provided a file path to a script and the name of a result (or a set of results), `clean` analyzes the script's code and extracts the lines of code required to produce the results. By default, code is formatted following general best practices recommended by the Tidyverse via the `styler` package. This code can then be passed to the `keep` function, which can either write the code to disk or copy the code to the user's clipboard (if no output file path is supplied) and the user can paste the code into another location (e.g. a script editor).

In the process of cleaning a script, it is likely that a user will not know the precise objects they want and might require some help analyzing it. There are several functions to help with this process. The `get_vars` function will return a list of possible results for a given script at a supplied file path. This is obviously an important step, and justifiably, the default behavior of the `clean` function is to run `get_vars` if no results are supplied. To help with limiting and checking the selection of results, the `code_graph` function creates a network graph of the relationships among the various results and lines of code in the script, which is the function used to create the figure demonstrating data provenance @ref(fig:prov-graph). Last, the `get_libs` function can be used to detect the packages that a given script depends on, which it will return as coded library calls that can be inserted into a cleaned script.

Data Provenance

All of these processes rely on the generation of data provenance. The term provenance means information about the origins of some object. Data provenance is a formal representation of the execution of a computational process (https://www.w3.org/TR/prov-dm/), to rigorously determine the the unique computational pathway from inputs to results (Carata et al. 2014). To avoid confusion, note that "data" in this context is used in a broad sense to include all of the information generated during computation, not just the data that are collected in a research project that are used as input to an analysis. Having the formalized, mathematically rigorous representation that data provenance provides guarantees that the analyses that `Rclean` conducts are theoretically sound. Most importantly, because the relationships defined by the provenance can be represented as a graph, it is possible to apply network search algorithms to determine the minimum and sufficient code needed to generate the chosen result in the `clean` function.

There are multiple approaches to collecting data provenance, but `Rclean` uses "prospective" provenance, which analyzes code and uses language specific information to predict the relationship among processes and data objects. `Rclean` relies on a library called `CodeDepends` to gather the prospective provenance for each script. For more information on the mechanics of the `CodeDepends` package, see (**???**). To get an idea of what data provenance is, take a look at the `code_graph` function. The plot that it generates is a graphical representation of the prospective provenance generated for `Rclean` @ref(fig:prov-graph).

"'{R prov-graph, fig.cap = "Network diagram of the prospective data provenance generated for an example script. Arrows indicate which lines of code (numbered) produced which objects (named).", echo = FALSE}

script <- system.file("example", "simple_script.R", package = "Rclean") code_graph(script)

```
Relatedly, it is important to point out that `Rclean` *does not* keep
comments present in code. This could be seen as a limitation of the
data provenance, which currently does not assign them a
```

relationships. Therefore, although there is often very useful or even invaluable information in comments, the `clean` function removes them. This is a general issue with automated methods for detecting the relationships between comments and the code itself. Comments at the end of lines are typically relevant to the line they are on, but this is not a requirement and could refer to other lines. Also, comments occupying their own lines usually refer to the following lines, but this is also not necessarily the case. As `clean` depends on the unambiguous determination of relationships in the production of results, it cannot operate automatically on comments. However, comments in the original code remain untouched and can be used to inform the reduced code. Also, as the `clean` function is oriented toward isolating code based on a specific result, the resulting code tends to naturally support the generation of new comments that are higher level (e.g. "The following produces a plot of the mean response of each treatment group."), and lower level comments are not necessary because the code is simpler and clearer.

In the future, it would also be useful to extend the existing framework to support other provenance methods. One such possibility is *retrospective provenance*, which tracks a computational process as it is executing. Through this active, concurrent monitoring, retrospective provenance can gather information that static prospective provenance can't. Greater details of the computational process would enable other features that could address some challenges, such as processing information from comments, parsing control statements and replicating random processes. However, using retrospective provenance comes at a cost. In order to gather it, the script needs to be executed. When scripts are computationally intensive or contain bugs that stop execution, then retrospective provenance can not be obtained for part or all of the code. Some work has already been done in the direction of implementing retrospective provenance for code cleaning in R (see http://end-to-end-provenance.github.io).

Results

Example

Data analysis can be messy and complicated, so it's no wonder that the code often reflects this. "What did I measure? What analyses are relevant to them? Do I need to transform the data? What's the function for the analysis I want to run?" This is why having a way to isolate code based on variables can be valuable. The following is an example of a script that has some complications. As you can see, although the script is not extremely long or complicated, it's difficult enough to make it frustrating to visualize it in its entirety and pick through it.

[1] “library(stats)”

[2] “x <- 1:100”

[3] “x <- log(x)”

[4] “x <- x * 2”

[5] “x <- lapply(x, rep, times = 4)”

[6] “### This is a note that I made for myself.”

[7] “### Next time, make sure to use a different analysis.”

[8] “### Also, check with someone about how to run some other analysis.”

[9] “x <- do.call(cbind, x)”

[10] “”

[11] “### Now I’m going to create a different variable.”

[12] “### This is the best variable the world has ever seen.”

[13] “”

[14] “x2 <- sample(10:1000, 100)”

[15] “x2 <- lapply(x2, rnorm)”

[16] “”

[17] “### Wait, now I had another thought about x that I want to work through.”

[18] “”

[19] “x <- x * 2”

[20] “colnames(x) <- paste0("X", seq_len(ncol(x)))”

[21] “rownames(x) <- LETTERS[seq_len(nrow(x))]”

[22] “x <- t(x)”

[23] “x[, "A"] <- sqrt(x[, "A"])”

[24] “”

[25] “for (i in seq_along(colnames(x))) {”

[26] " set.seed(17)"

[27] " x[, i] <- x[, i] + runif(length(x[, i]), -1, 1)"

So, let's say we've come to our script wanting to extract the code to produce one of the results `fit.sqrt.A`, which is an analysis that is relevant to some product. Not only do we want to double check the results, we also want to use the code again for another purpose, such as creating a plot of the patterns supported by the test. Manually tracing through our code for all the variables used in the test and finding all of the lines that were used to prepare them for the analysis would be annoying and difficult, especially given the fact that we have used "x" as a prefix for multiple unrelated objects in the script. Instead, we can easily do this automatically with `Rclean`.

```
clean(script, "fit_sqrt_A")
```

```
## x <- 1:100
## x <- log(x)
## x <- x * 2
## x <- lapply(x, rep, times = 4)
## x <- do.call(cbind, x)
## x <- x * 2
## colnames(x) <- paste0("X", seq_len(ncol(x)))
## rownames(x) <- LETTERS[seq_len(nrow(x))]
## x <- t(x)
## x[, "A"] <- sqrt(x[, "A"])
## for (i in seq_along(colnames(x))) {
##   set.seed(17)
##   x[, i] <- x[, i] + runif(length(x[, i]), -1, 1)
## }
## x[, 1] <- x[, 1] * 2 + 10
## x[, 2] <- x[, 1] + x[, 2]
## x[, "A"] <- x[, "A"] * 2
## x <- data.frame(x)
## fit_sqrt_A <- lm(I(sqrt(A)) ~ B, data = x)
```

As you can see, `Rclean` has picked through the tangled bits of code and found the minimal set of lines relevant to our object of interest. This code can now be visually inspected to adapt the original code or ported to a new, "refactored" script using `keep`.

```
fitSA <- clean(script, "fit_sqrt_A")
keep(fitSA)
```

This will pass the code to the clipboard for pasting into another document. To write directly to a new file, a file path can be specified.

```
fitSA <- clean(script, "fit_sqrt_A")
keep(fitSA, file = "fit_SA.R")
```

To explore more possible variables to extract, the `get_vars` function can be used to produce a list of the variables (aka. objects) that are created in the script.

```
get_vars(script)
```

```
## [1] "x"          "x2"         "i"          "x3"         "fit.23"
## [6] "fit.xx"     "fit_sqrt_A" "z"          "fit_anova"
```

Especially when the code for different variables are entangled, it can be useful to visual the code in order to devise an approach to cleaning. As seen above @ref(fig:prov-graph), `code_graph` will produce a visual of the code and the objects that they produce.

"'{R ex-code_graph, fig.cap = "Example of the plot produced by the code_graph function showing which lines of code produce which variables and which variables are used by other lines of code."}

code_graph(script)

```

After examining the output from `get_vars` and `code_graph`, it is possible that more than one object needs to be isolated. To do this is simple in `keep` by passing the set of desired objects to the *vars* argument.

```
clean(script, vars = c("fit_sqrt_A", "fit_anova"))
```

```
## x <- 1:100
## x <- log(x)
## x <- x * 2
## x <- lapply(x, rep, times = 4)
## x <- do.call(cbind, x)
## x2 <- sample(10:1000, 100)
## x2 <- lapply(x2, rnorm)
## x <- x * 2
## colnames(x) <- paste0("X", seq_len(ncol(x)))
## rownames(x) <- LETTERS[seq_len(nrow(x))]
## x <- t(x)
## x[, "A"] <- sqrt(x[, "A"])
## for (i in seq_along(colnames(x))) {
##   set.seed(17)
##   x[, i] <- x[, i] + runif(length(x[, i]), -1, 1)
## }
## x2 <- lapply(x2, function(x) x[1:10])
## x2 <- do.call(rbind, x2)
## x[, 1] <- x[, 1] * 2 + 10
## x[, 2] <- x[, 1] + x[, 2]
## x[, "A"] <- x[, "A"] * 2
## x <- data.frame(x)
## fit_sqrt_A <- lm(I(sqrt(A)) ~ B, data = x)
## z <- c(rep("A", nrow(x2) / 2), rep("B", nrow(x2) / 2))
## fit_anova <- aov(x2 ~ z, data = data.frame(x2 = x2[, 1], z))
```

Currently, because of the data provenance that is used, libraries can not be isolated directly during the cleaning process. So, the `get_libs` function provides a way to detect the libraries for a given script. Just supply a file path and `get_libs` will return the libraries that are called by that script.

```
get_libs(script)
```

```
## [1] "stats"
```

Software Availability

The software is currently hosted on Github, and we recommend using the `devtools` library to install directly from the repository (https://github.com/ROpenSci/Rclean). The package is open-source and welcomes contributions. Please visit the repository page to report issues, request features or provide other feedback.

Discussion

We have created `Rclean` to provide a simple, easy to use tool for scientists who would like help refactoring code. Using `Rclean` the code necessary to produce a specified result (e.g., an object stored in memory or a table or figure written to disk) can be easily and *reliably* isolated even when tangled with code for other results. This functionality is enabled by graph analysis of data provenance collected from the user's script. This is likely to be a broadly useful tool as statistical programming becomes more common across the sciences.
```

Tools, such as this, that make it easier to produce transparent, accessible code will be an important aid for improving scientific reproducibility.

Although the workflow described is impactful, we see promise in extending it to interface with other clean code and reproducibility tools. One example is the `reprex` package, which provides a simple API for sharing reproducible examples. Another possibility is to help transition scripts to function, package and workflow creation and refactoring via toolboxes like `drake`. `Rclean` could provide a reliable way to extract parts of a larger script that would be piped to a simplified reproducible example, in the case of `reprex`, or, since it can isolate the code from inputs to one or more outputs, be used to extract all of the components needed to write one or more functions that would be a part of a package or workflow, as is the goal of `drake`.

To conclude, we hope that `Rclean` makes writing scientific software easier for the R community. We look forward to feedback and help with extending its application, particularly in the area of reproducibility, such as using code cleaning in the creation of more robust capsules (Pasquier et al. 2018). To get involved, report bugs, suggest features, please visit the project page.

Acknowledgments

This work was improved by discussions with ecologists at Harvard Forest and through the helpful review provided by the ROpenSci community, particularly Anna Krystalli, Will Landau and Clemens Schmid. Much of the work was funded by US National Science Foundation grant SSI-1450277 for applications of End-to-End Data Provenance.

References

Baker, Monya. 2016. "1,500 Scientists Lift the Lid on Reproducibility." *Nature* 533: 452–54. doi:10.1038/533452a.

Carata, Lucian, Sherif Akoush, Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, Margo Seltzer, and Andy Hopper. 2014. "A Primer on Provenance." *Queue* 12 (3). ACM: 10–23. doi:10.1145/2602649.2602651.

Pasquier, Thomas, Matthew K. Lau, Xueyuan Han, Elizabeth Fong, Barbara S. Lerner, Emery R. Boose, Merce Crosas, Aaron M. Ellison, and Margo Seltzer. 2018. "Sharing and Preserving Computational Analyses for Posterity with encapsulator." *Comput. Sci. Eng.* 20 (4): 111–24. doi:10.1109/MCSE.2018.042781334.

Pasquier, Thomas, Matthew K. Lau, Ana Trisovic, Emery R. Boose, Ben Couturier, Mercè Crosas, Aaron M. Ellison, Valerie Gibson, Chris R. Jones, and Margo Seltzer. 2017. "If these data could talk." *Sci. Data* 4 (September): 170114. doi:10.1038/sdata.2017.114.

Peng, Roger D, B. Hanson, A. Sugden, B. Alberts, R. D. Peng, F. Dominici, S. L. Zeger, et al. 2011. "Reproducible research in computational science." *Science* 334 (6060). American Association for the Advancement of Science: 1226–7. doi:10.1126/science.1213847.

Stodden, Victoria, Jennifer Seiler, and Zhaokun Ma. 2018. "An empirical analysis of journal policy effectiveness for computational reproducibility." *Proc. Natl. Acad. Sci. U. S. A.* 115 (11). National Academy of Sciences: 2584–9. doi:10.1073/pnas.1708290115.

Visser, Marco D., Sean M. McMahon, Cory Merow, Philip M. Dixon, Sydne Record, and Eelke Jongejans. 2015. "Speeding Up Ecological and Evolutionary Computations in R; Essentials of High Performance Computing for Biologists." Edited by Francis Ouellette. *PLOS Comput. Biol.* 11 (3). Springer-Verlag: e1004140. doi:10.1371/journal.pcbi.1004140.