Rclean: A Tool for Writing Cleaner, More Transparent Code

6 December 2019

Introduction

The growth of programming in the sciences has been explosive in the last decade. In particular, the statistical programming language `R` has grown exponentially to become one of the top ten programming languages in use today. Increasingly, concerns have arisen over the reproducibility of scientific research (R. D. Peng et al. 2011 Baker (2016) Stodden, Seiler, and Ma (2018)) and the potential issues stemming from the complexity and fragility of analytical software (Pasquier et al. 2017). There is now a recognition that simply making the code open is not enough, and that there is a need for improvements to documentation and transparency (Pasquier et al. 2018).

At it's root R is a statistical programming language, that is, it was designed for use in analytical workflows. As such, 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. However, these same users are also frequently frustrated by code that is fragile, buggy and complicated enough to quickly become obtuse (even to themselves) in a very short amount of time, which leads to frequently re-writing code unecessarily. Also, when scripts take an extremely long time to execute, reducing unnecessary analyses can increase computational efficiency.

The stability, reproducibility and re-use of scientific analyses in R would be improved by refactoring. From this perspective, tools that can lower the time and energy required to re-factor and streamline analytical scripts, and otherwise help to "clean" code will have a significant impact on scientific reproducibility across all disciplines (Visser et al. 2015). 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, 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. reduce to the desired subset). In part, this is driven a desire to record the entire scripting/analytical process and not lose work, which is alleviated by the use of proper version control systems (e.g. git and subversion); however, refactoring will always be a part of the analytical process regardless.

To support 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

The main goal of *Rclean* is to provide a means to simplify code. To keep the process simple and straightforward, the API has been kept to a minimum set of functions, which enable a user to conduct the basic workflow of getting information about the posible results in a script, obtaining the minimum code for a set of results and then creating a new "cleaned" script or other software (e.g. a function, reproducible example, web-app, etc.).

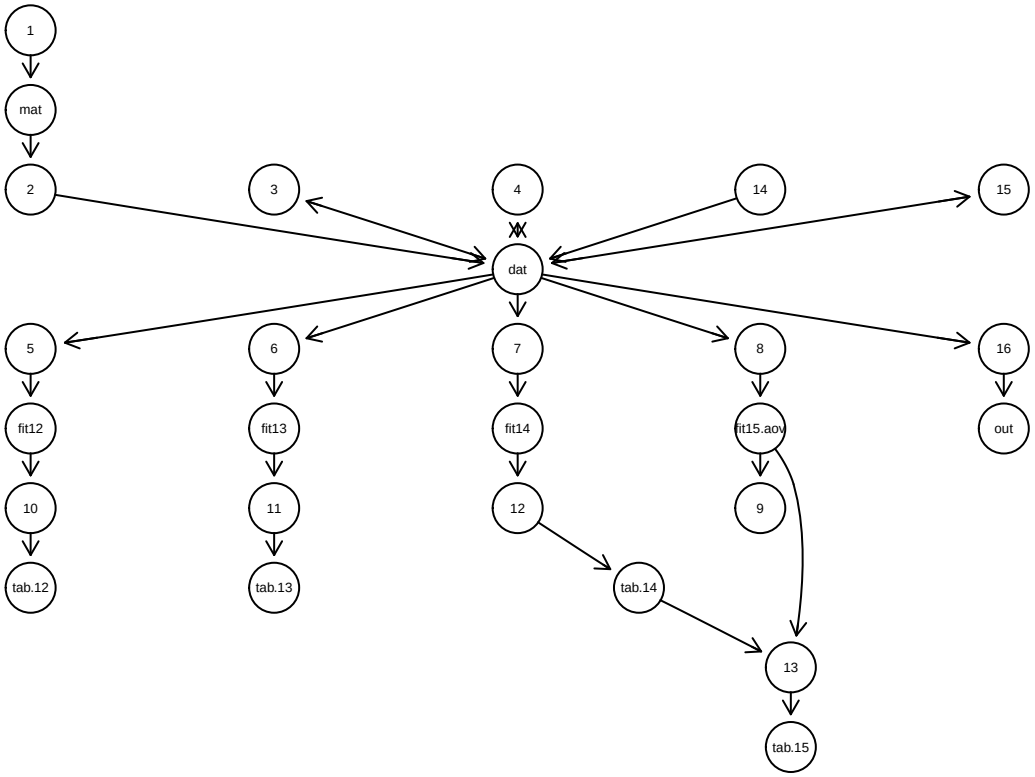
The package's main functions are `clean` and `keep`. When provided a file path to a script and the name of a result (or a set of results), `clean` analyzes the script's code and extracts the lines of code required to produce the results. This code can then be passed to the `keep` function, which can either write the code to disk or copy the code to the user's clipboard (if no ouptput 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 might require some help analyzing the script. There are several functions to help. 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 justsifiably, 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. 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.

All of these processes rely on the generation of data provenance (Carata et al. 2014), which 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. 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 gaurantees that the analyses that *RClean* conducts are theoretically sound. Most importantly, it makes it possible to apply a network search algorithm 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*.

```
code_graph(script)
```



Results

Example

Conducting analyses is challenging in that it requires thinking about multiple concepts at the same time. What did I measure? What analyses are relevant to them? Do I need to transform the data? How do I go about managing the data given how they were entered? What's the code for the analysis I want to run? And so on. Data analysis can be messy and complicated, so it's no wonder that code reflects this. And this is a reason 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, it's long 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)"
## [28] "}"
## [29] ""
## [30] "### Ok. Now I can get back to x2."
## [31] "### Now I just need to check out a bunch of stuff with it."
## [32] ""
## [33] "lapply(x2, length)[1]"
## [34] "max(unlist(lapply(x2, length)))"
## [35] "range(unlist(lapply(x2, length)))"
## [36] "head(x2[[1]])"
## [37] "tail(x2[[1]])"
## [38] ""
## [39] "## Now, based on that stuff, I need to subset x2."
## [40] ""
## [41] "x2 <- lapply(x2, function(x) x[1:10])"
## [42] ""
## [43] "## And turn it into a matrix."
## [44] "x2 <- do.call(rbind, x2)"
## [45] ""
## [46] "## Now, based on x2, I need to create x3."
## [47] "x3 <- x2[, 1:2]"
## [48] "x3 <- apply(x3, 2, round, digits = 3)"
## [49] ""
## [50] "## Oh wait! Another thought about x."
## [51] ""
## [52] "x[, 1] <- x[, 1] * 2 + 10"
## [53] "x[, 2] <- x[, 1] + x[, 2]"
## [54] "x[, \"A\"] <- x[, \"A\"] * 2"
## [55] ""
## [56] "## Now, I want to run an analysis on two variables in x2 and x3."
## [57] ""
## [58] "fit.23 <- lm(x2 ~ x3, data = data.frame(x2[, 1], x3[, 1]))"
## [59] "summary(fit.23)"
## [60] ""
## [61] "## And while I'm at it, I should do an analysis on x."
## [62] ""
## [63] "x <- data.frame(x)"
## [64] "fit.xx <- lm(A~B, data = x)"
## [65] "summary(fit.xx)"
## [66] "shapiro.test(residuals(fit.xx))"
## [67] ""
## [68] "## Ah, it looks like I should probably transform A."
## [69] "## Let's try that."
```

```
## [70] "fit_sqrt_A <- lm(I(sqrt(A))~B, data = x)"
## [71] "summary(fit_sqrt_A)"
## [72] "shapiro.test(residuals(fit_sqrt_A))"
## [73] ""
## [74] "## Looks good!"
## [75] ""
## [76] "## After that. I came back and ran another analysis with "
## [77] "## x2 and a new variable."
## [78] ""
## [79] "z <- c(rep(\"A\", nrow(x2) / 2), rep(\"B\", nrow(x2) / 2))"
## [80] "fit_anova <- aov(x2 ~ z, data = data.frame(x2 = x2[, 1], z))"
## [81] "summary(fit_anova)"
```

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.long, "fit_sqrt_A")
```

```
## Warning: Could not use colored = TRUE, as the package prettycode is not
## installed. Please install it if you want to see colored output or see `?
## print.vertical` for more information.
```

```
## 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.

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

The `Rclean` package provides a simple, easy to use tool for scientists conducting analyses in the R programming language. Using graph analytic algorithms, `Rclean` isolates the code necessary to produce a specified result (e.g., an object stored in memory or a table or figure written to disk). As statistical programming becomes more common across the sciences, tools that make the production of accessible code will be an important aid for improving scientific reproducibility. `Rclean` has been designed to take advantage of recent advances in data provenance capture techniques to create a minimal tool for this purpose.

It is worth mentioning and discussing that `Rclean` does not keep comments present in code. Although, there is often very useful or even invaluable information in comments, the `clean` function removes comments. This is primarily due to the lack of a mathematically formal method for determining their relationship to the code itself. Comments at the end of lines are typically relevant to the line they are on, but this is not explicitly required. 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.

The existing framework could be extended to support new provenance capture methods. One 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. Only using prospective provenance means that the outcomes of some processes can not be predicted. For example, if there is a part of a script that is determined by a random number, the current implementation of prospective provenance can not predict the path that will be taken through the code. Therefore, the code cannot be reduced to exclude the pathway that would not be taken. 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).

To conclude, we hope that `Rclean` makes writing scientifi software easier. We look forward to feedback and help with extending its applications, 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.