6-29-2009

Caching and Visualizing Statistical Analyses

Roger D. Peng
Johns Hopkins University, rpeng@jhsph.edu

Duncan Temple Lang
UC Davis

Title: Caching and Visualizing Statistical Analyses

Authors: Roger D. Peng, Duncan Temple Lang

Contact Information

Roger D. Peng
Department of Biostatistics
Johns Hopkins Bloomberg School of Public Health
615 North Wolfe Street
Baltimore MD 21205
USA
E-mail: rpeng@jhsph.edu
URL: http://www.biostat.jhsph.edu/~rpeng/

Duncan Temple Lang
Department of Statistics
University of California, Davis
4210 Mathematical Sciences Building
One Shield Avenue
Davis CA 95616
USA
E-mail: duncan@wald.ucdavis.edu
URL: http://www.stat.ucdavis.edu/~duncan/

Abstract

We present the cacher and CodeDepends packages for R, which provide tools for (1) caching and analyzing the code for statistical analyses and (2) distributing these analyses to others in an efficient manner over the web. The cacher package takes objects created by evaluating R expressions and stores them in key-value databases. These databases of cached objects can subsequently be assembled into "cache packages" for distribution over the web. The cacher package also provides tools to help readers examine the data and code in a statistical analysis and reproduce, modify, or improve upon the results. In addition, readers can easily conduct alternate analyses of the data. The CodeDepends package provides complementary tools for analyzing and visualizing the code for a statistical analysis and this functionality has been integrated into the cacher package. In this chapter we describe the cacher and CodeDepends packages and provide examples of how they can be used for reproducible research.

Introduction

The replication of scientific findings using independent investigators, methods, data, equipment and protocols is the standard by which scientific claims are evaluated. However, in many fields of study there are examples of scientific investigations that cannot be fully replicated. Common reasons for a lack of replicability include a lack of time or resources. When scientific studies cannot be replicated, there is a need for a minimum standard that can fill the void between full replication and nothing. One candidate for this minimum standard is reproducibility, which requires that datasets and computer code implementing analyses be made available to others for verifying published results and conducting alternative analyses.

The need for publishing reproducible results is increasing for a number of reasons. Investigators are more frequently examining weak associations and complex interactions for which the data contain relatively little information. New technologies allow scientists in all areas to compile complex high-dimensional databases and the ubiquity of powerful statistical and computing capabilities allows investigators to explore those databases and identify associations of potential interest. However, with the increase in data and computing power comes a greater potential for identifying spurious associations. In addition to these developments, recent reports of fraudulent research being published in the biomedical literature have highlighted the need for reproducibility in biomedical studies and have invited the attention of the major medical journals (e.g. Laine et al. 2007). Even without the presence of deliberate fraud, it should be noted that as analyses become more complicated, the possibility of inadvertant errors resulting in misleading findings looms large. In the example of Baggerly et al. (2005), the errors discovered were not necessarily simple or obvious and the examination of the problem itself required a sophisticated analysis. Misunderstandings about commonly used software can also lead to problems, particularly when such software is applied to situations not originally imagined.

While many might agree with the benefits of disseminating reproducible results, there is unfortunately a general lack of infrastructure for supporting such endeavors. Investigators who are willing to make their results reproducible are confronted with a number of barriers, one of which is the need to distribute, and make available for an indefinite amount of time, the supplementary materials required for reproducing the results. Readers who are interested in reproducing the results of others often need to expend substantial effort to gather the materials and study the statistical analysis code. There is currently a need and opportunity for the development of software to more efficiently connect authors and readers so that results can be reproduced and science can be advanced.

The distribution of reproducible results is a problem for which the solution varies depending on the complexity of the research. Small investigations involving moderately sized datasets and standard computational techniques can be archived and distributed in their entirety. Readers can subsequently study and rerun the entire analysis from start to finish to see if they can obtain the same results as the authors. Complex investigations involving large or multiple linked datasets and sophisticated statistical computations will be more difficult for readers to reproduce because of the resources and time required for running the analysis. In such a situation a method is needed to

give readers without equivalent resources the ability to conduct an initial examination of the details of the investigation and to partially reproduce or verify the results. In addition, complex statistical analyses will typically involve complex statistical code whose details may be difficult to understand upon first glance. Software that can help to visualize the analysis code itelf can be useful to readers for understanding the flow of the analysis and for identifying potential points of interest.

A framework in which reproducible results can be distributed using cached computations is described in Peng and Eckel (2008). Cached computations are results that are stored in a database as an analysis is being conducted. These stored results can be distributed via web sites or central repositories so that others may explore the datasets and computer code for a given scientific investigation.

We have developed some tools for assisting authors and researchers in conducting reproducible research. In this chapter we describe the cacher and CodeDepends packages for the R system. For authors, these packages provide tools for caching statistical analyses and for distributing these analyses to others in an efficient manner. For readers, these packages provide tools for visualizing the code and the results of a statistical analysis.

Description of Software

The cacher package is available from the Comprehensive R Archive Network (http://cran.r-project.org/). The CodeDepends package is available from the Omegahat project (http://www.omegahat.org/CodeDepends/). The Rgraphviz package (also available from CRAN) is required to use some of the functionality of the CodeDepends package described here. The cacher package is written primarily in R with a few components written in C. The CodeDepends package is written entirely in R. Both packages are licensed on the GNU GPL version 2 or higher.

The cacher package provides interfaces for two types of users. The first type consists of authors of statistical analyses who wish to cache their analyses in a database and distribute the cached analysis to others. The second type of user consists of readers who wish to obtain cached analyses over the web and explore the data and code in those analyses in an efficient manner. In this chapter we give a brief overview of the capabilities of both packages. Complete information about the design of the cacher package can be found in Peng (2008).

The primary function in the cacher package for authors of statistical analyses is the `cacher` function, which takes the name of an R source file as its first argument. This should be a standard source file containing R code to be evaluated and cached. The remaining two arguments to `cacher` specify the location of the cache directory (the default is `.cache`). Optionally, the log file can be specified where messages on the progress of `cacher` will be printed. The simplest invocation of `cacher` is

```
library(cacher)
cacher("myanalysis.R")
```

where “myanalysis.R” is the name of an R source file. The basic procedure of `cacher` is to read each R expression in the source file, evaluate it, cache the results to a key-value database, and then move to the next expression, until the end of the file is reached. More specifically, `cacher` will

1. Parse the R source file;
2. Create the necessary cache directories and subdirectories;
3. Set various configuration variables and hook functions for plotting;
4. Copy the source file to the cache directory;
5. Cycle through each expression in the source file:
    a. If an expression has never been evaluated, evaluate it and store any resulting R objects in the cache database,
    b. If a cached result exists, lazy-load the results from the cache database and move to the next expression,
    c. If an expression does not create any R objects (i.e., there is nothing to cache), add the expression to the list of expressions where evaluation needs to be forced,
    d. Write out metadata for this expression to the metadata file.

If a source file needs to be executed multiple times (e.g. because of revisions), cached results from previous runs can be used in place of actual evaluation in order to minimize the total time for evaluation. However, if the code changes then some parts will need to be re-evaluated. In order to assess changes to the R code, the cacher function creates a unique identifier for each expression in an R code file by taking the SHA-1 digest of the expression, the expression history (i.e. expressions preceding the current expression), and the name of the source file. For the first expression, the expression history is of length zero. Therefore, if the code in the source file changes, the digest of the expression wil also change. Using the expression history to identify individual R expressions is a way to prevent expressions such as

```
x <- 1
y <- x^2
x <- 2
y <- x^2
```

from being inappropriately loaded from the cache. In this case, the expression `y <- x^2` appears twice, but the value of `x` changes in between. An expression such as this one may appear multiple times in a source file and we do not want to load the same value for `y` every time since the value of `x` may be changing. Using the expression history can uniquely identify each occurrence of a duplicate expression. The CodeDepends package has tools that can also keep track of the sequence of expressions and determine that the value of `x` has changed since `y` was last defined.

Distributing Cached Analyses

Authors who wish to distribute a cached statistical analysis over the web and also have access to a local webserver, can post the cache directory on the webserver so that others can download the materials using the `clonecache` function. All that is required is for the user to copy the directory to a location on the webserver that is visible to outside

users. We have placed a number of example analyses on the web site of the Reproducible Research Archive (http://penguin.biostat.jhsph.edu), which is currently under development. Alternatively, an author can use the `cachepackage()` function which creates a zip file of the entire cache. This zip file could then be distributed to others (e.g. via email or the web) who can subsequently unzip the file and explore the contents of the cache using the functions described below.

The primary function for downloading a cached analysis is the `clonecache` function. The user can pass to `clonecache` the URL of the directory containing a cached analysis. Given a URL, `clonecache` creates a cache directory on the user's local machine and downloads the source files and metadata from the remote machine. The `clonecache` function also takes an ID string that can be used to retrieve analyses stored on the Archive web site. By default, `clonecache` does not download any of the database files since these could be very large and the user may not be interested in every R object in the analysis. Rather, these database files are downloaded as needed when the users explores a cached analysis. In order to force the downloading of all database objects when initially cloning, the user needs to set the option `all.files = TRUE` when calling `clonecache`. Once an analysis is cloned the functions described in the following section can be used to explore the code and data objects in the analysis.

Exploring and Visualizing Cached Analyses

The cacher package provides some basic tools to allow users to interact with the code and data provided in a cached analysis. The primary functions making up the user interface for readers wishing to explore a cached analysis written by someone else are

- `showfiles`: Show what source files are available in the cache to be examined by the user. If the author of the package cached analyses from multiple source files, then this function can be used to determine which analysis should be examined. One can switch between different source files by calling the `sourcefile` function.
- `sourcefile`: Get or set the current source file for analysis.
- `code`: Show the expressions for a given source file. By default, code shows all expressions in a file in a one-line abbreviated form along with their expression sequence numbers. To see each expression in its entirety, the argument `full = TRUE` must be set.
- `showcode`: Show the original source file in the pager, which can be useful if one is interested in seeing any comments.
- `loadcache`: Lazy-load cached computation databases into an environment. This function takes a numeric vector of expression sequence numbers and loads objects associated with those expressions in the order that the expressions are specified. Once a cache database is lazy-loaded, the object names appear in the environment into which the database was loaded, but they do not occupy any memory until they are first accessed. If `loadcache` is used to load objects from a remote cache, then the corresponding database files will be downloaded on the object's first access.
- `runcode`: This function takes as input a numeric vector of expression sequence numbers executes the code in those expressions. Each expression is evaluated in

the order in which it appears in the input vector. By default, if a cached computation database is associated with an expression, then the database is lazy-loaded via loadcache rather than executed. In order to force evaluation of code in an expression, one needs to set `forceAll = TRUE` when calling runcode. If an error occurs when executing the code in an expression, a message is printed to the console indicating the error and the expression is skipped. While the runcode function can be used to evaluate individual expressions, the results of such evaluation may not be correct if the dependent expressions have not previously been evaluated. At this point in development of the cacher package, reproducible results for a specific expression in an analysis can only be obtained by evaluating all of the expressions in order up to that expression. The CodeDepends package has functions for tracking the dependencies of R objects in an analysis and we will be working to integrate that functionality into the cacher package in a future release.

- `graphcode`: This function reads the source code for the cached analysis and creates a directed graph showing the relationships between the R objects created in the analysis and how they are used in defining each other. The `graphcode` function uses the capabilities implemented in the CodeDepends package to statically examine code and compute the various dependencies. For the creation of the graph itself, the Rgraphviz package is required.
- `objectcode`: This function takes the name of an R object and returns the sequence of R expressions that leads to the creation of that object. It returns the indices of the sequence of R expressions which could subsequently be passed to a function like `runcode`. This function can be useful for identifying the code for reconstructing an R object without having to run an entire analysis, which may contain many unrelated parts.

Example

As an example of how the cacher and CodeDepends packages can be used we present a brief statistical analysis of particulate matter (PM) air pollution and mortality data. The data that we use come from the National Morbidity, Mortality, and Air Pollution Study (NMMAPS) and details about the data can be obtained from the Internet-based Health and Air Pollution Surveillance System web site (http://www.ihapss.jhsph.edu/). Information about the original study is presented in Samet et al. 2000. The analysis presented here estimates the short-term association between daily PM levels and daily mortality. Briefly, a Poisson generalized linear time series model is fit to daily mortality count and PM data from the largest 20 cities in the United States, adjusting for other factors like temperature, humidity, and seasonal trends. The primary target of inference is the log relative risk associated with short-term changes in ambient PM levels for each of the 20 cities. Further details about the methodology used in the analysis can be found in Peng and Dominici (2008).

We start by downloading the analysis from the Reproducible Research Archive web site using the `clonecache` function. Each analysis on the Archive web site is assigned a unique 40 character ID string that can be passed to the `clonecache` function. Any unique prefix of this ID string can also be used, and typically 4—8 characters is enough.

```
> library(cacher)
> clonecache(id = "092d")
created cache directory '.cache'
```

By default, `clonecache` downloads the source files and various metadata files about the analysis but does not download any data files. We can see what source files are available in this cache by using the `showfiles` function. In this case there is only one file and so we deisgnate that file as the "active" file via the sourcefile function.

```
> showfiles()
[1] "top20.R"
> sourcefile("top20.R")
```

In this case, because there is only one source file available in this package, there is no need to call the `sourcefile` function explicitly because that file will be used by default. For analyses involving multiple files, the `sourcefile` function needs to be called to indicate the file to be examined. To obtain a listing of the code in the analysis we can use the code function. By default, the code function shows an abbreviated one-line representation of each expression.

```
> code()
source file: top20.R
1  cities <- readLines("citylist.txt")
2  classes <- readLines("colClasses.txt")
3  vars <- c("date", "dow", "death",
4  data <- lapply(cities, function(city) {
5  names(data) <- cities
6  estimates <- sapply(data, function(city) {
7  effect <- weighted.mean(estimates[1,
8  stderr <- sqrt(1/sum(1/estimates[2,
```

The original source file for this analysis was called "top20.R" (shown at the top of the listing) and there are 8 expressions in the analysis. To the left of each expression is its expression sequence number. We can get a quick "sense" of the analysis by calling `graphcode` to see how the various objects relate to each other. The graph produced by `graphcode` is shown in Figure 1. From the graph we can see clearly that a number of elements come together to form the "data" which leads to an object called "estimates". From the estimates we obtain an effect and a standard error. Given this visualization of the analysis code itself, we can decide on which objects we might wish to inspect more closely. For example, we can examine the code expressions that lead to the creation of the "data" object using the `objectcode` function.

```
> objectcode("data")
source file: top20.R
1  cities <- readLines("citylist.txt")
2  classes <- readLines("colClasses.txt")
3  vars <- c("date", "dow", "death", "tmpd", "rmtmpd", "dptp",
           "rmdptp", "l1pm10tmean")
4  data <- lapply(cities, function(city) {
       filename <- file.path("data", paste(city, "csv",
```

```
                                sep = "."))
            d0 <- read.csv(filename, colClasses = classes,
                           nrow = 5200)
            d0[, vars]
    })
5   names(data) <- cities
```

We can also inspect individual objects by printing them to the console. Here we examine the "cities" object which is simply a character vector that contains the abbreviated names of the 20 cities used in the analysis. Before examining the object, we must load it from the cache with the `loadcache` function.

```
> loadcache()
> cities
/ transferring cache db file b8fd490bcf1d48cd06...
 [1] "la"   "ny"   "chic" "dlft" "hous" "phoe"
 [7] "staa" "sand" "miam" "det"  "seat" "sanb"
[13] "sanj" "minn" "rive" "phil" "atla" "oakl"
[19] "denv" "clev"
```

The `loadcache` function does not load the object directly, but rather "lazy-loads" the object into the workspace. When the "cities" object is accessed for the first time it is downloaded from the remote cache and then made available to the user. If the "verbose" option is set to `TRUE` for cacher (via the `setConfig` function), then the message "transferring cache db file" will be printed. This message indicates that an object needs to be downloaded from the remote cache. Once an object has been downloaded it is available for future access and does not need to be downloaded again.

Finally we can see the estimated effect pooled across the 20 cities.

```
> effect
/ transferring cache db file 584115c69e5e2a4ae5...
[1] 0.0002313219
```

This would translate into an approximately 0.23% increase in daily mortality associated with a 10 unit increase in ambient PM air pollution (10 units is a commonly used increment).

Summary

In this chapter we have given a brief presentation of the capabilities of the cacher and CodeDepends packages. These packages provide functions for caching, distributing, exploring, and visualizing statistical analyses conducted in R. For the purposes of reproducible research, obtaining data and code as well as visualizing the flow of an analysis is critical. The cacher package comes with a vignette that contains more details of the functions in the package. Both packages are currently under active development and the addition of features and capabilities is planned for future releases.

References

Baggerly, K., Morris, J., Edmonson, S., and Coombes, K. (2005), “Signal in noise: evaluating reported reproducibility of serum proteomic tests for ovarian cancer,” *J. Natl. Cancer Inst.*, 97, 307–309.

Laine, C., Goodman, S. N., Griswold, M. E., and Sox, H. C. (2007), “Reproducible Research: Moving toward Research the Public Can Really Trust,” *Annals of Internal Medicine*, 146, 450–453.

Peng RD (2008). "Caching and distributing statistical analyses in R," *Journal of Statistical Software*, 26 (7), 1--24.

Peng RD, Dominici F (2008). *Statistical Methods for Environmental Epidemiology in R: A Case Study in Air Pollution and Health*. Springer, NY.

Peng RD, Eckel SP (2009). "Distributed reproducible research using cached computations," *IEEE Computing in Science and Engineering*, 11 (1), 28–34.

Samet JM, Dominici F, Curriero F, Coursac I, and Zeger SL. "Particulate Air Pollution and Mortality: Findings from 20 U.S. Cities," *New England Journal of Medicine*, 343 (24),1742-1757.

Figure Captions

Figure 1: Graph of statistical analysis code.

Figures

Figure 1