4-29-2008

Caching and Distributing Statistical Analyses in R

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

Caching and Distributing Statistical Analyses in R

Roger D. Peng
Department of Biostatistics
Johns Hopkins Bloomberg School of Public Health

April 29, 2008

Abstract

We present the **cacher** package for R, which provides tools for caching statistical analyses and for distributing these analyses to others in an efficient manner. 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 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. We describe the design and implementation of the **cacher** package and provide two examples of how the package can be used for reproducible research.

1 Introduction

Reproducible research is a phrase that is used to describe research where the published results are accompanied by the software environment and data used to produce these results (Buckheit and Donoho, 1995; Schwab et al., 2000; Gentleman and Temple Lang, 2007). With the data and software, results can be recreated independently by running the original analysis programs on the original data. Reproduction is distinguished from replication in that replication requires an independent investigator to obtain similar results using new data and a comparable or identical analytic approach. A minimum requirement for reproducible research is that the data and computer programs used to analyze the data are made available and distributed to others (Peng et al., 2006).

The distribution of reproducible research 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 re-run 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 reproduce or verify some of the results.

A framework in which reproducible research can be distributed using *cached computations* is described in Peng and Eckel (2007). Cached computations are results that are stored in a database as an analysis is being conducted. These stored results can be distributed via websites or central repositories so that others may explore the datasets and computer code for a given scientific investigation. A set of R packages (**cacheSweave**, **stashR**, **SRPM**) is available from CRAN that can be used in conjunction with the literate programming tools Sweave and LaTeX to cache computations in key-value databases.

In this paper we describe in detail the design and implementation of the **cacher** package, which provides tools for caching arbitrary statistical analyses and distributing them over the web. The conceptual model that we use is similar to that of Peng and Eckel (2007) in that the **cacher** package also uses databases of cached computations. However, we will not be working in the specific context of producing a journal article or technical document. Rather, we will consider statistical analyses more generally as source files to be evaluated in a statistical analysis environment such as R. Theses source files serve the purposes of reading in datasets, incorporating code from other sources (e.g. loading other R packages, reading external source files),

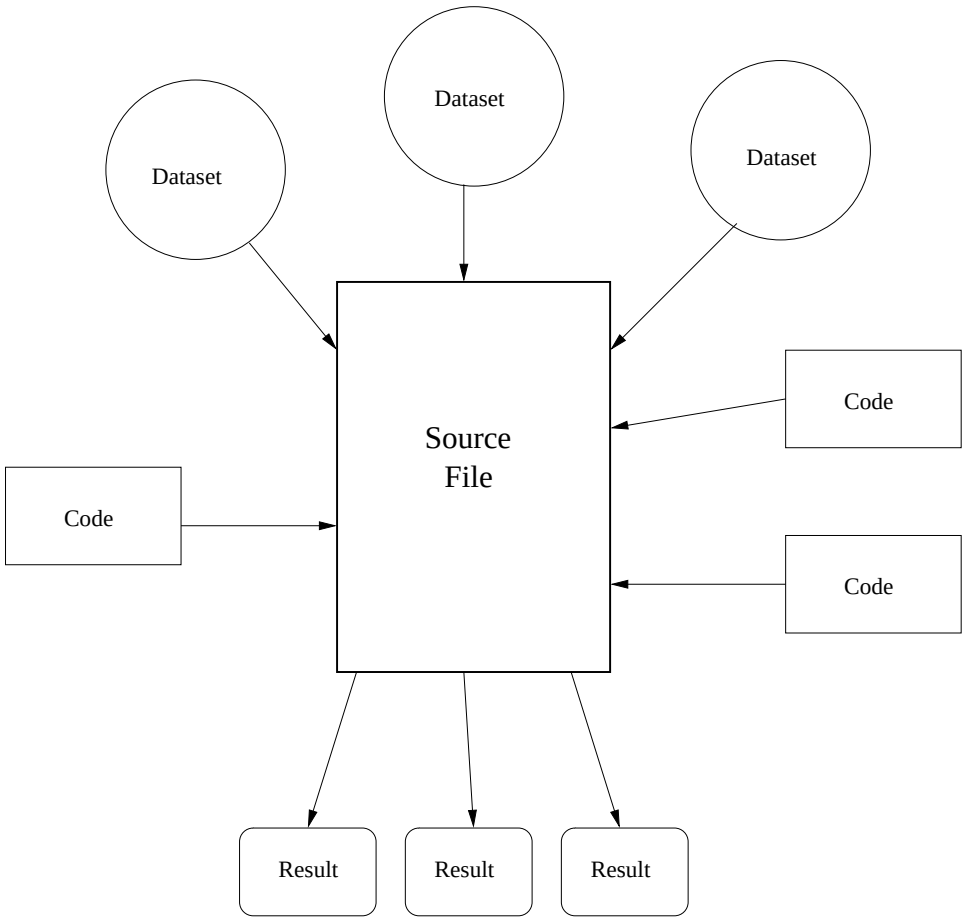

Figure 1: Conceptual model for the **cacher** package.

executing analysis code, and producing results. This conceptual model is sketched in Figure 1. The **cacher** package caches the results produced along with any datasets or external code that have been incorporated into the analysis. This way the reader can have access to those objects when attempting to reproduce the analysis.

In Sections 2–5 we describe the basic features of the **cacher** package. Section 6 provides examples of how the **cacher** package can be used to cache, distribute, and verify statistical analyses.

2 Caching Statistical Analyses

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.

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 (defaults to `.cache`) and the log file (defaults to creating a log file in the cache directory).

The simplest invocation of `cacher` is

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

where `myanalysis.R` is an R source file. To print log messages to the console, one can invoke

```
> cacher("myanalysis.R", logfile = NA)
```

and obtain more information about what `cacher` is doing.

The basic procedure of `cacher` is to

1. parse the R source file;
2. create the necessary cache directories and subdirectories;
3. set various configuration variables and hook functions for plotting (see Section 2.1);
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,
   (c) if an expression does not create any R objects (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.

The `cacher` function identifies each expression in a source file by taking the SHA-1 digest of the expression, the expression history, and the name of the source file. The expression history is simply the expression object containing every expression preceding the current expression. For the first expression, the expression history is of length zero. Using the expression history is a way to prevent expressions such as

```
> y <- x^2
```

from being inappropriately loaded from the cache. Such an expression 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.

For each cached expression, a database file is created in the database directory of the cache containing the serialized R objects associated with that expression (if any). If `cacher` encounters an expression that has already been evaluated and for which objects exist in the database, those objects will be lazy-loaded into the user's workspace (see e.g. Ripley, 2004). Hence, an expression that has not been alterned since a previous evaluation does not need to be reevaluated—often loading objects from the cache will be faster than reevaluating the expression.

In addition, there may be instances where an expression need not be evaluated after its initial evaluation. For example, take the following sequence of expressions.

```
> x <- readLargeDataFrame("largeDataFrame.txt")
> s <- summary(x)
> print(s)
```

In the initial evaluation, the large data frome `x` is loaded and its summary is computed and stored in `s`. In subsequent invocations of `cacher`, `x` and `s` will be lazy-loaded into the workspace. If our only interest is in seeing the summary of `x`, then the `print(s)` expression only needs to load the object `s` from the database. The large object `x` is not needed anymore and will not be loaded into the workspace.

2.0.1 Metadata

As the `cacher` function is running, it writes out metadata for each expression to a metadata file in the cache directory. This metadata file is not used directly by the `cacher` function, but it is used by the various other functions for exploring a cached analysis (these functions are described in Section 4). Each source file processed by `cacher` possesses its own metadata file and each entry of the metadata file corresponds to an expression in the source file. For each expression, the metadata file contains a snippet of the expression itself, the expression's SHA-1 digest, the names of any R objects created by the expression, and whether evaluation of the expression needs to be forced.

2.0.2 Using Multiple Source Files

As mentioned above, each expression in a source file is identified by the digest of the expression itself, the expression history, and the name of the source file. The reason for including the name of the source file is that a given cache directory can be used to process multiple source files. Since the same expression may occur in different source files, it is important that we not load the value for an expression associated with one file while processing another source file. In Section 4 we describe how the user can switch between exploring analyses from different source files using the `sourcefile` function.

2.1 Expressions with Side Effects

Simple expressions, such as assignments, will typically result in a single object being created in the global environment. For example, the expression

```
> x <- 1:100
```

results in an object named `x` being created in the global environment whose value is an integer sequence from 1 to 100.

However, there are other types of expressions which can result in either multiple objects being created in the user's workspace or no objects being created. For example, the `source` function is often used to load objects from an R code file. Unless the `local` argument is set to `TRUE`, these objects will by default be created in the global environment. When the `cacher` function evaluates an expression that contains a call to `source`, there will be objects created outside of the temporary environment in which the expression is evaluated (again, unless the argument `local = TRUE` is specified in the call to `source`). The `set.seed` function behaves in a similar way by modifying (or creating) the `.Random.seed` object in the global environment.

In order to handle the effects of functions like `source` the function `evalAndCache`, which evalutes an expression and saves the results to the cache database, first obtains a character vector of the names of all the objects in the global environment. After evaluating the expression in a temporary environment, a check is made to see if any new objects have been created or modified in the global environment. If so, those objects are saved to the database as well as any objects that were created in the temporary environment. Note that we currently make a special case of the global environment. If the code being evaluated creates objects in some other environment, then `cacher` will not be able to cache those objects.

Another example of a function with side effects is the `plot` function (and related functions) from the **graphics** package. Since `plot` does not create any objects in the global environment, but rather creates a plot on a graphics device, there is nothing for `cacher` to cache. Currently, the approach of `cacher` is to detect when a plot has been created by setting a hook function for the `plot.new` function. Each time `plot.new` is called, an internal flag is set so that `cacher` knows that evaluation of this expression needs to be forced rather than cached. We similarly set a hook function for `grid.new` to detect the creation of **lattice** plots.

There are many other types of expressions that have side effects and do not result in the creation of objects in the global environment. Expressions such as calls to `system` or functions which write out files (e.g. `save`, `save.image`, `write.table`, `dput`, etc.) all result in objects being created outside of R. In general, these expressions cannot yet take advantage of the caching mechanism in **cacher** and must be executed every time `cacher` is run.

2.2 Caching Individual Expressions

The `cc` function can be used to cache individual expressions as opposed to caching the entire contents of a source file. The usage of `cc` is likely to be independent of `cacher` in that the two functions do not need to be used in conjunction with each other. However, since the `cc` function uses of the `cacher` implementation, it makes sense to describe the operation of `cc` here.

The `cc` function takes an expression as its first argument and then a cache directory path and a source file name. The source file name is optional and defaults to `NULL`; the cache directory defaults to ".cache". If the source file argument is `NULL`, then `cc` computes an SHA-1 digest of the expression using the **digest** package (Eddelbuettel et al., 2007) and uses the digest as the source file name. The expression code is then written to a temporary file with this name and the `cacher` function is subsequently called using this

temporary file as input. Hence, `cc` is essentially a wrapper for `cacher` which generates a source file name from a given expression. When `cc` evaluates an expression it may find that the source file associated with that expression already exists in the cache directory. In that case, evaluation of the expression is skipped and the results of the evaluation are instead loaded from the cache.

The `cc` function can be used inside longer computations where a user might want to cache a critical section which is time consuming or resource intensive. For example, take the following pseudo-analysis,

```
> x <- readDataset("mydataset.txt")
> results <- longCalculation(x)
> answer <- transformation(results)
```

where `longCalculation` might implement a resampling or Markov Chain Monte Carlo algorithm. If that section is not likely to change, then it may be worthwhile to cache the results (i.e. "`results`" and "`answer`") of the first evaluation and subsequently load the results from the cache.

```
> cc({
+     x <- readDataset("mydataset.txt")
+     results <- longCalculation(x)
+     answer <- transformation(results)
+ })
```

This way, if other parts of the analysis following this section, are modified and need to be reevaluated, they can rely on the cached results. However, if the cached section depends on expressions that precede, then the cached section may need to be reevaluated if those expressions change. If the expression wrapped in the `cc` command itself changes, `cc` will detect that change and reevaluate the expression from scratch.

The **weaver** package (Falcon, 2007) from the Bioconductor Project and the **cacheSweave** package implement similar functionality in the context of using Sweave and LaTeX. The **weaver** package caches the results of expressions in separate files stored in the R workspace format. These results are then loaded (rather than lazy-loaded) into R upon subsequent evaluation of the same expression. The **cacheSweave** package uses the **stashR** package for its database backend.

3 Distributing a Cached Analysis over the Web

Users 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.

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. 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. In order to force the downloading of all database objects when cloning, the user needs to set `all.files = TRUE` when calling `clonecache`. Once an analysis is cloned the functions described in Section 4 can be used to explore the code and data objects in the analysis.

4 Exploring a Cached Analysis

The **cacher** package provides some basic tools to allow users to interact with the code and data provided in a cached analysis. The following functions make up the primary user interface for readers of a cached analysis.

- `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. If any expressions have been marked to be skipped by `skipcode`, those expressions will be annotated with an asterisk.
- `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 (see Section 3), 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.
- `skipcode`: Force certain expressions to be skipped from evaluation when using the `runcode` function (for example, if certain external resources are not available). There is a globally maintained list of expressions that will be skipped for a given source file. If `num` is `NULL`, then the list of skipped expressions is cleared.
- `showobjects`: Given an expression sequence number, `showobjects` shows what objects were created (and hence cached) by that expression. These objects can subsequently be loaded into the workspace with `loadcache`. If `num` is a sequence, then a single character vector is returned containing the union of the names of the objects cached.

5 Verifying an Analysis

The **cacher** package provides the `checkcode` function for verifying the objects in a cached analysis. A user who has downloaded a cached analysis via `clonecache` or in some other manner can verify a given R expression by evaluating the code on his/her own machine and checking to see if the resulting object is equivalent to the object stored in the database corresponding to that expression. The `checkcode` function takes a numeric vector of expression sequence numbers and evaluates the corresponding expressions while verifying that the resulting objects match the cached objects. The comparison of objects is done with the `all.equal` function to allow for some minor differences, for example, with floating point calculations. Called with no arguments, `checkcode` will check every expression in the source file. If a given expression does not have any R objects associated with it, then there is nothing to check and `checkcode` moves to the next expression.

When `checkcode` encounters an expression that cannot be evaluated or where the computed object does not match the cached object, a message is printed to the console indicating the problem. In addition, `checkcode` will lazy-load the cached object into the evaluation environment and continue checking subsequent expressions in the source file. Therefore, any expressions which depend on the object corresponding to the non-verified expression will use the cached object rather than the computed one.

For example, an analysis will typically contain an expression which reads in a dataset using a function such as `load` or `read.table`. These functions read data from external connections (usually files) and load them into the user's workspace. If the user does not also have possession of these external files, then there is no way for the user to verify the evaluation of that expression. However, the data from the file is nevertheless

cached in the database so it is possible to evaluate subsequent expressions based on the cached version of the data. Section 6.1.5 gives an example of how `checkcode` handles this particular situation.

5.1 Verifying the Integrity of Objects

In addition to comparing the output of evaluating code with cached objects, one can check the integerity of the cached objects to make sure that there has not be any corruption to the files (particularly, when being transferred over the network). Each object stored in the cache has with it the SHA-1 digest of the object itself. The `checkobjects` function can be used to compare this SHA-1 digest with the digest of the object. Any corruption of the data will result in a mismatch between the stored digest and computed digest.

6 Examples

6.1 Basic Usage

To illustrate some of the features of the **cacher** package we will use the following simple statistical analysis of the `airquality` dataset from the **datasets** package which comes with R. The code for the entire analysis is printed below.

```
library(datasets)
library(stats)

data(airquality)

fit <- lm(Ozone ~ Wind + Temp + Solar.R, data = airquality)
summary(fit)

## Plot some diagnostics
par(mfrow = c(2, 2))
plot(fit)

## Interesting non-linear relationship
temp <- airquality$Temp
ozone <- airquality$Ozone

par(mfrow = c(1, 1))
plot(temp, ozone)
```

The code is contained in a file called "sample.R" which comes with the **cacher** package. The above analysis is fairly simple and not very time-consuming so it is easily reproduced by anyone who can run R, without any need for caching. Nevertheless, it is useful for demonstrating how the **cacher** package works.

The first step is to install the **cacher** package from the Comprehensive R Archive Network (CRAN) and load it into R.

```
> library(cacher)
> setConfig("verbose", TRUE)
```

For now, we also set the global `verbose` option to be `TRUE`, making `cacher` be somewhat more "chatty" (the default is `FALSE`).

The `cacher` function accepts a file name as its first argument. This file should contain the code for the analysis that you want to cache. Other arguments include the name of the cache directory (defaults to `.cache`) and the name of the log file (defaults to `NULL`). If `logfile = NULL` then messages will be printed to a file in the cache directory. Setting `logfile = NA` will send messages to the console.

The "sample.R" file containing the above analysis comes with the **cacher** package and can be copied into your working directory. Given a file containing the code of an analysis, you can call the `cacher` function as

```
> cacher("sample.R")

creating cache directory '.cache'

Call:
lm(formula = Ozone ~ Wind + Temp + Solar.R, data = airquality)

Residuals:
    Min      1Q  Median      3Q     Max
-40.485 -14.219  -3.551  10.097  95.619

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -64.34208   23.05472  -2.791  0.00623
Wind         -3.33359    0.65441  -5.094 1.52e-06
Temp          1.65209    0.25353   6.516 2.42e-09
Solar.R       0.05982    0.02319   2.580  0.01124

Residual standard error: 21.18 on 107 degrees of freedom
  (42 observations deleted due to missingness)
Multiple R-squared: 0.6059,        Adjusted R-squared: 0.5948
F-statistic: 54.83 on 3 and 107 DF,  p-value: < 2.2e-16
```

The `cacher` function evaluates each expression in the file and prints any resulting output to the console. For example, the summary of the fitted linear model is printed to the console while the two plots are sent to the appropriate graphics device. The log messages are written to a file in `.cache/log/sample.R.log` which contains information about each expression evaluated by `cacher`. We will discuss the contents of the log file in Section 6.1.2.

When cacher evaluates each code expression, the results of the evaluation are cached to the database and lazy-loaded back into the workspace. After running the "sample.R" analysis, we can see that there are the following objects now in the workspace:

```
> ls()

[1] "airquality" "fit"        "ozone"      "temp"
```

Since the objects are lazy-loaded, they do not occupy any memory until they are accessed. The lazy-loading is not as important on the first evaluation but can reduce the amount of evaluation time required on subsequent analysis

For example, take the following very simple set of expressions.

```
x <- rnorm(1000000)
s <- summary(x)
print(s)
```

The first expression creates a vector of 1 million standard Normal random variates and the second computes a summary (a five-number summary plus the mean). The amount of time to evaluate these expressions the first time is

```
> systime <- system.time(cacher("bigvector.R"))

      Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
-4.8930000 -0.6747000 -0.0007293 -0.0012700  0.6721000  5.0190000

> print(systime)

   user  system elapsed
  1.864   0.296   2.159
```

In this case, the evaluation time is about 1.9 seconds. After running `cacher`, the objects `x` and `s` reside in the workspace and have been cached to the database. Subsequent evaluations of the same code should take much less time since we can simply load `x` and `s` from the cache.

```
> rm(x, s)
> systime <- system.time(cacher("bigvector.R"))

      Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
-4.8930000 -0.6747000 -0.0007293 -0.0012700  0.6721000  5.0190000

> print(systime)

   user  system elapsed
  0.028   0.000   0.027
```

Now the evaluation only takes about 0.028 seconds. In fact, the analysis here is particularly quick because we do not need the `x` vector at all. We can simply print the summary object `s`.

Even if we did need to access the vector `x`, loading from the cache is often faster than regenerating all of the random Normals using `rnorm`. For example, if we wanted to calculate the 95th percentile of the data, then we could simply do

```
> systime <- system.time(q95 <- quantile(x, 0.95))
> print(q95)

     95%
1.643605

> print(systime)

   user  system elapsed
  0.500   0.044   0.544
```

6.1.1 Exploring a cached analysis

Once an analysis has been cached using `cacher`, it can be explored using the utilities provided in the **cacher** package. Since you can cache analyses from multiple files (as we have done above), we can show which analyses have already been cached using the `showfiles` function.

```
> showfiles()

[1] "sample.R"    "bigvector.R"
```

Here we see the file names corresponding to the two files that we analyzed in the previous section. If you want to examine a particular analysis, you can use the `sourcefile` function to choose that analysis and `showcode` will simply display the raw source file.

```
> sourcefile("bigvector.R")
> showcode()

x <- rnorm(1000000)
s <- summary(x)
print(s)
```

You can also use the `code` function to display the code in a summary form

```
> sourcefile("sample.R")
> code()
```

```
source file: sample.R
1  library(datasets)
2  library(stats)
3  data(airquality)
4  fit <- lm(Ozone ~ Wind + Temp +
5  summary(fit)
6  par(mfrow = c(2, 2))
7  plot(fit)
8  temp <- airquality$Temp
9  ozone <- airquality$Ozone
10  par(mfrow = c(1, 1))
11  plot(temp, ozone)
```

The `code` function truncates expressions to a single line and also shows the sequence number assigned to each expression in the order that the expression is encountered in the source file. In order to see the full code for each expression, you can set the `full = TRUE` option to `code`.

The first thing you might do when exploring a cached analysis is to explore the elements of the cache database itself. You can list the objects available using the `showobjects` function, which returns a character vector of the names of each object in the database. Passing an expression sequence number to `showobjects` via the `num` argument shows the objects created by that expression.

```
> showobjects()

[1] "airquality" "fit"        "temp"       "ozone"

> showobjects(8)

[1] "temp"

> showobjects(1)

character(0)
```

These objects can be lazy-loaded into the workspace using the `loadcache` function.

```
> loadcache()
> ls()

[1] "airquality" "fit"        "ozone"      "temp"
```

Now, we can print the linear model fit (without actually fitting the model) by calling

```
> print(fit)

Call:
lm(formula = Ozone ~ Wind + Temp + Solar.R, data = airquality)

Coefficients:
(Intercept)         Wind         Temp      Solar.R
  -64.34208     -3.33359      1.65209      0.05982
```

The `loadcache` function takes a `num` argument which can be a vector of indices indicating code expression sequence numbers. For example, if you want to load only the objects associated with expression 4 (i.e. the `fit` object), then you can call `loadcache(4)`.

In addition to exploring the objects in the cache database, you may wish to run the analysis on your own computer for the purposes of reproducing the original results. You can run individual expressions or a sequence of expressions with the `runcode` function. The `runcode` function accepts a number or a sequence of numbers indicating expressions in an analysis. For example, in order to run the first four expressions in the "sample.R" analysis, we could call

```
> rm(list = ls())
> code(1:4)

source file: sample.R
1  library(datasets)
2  library(stats)
3  data(airquality)
4  fit <- lm(Ozone ~ Wind + Temp +

> runcode(1:4)

evaluating expression 1
evaluating expression 2
loading cache for expression 3
loading cache for expression 4

> ls()

[1] "airquality" "fit"
```

In this case, expressions 1 and 2 are evaluated but expressions 3 and 4 are loaded from the cache. By default, `runcode` does not evaluate expressions for which it can load the results from the cache. In order to force evaluation of all expressions, you need to set the option `forceAll = TRUE`.

6.1.2 Understanding the log file

As each expression is being evaluated, `cacher` keeps track of which expressions result in the creation of new objects (including modification of existing objects) and which expressions have side effects. Expressions with side effects cannot be cached and therefore must always be evaluated. The primary operation falling into this category is plotting, which launches a graphics device and makes changes to that device. One exception is **lattice** plots which can be stored as objects and therefore cached. The log file contains information about each expression and whether it needs to force evaluation. Here we print the first few lines of the log file for this analysis.

```
1: library(datasets)
  eval expr and cache
  expression has side effect: f92adaa84e2ae7800e91ee5fead3a3db06d18f9a
2: library(stats)
  eval expr and cache
  expression has side effect: 223a036ba6a2561e5c23716840e9090670824ff4
3: data(airquality)
  eval expr and cache
4: fit <- lm(Ozone ~ Wind + Temp +
  eval expr and cache
5: summary(fit)
  eval expr and cache
```

Understanding the log file output is not critical to using `cacher` but it is occasionally useful to know what the function is doing for a given expression. Each expression is assigned a number based on when it is encountered in the source file and a snippet of the expression is printed immediately after the number. Below, `cacher` will indicate if the expression needs to be evaluated and cached and will try to determine if the expression resulted in a side effect. The check for side effects is rudimentary and will not catch all cases. Once the expression has been cached, `cacher` will reload the results from the cache into the global environment (i.e. workspace) and move to the next expression.

Running the analysis a second time with `cacher` results in the following log file being generated.

```
1: library(datasets)
  -- loading expr from cache
2: library(stats)
  force expression evaluation
3: data(airquality)
  -- loading expr from cache
4: fit <- lm(Ozone ~ Wind + Temp +
  -- loading expr from cache
```

Here we see that expressions 1 and 2 were forced to be evaluated because the `library` function results in a side effect (i.e. altering the search list). Expressions 3 and 4 create objects in the workspace so they can be lazy-loaded from the cache. Note here that although the `airquality` dataset is loaded from the cache, it is not needed if you are primarily interested in examining the `fit` object from the `lm` call. This is where lazy-loading is very useful. However, if you want to fit a different model, say, with some interactions, then of course the original data will be loaded into the workspace the first time it is accessed.

6.1.3 Caching Individual Expressions

Indivdual expressions can be cached using the `cc` function. For example, you might want to cache just a single critical section of an analysis rather than the entire analysis itself. In this case, you can wrap the experssion with the `cc` function and subsequent evalutions of the expression will be loaded from the cache (assuming the expression hasn't changed in the meantime).

For example, the following artificial expression takes at least 2 seconds to evaluate before the value of 5 is assigned to `x`.

```
> cc({
+     x <- local({
+         Sys.sleep(2)
+         5
+     })
+ })
```

Caching the expression with `cc` allows the value of `x` to be loaded from the cache on subsequent evaluations. The cache database corresponding to this expression can be explored as in the previous examples.

```
> showfiles()

[1] "4a99d95e572b4879d1782c76567bdf8ac678e9f3"
```

Note that since there is no source file to use, a file name is generated based on a digest of the expression being cached.

```
> sourcefile(showfiles())
> showcode()

{
    x <- local({
        Sys.sleep(2)
        5
    })
}
```

It is probably not so useful to explore the cache associated with a single expression. More likely, you will want to cache an expression so that subsequent evaluations of the same expression will be loaded from the cache and therefore not take as long.

6.1.4 Posting a cache directory

If you have access to a webserver you can post your cache directory directly on the webserver for others to access. Once made available on a webserver, others can access your cache directory by using the `clonecache` function in the **cacher** package and the URL of the directory on your webserver. For example, we can download the analysis corresponding to the "bigvector.R" file by calling

```
> clonecache("http://www.biostat.jhsph.edu/rr/bigvector.cache")

created cache directory '.cache'
downloading source file list
downloading metadata
downloading source files
downloading cache database file list
```

This call to `clonecache` downloads all of the relevant cache files related to the analysis except for the cache database files. In order to download the cache database files, the option `all.files = TRUE` must be set.

Once a cache package has been downloaded using `clonecache` you can use all of the tools described in the previous sections to explore the cache and the run some of the analyses.

```
> showfiles()

[1] "bigvector.R"

> sourcefile("bigvector.R")
> code()

source file: bigvector.R
1  x <- rnorm(1000000)
2  s <- summary(x)
3  print(s)

> showobjects()

[1] "x" "s"

> loadcache()
> print(s)

/ transferring cache db file d7952a4732ffa55c045958205340fa4c1b68b608
      Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
-4.6570000 -0.6737000  0.0006063  0.0012460  0.6755000  5.1400000
```

By default, `clonecache` does not download the cache database files until they are needed in order to minimize the amount of data that is transferred. Cache database files are only transferred from the remote host when the objects associated with them are first accessed.

In the above example, the database file corresponding to the object `s` is only transferred when we call `print(s)`. When a database object has to be downloaded from the remote site, a message will be printed to the screen indicating the transfer.

6.1.5 Verifying a Cached Analysis

Once you have cloned an analysis conducted by someone else, you may wish to verify that the computation that you run on your computer leads to the same results that the original author obtained on his/her computer. This can be done with the `checkcode` function. The `checkcode` function essentially evaluates each expression locally (if it can) and compares the output with the corresponding value stored in the cache database.

If the locally created object and the cached object are the same, then that expression is considered verified. If an expression does not create any objects, then there is nothing to compare. If the locally created object and the cached object are different, the the verification fails and `checkcode` will indicate which objects it could not verify.

For example, we can run the `checkcode` function on the analysis of the `airquality` dataset from before. Here we will only check the first four code expressions.

```
> unlink(".cache", recursive = TRUE)
> clonecache("http://www.biostat.jhsph.edu/rr/combined.cache")

created cache directory '.cache'
downloading source file list
downloading metadata
downloading source files
downloading cache database file list
downloading metadata
downloading source files
downloading cache database file list

> sourcefile("sample.R")
> showobjects(1:4)

[1] "airquality" "fit"

> checkcode(1:4)

evaluating expression 1
evaluating expression 2
checking expression 3
/ transferring cache db file 142d241ba5b4fbb5646a189fce5c552fd67706e7
+ object 'airquality' OK
checking expression 4
/ transferring cache db file ad5720cbda29135e8412d130a6de9dbff3d0e5e7
+ object 'fit' OK
```

In the first four expressions, there are two objects created: the dataset `airquality` and the linear model object `fit`. The `checkcode` function compares each of those objects with the version stored in the cache database (which we previously cloned from the web). In this case, the objects match and the computations are verified. Notice that in expression 3, the database file for the `airquality` object had to be downloaded so that it could be checked against the locally created version.

We can check the code in the "bigvector.R" analysis also. In this analysis there are two objects that need to be verified: `x`, the vector of standard Normals and `s` the "summary" object.

```
> sourcefile("bigvector.R")
> checkcode()

checking expression 1
/ transferring cache db file fb877f8375799370cef47fce86a93acc60abd26d
- object 'x' not verified, FAILED
- Mean relative difference: 1.414591
checking expression 2
/ transferring cache db file d7952a4732ffa55c045958205340fa4c1b68b608
- object 's' not verified, FAILED
- Mean relative difference: 0.009960198
evaluating expression 3
       Min.     1st Qu.      Median        Mean     3rd Qu.        Max.
-4.99800000 -0.67540000  0.00006827 -0.00002781  0.67470000  4.87800000
```

Notice that expressions 1 and 2 failed for a common reason (expression 3 had no objects to verify). Since the analysis did not set the random number generator seed in the beginning, the generation of the Normal random variates on the local machine is not the same as that for the original analysis. Therefore, the object `x` is not reproducible (nor is `s`).

Of course, there are limitations to verifying statistical analyses. Analyses may take a long time to run and therefore it may take a long time to verify a given computation. If one does not have the necessary external resources (i.e. hardware, software) then it may not be possible to verify an analysis at all. Currently, verification of analyses is limited to R objects only. We cannot verify the output of summary or print functions nor can we verify plots (although lattice plots can be verified if they are stored as R objects).

Certain analyses may load external datasets or inputs which will generally not be available to the other users. A typical analysis might be of the form

```
data <- read.csv("faithful.csv")
with(data, plot(waiting, eruptions))

library(splines)
fit <- lm(eruptions ~ ns(waiting, 4), data = data)

xpts <- with(data, seq(min(waiting), max(waiting), len = 100))
lines(xpts, predict(fit, data.frame(waiting = xpts)))
```

This analysis reads in the the "Old Faithful" dataset which contains eruption times and waiting periods for the Old Faithful geyser in Yellowstone National Park. Although this dataset is available from the R installation, we have exported it here to a comma-separated-value file for demonstration.

The original author of this analysis can run the `cacher` function on this analysis file and distributed it to others.

```
> cacher("faithful.R")
```

However, another user (presumably on a different computer) will not be able to verify all of the code in this analysis

```
> sourcefile("faithful.R")
> checkcode()

checking expression 1
- problem evaluating expression, FAILED
- simpleWarning: cannot open file 'faithful.csv': No such file or
- directory
- loading objects from cache
/ transferring cache db file 255fb954f855b0e53bafa43091fc99e086ef757a
evaluating expression 2
evaluating expression 3
checking expression 4
/ transferring cache db file a15033591616f8a9b693ca1e4faf86aff95e1316
+ object 'fit' OK
checking expression 5
/ transferring cache db file bed8272d401434750a5c0c4c47ce13383cdb1cff
+ object 'xpts' OK
evaluating expression 6
```

Here, the first expression, which reads the dataset in via `read.csv` cannot be verified because the "faithful.csv" file is not available. However, the other expressions can be run on the local machine and are verifiable since they can use the cached copy of the dataset.

6.2 Conducting an Alternate Analysis

In this section we will illustrate the use of the **cacher** package to reproduce some results from a large epidemiological study of the health effects of fine particulate matter air pollution. This study was a multi-site time series study examining the short-term relationship between particulate matter $\leq 2.5\mu$m in aerodynamic diameter ($PM_{2.5}$) and daily hospital admission rates for various cardiovascular and respiratory diseases (Dominici et al., 2006).

This study produced a county-specific estimate of the log relative risk relating increases in daily $PM_{2.5}$ with daily hospital admission rates. These risks can be found at the study's website at http://www.biostat.jhsph.edu/MCAPS/ Below, we present a sensitivity analysis of these log relative risks and demonstrate how they can be pooled together to obtain a "national average" risk estimate using a two-level Normal hierarchical model (more details in Dominici et al., 2000).

First, we can clone the cached analysis by calling `clonecache`.

```
> clonecache("http://www.biostat.jhsph.edu/rr/mcaps.cache")

created cache directory '.cache'
downloading source file list
downloading metadata
downloading source files
downloading cache database file list
```

Here we see that there is only one source file available, the `mcaps.R` file.

```
> showfiles()

[1] "mcaps.R"

> sourcefile("mcaps.R")
```

We can list the code expressions with the `code` function.

```
> code(1:7)

source file: mcaps.R
1  Sys.setlocale(locale = "C")
2  estimates <- read.csv("http://www.biostat.jhsph.edu/MCAPS/estimates-subset...
3  estimates <- transform(estimates,
4  library(tlnise)
5  HF <- subset(estimates, outcome ==
6  initTLNise()
7  pooled <- with(HF, tlnise(beta,
```

The first six code expressions read the data from the website and pool the risk estimaets for heart failure across the 202 counties in the study. For the pooling, we use Phil Everson's TLNise software (Everson and Morris, 2000), an R version of which is available on CRAN. The first thing we can do is the verify that we are capable of producing the same results that the original authors did. The `checkcode` function can be used to check the first six expressions.

```
> checkcode(1:7)

evaluating expression 1
checking expression 2
/ transferring cache db file d33bf70c06481a745ebfd57a0f0e55fcef3d1342
+ object 'estimates' OK
checking expression 3
/ transferring cache db file 0d389a6f8121c60b66c13389cf3a0258bcb506da
+ object 'estimates' OK
```

```
evaluating expression 4
Two-level normal independent sampling estimation (version 0.2-7)
checking expression 5
/ transferring cache db file 48dc3abbde978864d5ea447038de4e8f8b848718
+ object 'HF' OK
evaluating expression 6
checking expression 7
/ transferring cache db file 51999c8bd7db146ebe049d4fb4de06d9bc693152
+ object 'pooled' OK
```

Here we see that the six expressions were evaluated properly and the objects created matched those created by the original authors. Database objects were downloaded from the archive as needed.

The original pooled national average log relative risk for hospitalization for heart failure can be found by loading the cached objects for expression 7.

```
> loadcache(7)
> pooled$gamma

          est           se   est/se
0 0.001291823 0.0002505152 5.156663
```

This risk estimate shown in the `est` column can be interpreted as a 1.29% increase in admissions of heart failure associated with a 10 μg/m^3 increase in ambient $PM_{2.5}$.

One important issue in this analysis is the sensitivity of the Bayesian hierarchical model to the specification of the prior distribution. In particular, the TLNise software places a uniform prior on the second-level covariance matrix, sometimes referred to as the heterogeneity matrix, which describes the natural variation of the relative risks across counties. Since the original authors used the default settings, it is of interest to see if the national average estimates vary when this prior specification is altered.

The `tlnise` function has an option called `prior` which can be used to change the nature of the prior distribution on the second-level covariance matrix. Here we try two alternate priors. First, we need to call `loadcache` first in order to obtain the data frame `HF`.

```
> loadcache(1:7)
> library(tlnise)
> p0 <- with(HF, tlnise(beta, var, prnt = FALSE, prior = 0))
> p2 <- with(HF, tlnise(beta, var, prnt = FALSE, prior = 2))
```

We can now compare the estimates obtained using the two alternative prior specifications with the original estimates

```
> rbind(p0$gamma, p2$gamma, pooled$gamma)

          est           se   est/se
0 0.001290191 0.0002493016 5.175222
0 0.001296439 0.0002516423 5.151911
0 0.001291823 0.0002505152 5.156663
```

Here we see that there is some variation between the estimates but the estimates are qualitatively similar.

7 Discussion and Future Work

The idea of a "compendium" is described by Gentleman and Temple Lang (2007) as a way to publish a reproducible analysis by including multiple levels of detail. Readers with a casual interest in the paper may only read the finished product while more interested readers can dig deeper into the specifics of the data and computation. The number of tools for conducting reproducible research via compendiums in R and other languages is generally increasing. R users have tools such as Sweave (Leisch, 2002) and Emacs

Speaks Statistics (Rossini et al., 2004) to assist them in the development of such compendiums as well as the **weaver** package (Falcon, 2007) and others (Peng, 2007) for caching computations. Packages such as the SASWeave (Lenth and Hojsgaard, 2007) have emerged for literate programming using SAS and LaTeX; for non-LaTeX users, the **odfWeave** package (Kuhn and Weaston, 2007) is available for R users wishing to write documents in the Open Document Format (e.g. via OpenOffice).

Currently, authors can use the **cacher** package to cache an analysis and distribute the analysis over the web. The package provides readers tools for downloading these cached analyses and exploring the code and data within them. The **cacher** package is limited in that it cannot take advantage of the extra information provided in the literate programming context. For example, the **cacheSweave** and **weaver** packages organize code by the "chunks" defined in the combined R/LaTeX document. The code chunks can provide extra information about the context of a set of code expressions. For example, it is possible to find out whether a figure is being produced. Without the existence of code chunks, the **cacher** package must evaluate evaluate each code expression individually.

One possible direction for future work is to provide the ability to annotate the code in a source file. Such annotations could provide hints to `cacher` regarding what the code is doing. For example, if a set of expressions gives rise to a figure (e.g. a PDF file), then we can associate that set of expressions with the figure and give the reader more information about the analysis via the reader tools. Similar annotations could be provided for tables and other results that the author deems interesting. This way, a reader interested in particular table/figure of results could quickly identify the segment of code that produced the results.

Another area for further development includes providing tools to help readers edit cached analyses and to integrate their modifications into the original computations. Currently, the reader tools are "read-only", allowing readers to examine and explore an analysis, but not allowing them to edit the original source file. For example, in order to test the sensitivity of an analysis to a set of assumptions (as in Section 6.2), a reader might want to alter a specific group of R expressions and re-run the entire analysis with the altered expressions.

Currently, only R users can interact with the data and code in cache directories that have been posted to the web (i.e. via the **cacher** package's `clonecache` function). It might be desirable to provide a more useful web-based interface to provide more information about each of the cache packages to casual readers. Also, while storing data in R's native serialization format is simple and efficient, selectively using more generic data formats might allow other analysis systems with which people are familar to interact with the data.

8 Acknowledgements

This research was supported in part by a Faculty Innovation Fund award from the Johns Hopkins Bloomberg School of Public Health, grant ES012054-03 from the National Institute of Environmental Health Sciences.

References

Buckheit, J. and Donoho, D. L. (1995), "Wavelab and reproducible research," in *Wavelets and Statistics*, ed. Antoniadis, A., Springer-Verlag, New York.

Dominici, F., Peng, R. D., Bell, M. L., Pham, L., McDermott, A., Zeger, S. L., and Samet, J. M. (2006), "Fine Particulate Air Pollution and Hospital Admission for Cardiovascular and Respiratory Diseases," *Journal of the American Medical Association*, 295, 1127–1134.

Dominici, F., Samet, J. M., and Zeger, S. L. (2000), "Combining Evidence on Air pollution and Daily Mortality from the Twenty Largest US cities: A Hierarchical Modeling Strategy (with discussion)," *Journal of the Royal Statistical Society, Series A*, 163, 263–302.

Eddelbuettel, D., Lucas, A., Tuszynski, J., Bengtsson, H., and Urbanek, S. (2007), *digest: Create cryptographic hash digests of R objects*, r package version 0.3.1.

Everson, P. J. and Morris, C. N. (2000), "Inference for Multivariate Normal Hierarchical Models," *Journal of the Royal Statistical Society, Series B*, 62, 399–412.

Falcon, S. (2007), *weaver: Tools and extensions for processing Sweave documents*, R package version 1.2.0.

Gentleman, R. and Temple Lang, D. (2007), "Statistical Analyses and Reproducible Research," *Journal of Computational and Graphical Statistics*, 16, 1–23.

Kuhn, M. and Weaston, S. (2007), *odfWeave: Sweave processing of Open Document Format (ODF) files*, r package version 0.6.0.

Leisch, F. (2002), "Sweave: Dynamic generation of statistical reports using literate data analysis," in *Compstat 2002 — Proceedings in Computational Statistics*, eds. Härdle, W. and Rönz, B., Physika Verlag, Heidelberg, Germany, pp. 575–580, ISBN 3-7908-1517-9.

Lenth, R. V. and Hojsgaard, S. (2007), "SASWeave: Literate Programming Using SAS," *Journal of Statistical Software*, 19, 1–20.

Peng, R. D. (2007), "A Reproducible Research Toolkit for R," Tech. Rep. 142, Johns Hopkins University Department of Biostatistics, http://www.bepress.com/jhubiostat/paper142.

Peng, R. D., Dominici, F., and Zeger, S. L. (2006), "Reproducible Epidemiologic Research," *American Journal of Epidemiology*, 163, 783–789, doi:10.1093/aje/kwj093.

Peng, R. D. and Eckel, S. P. (2007), "Distributed reproducible research using cached computations," Tech. Rep. 147, Johns Hopkins University Department of Biostatistics, http://www.bepress.com/jhubiostat/paper147/.

Ripley, B. D. (2004), "Lazy Loading and Packages in R 2.0.0," *R News*, 4, 2–4.

Rossini, A. J., Heiberger, R. M., Sparapani, R. A., Mächler, M., and Hornik, K. (2004), "Emacs Speaks Statistics: A Multiplatform, Multipackage Development Environment for Statistical Analysis," *Journal of Computational and Graphical Statistics*, 13, 247–261.

Schwab, M., Karrenbach, N., and Claerbout, J. (2000), "Making scientific computations reproducible," *Computing in Science & Engineering*, 2, 61–67.