Monitoring data in R with the lumberjack package

Mark P.J. van der Loo
Statistics Netherlands

Abstract

Monitoring data while it is processed and transformed can yield detailed insight into the dynamics of a (running) production system. The **lumberjack** package is a lightweight package allowing users to follow how an R object is transformed as it is manipulated by R code. The package abstracts all logging code from the user, who only needs to specify which objects are logged and what information should be logged. A few default loggers are included with the package but the package is extensible through user-defined logger objects.

Keywords: Data Quality, Process Monitoring, Logging, Debugging, R.

1. Introduction

It is common practice to monitor a data analyses process while it is running. Especially in production environments where analyses are run repeatedly on different but structurally comparable data sets. Following a running procedure is usually done with some form of logging system, where the running process updates a log that can be tracked by users as it proceeds.

One can distinguish two types of monitoring. On the one hand there is *process logging*, or just *logging* for short. Here, the running system notifies users of progress and significant events, usually by writing short time-stamped messages to a file (where 'file' can be a flat text file, database, screen or any other device accepting such input). The aim of these messages is to signal whether procedures have concluded successfully, and if they haven't, to report what went wrong. Such information is highly valuable in *post-mortem* investigations, for example when a production script has crashed. On the other hand there is *tracing* where the state of variables is followed over the course of the process. Tracing is usually applied at the development stage as a debugging tool, often using an interactive interface tool to run the code line by line while inspecting the state of variables. One of the purposes of this paper is to demonstrate that targeted forms of automated tracing can be useful at the production stage as well.

The ability to trace the state of variables for debugging purposes is common across languages for technical or statistical computing. Focussing on julia (Bezanson, Edelman, Karpinski, and Shah 2017), python (van Rossum and Drake 2009), and R (R Core Team 2019), we see that all have this capability built into their standard libraries. In julia, the **Debugger** module provides ways to set break points that allow programmers to investigate the scope of a running function at that point, to browse the call stack, and to execute the code step-by-step. Similar functionality is offered in python through the **pdb** module and by R's **base** and **utils** packages. Although there are some differences, the functionality across these languages is comparable.

When it comes to process logging, R differs significantly from julia or python. The latter two languages offer a logging module as part of their standard library, respectively called **Logging** and **logging**. In julia, the **Logging** module offers a mechanism that is somewhat comparable to how exceptions are handled: programmers can insert logging statements throughout their code and use default or self-written local or global handlers to process and store log messages. Logging handlers are organized in a type hierarchy where the 'root' handler ultimately handles all logging messages that are not taken care of by lower-level loggers. This is similar to how logging is organized in python's **logging** module. One difference is that in python the logging configuration, including logging level and output file (via `logging.basicConfig()`) can be set only once per session.

R has no native logging mechanism, but for process logging several packages are available via CRAN[1]. The two most popular ones by far[2] are currently **futile.logger** (Rowe 2016), and **logging** (Frasca 2019). Other implementations include **logger** (Daróczi 2019), **loggit** (Price 2018), **log4r** (White and Jacobs 2020), and **rsyslog** (Jacobs 2018). Typical features for these packages include the ability to distinguish between different classes of messages, setting a logging level (threshold) that decides which messages are created at runtime, and customizing output messages. Typical message types include 'information', 'warning', 'error', and sometimes 'debug'. When comparing the functionality of these packages, **futile.logger**, **logging**, and **logger** are especially similar as all of them are inspired by a Java logging system called **log4j**[3]. This system is again similar to julia's **Logger** or python's **logging**, with a configurable hierarchy of log handlers. The three R packages mainly differ on details such as the granularity of available logging thresholds (**logging** has the most), available output channels (**logger** offers the most), and look-and-feel. The **loggit** package distinguishes itself by offering dedicated logging of non-standard conditions: it sends error, warning, and message conditions to JSON output as well as passing them through to `stderr`. Finally, the **rsyslog** package is written to resemble an operating system's `syslog` interface. On POSIX complient operating systems, all logging messages are send to the central `syslog` file.

Between the possibilities of interactive variable tracing and process logging during production runs there seems to be a gap in functionality where the state of variables is traced automatically while running in production. Such functionality may serve interesting use cases. For example, consider a frequently running production system that includes elaborate data cleaning, imputation, and transformation steps. It is interesting to monitor the effect that each step has on the variables, both to understand their relative importance within the whole procedure, and to monitor changes in this relative importance over production runs. Significant changes over time may indicate that data circumstances have changed to the extent that assumptions upon which the data processing is developed may need to be reconsidered. At the development stage, such monitoring can help deciding whether the contribution of each processing step is worth the extra complexity and runtime of the whole procedure.

The **lumberjack** package (van der Loo 2020a) presented in this paper aims to fill this gap between interactive tracing and process logging. It allows users to specify which objects should be monitored and how. Users can either follow a (summary of) the state of an object

[1] `https://cran.r-project.org`

[2] Based on download statistics obtained with **dlstats** by Yu (2019).

[3] `https://logging.apache.org/log4j/2.x/`

or measure differences between consecutive versions of an object as it gets processed. For example, one can follow the average of a variable in a data frame that gets processed or count the number of cells that changed after each operation. In the simplest case this can be done by adding just a single line code to an existing R script.

The package is designed with three core design principles in mind. First, a user should not have to worry about data monitoring while developing the main process. Ideally, a user develops a production script and later simply adds a specification stating which variables to monitor and how. This means that the package should *separate concerns* between developing a production script and monitoring data. Second, the monitoring process should neither require any change in user code, nor rely on behaviour of code used from other packages: monitoring must be *agnostic* with respect to the code that actually processes the data. Third, the package must allow users and developers complete *flexibility* in how to track changes in data. Depending on the objects that are followed, many different parameters may be interesting and the package must therefore be extensible with user-defined monitoring capabilities.

The following Section demonstrates how to monitor R objects with **lumberjack**, both in batch and in interactive mode. In Section 3 it is shown how the package can be extended with custom loggers by users or package developers. A conclusion is given in Section 5.

2. Monitoring R objects

In what follows a running example will be used based on the 'supermarkets' data set that is included in the supplementary materials. The data set is derived from the `retailers` data set of the **validate** package (van der Loo and de Jonge 2019).

```
R> head( read.csv("supermarkets.csv"), 3 )
```

```
     id staff turnover other.rev total.rev
1 SPM01    75       NA        NA      1130
2 SPM02     9     1607        NA      1607
3 SPM03    NA     6886       -33      6919
```

Besides an identifying variable in the first column it contains 'staff' numbers, 'turnover', 'other revenue' and 'total revenue' in kEUR of sixty establishments.

2.1. Monitoring changes in production scripts

A script called `supermarkets.R` shown in Figure 1 will serve as example production script. It reads `supermarkets.csv` and then imputes and corrects 'other revenue' values where deemed necessary. Next it uses a ratio estimator to impute 'staff' numbers based on 'turnover' amounts. Finally, it derives a new variable called 'ratio' holding the ratios between 'turnover' and 'total revenue' and then writes the output to a new CSV file. In production circumstances such a file could be run using `source("supermarkets.R")` or as follows while invoking R.

```
  R -q -f supermarkets.R
```

To track all possible changes in the supermarket data, a user assigns one or more *loggers* to existing R objects. Here this is done by adding a single line at the beginning of the script,

```
spm <- read.csv("supermarkets.csv")

# assume empty values should be filled with 0
spm <- transform(spm
        , other.rev = ifelse(is.na(other.rev), 0, other.rev))

# assume that negative amounts have only a sign error
spm <- transform(spm, other.rev = abs(other.rev))

# ratio estimator for staff conditional on turnover
Rhat <- with(spm
        , mean(staff, na.rm = TRUE)/mean(turnover, na.rm = TRUE))

# impute 'staff' variable where possible using ratio estimator
spm <- transform(spm
      , staff = ifelse(is.na(staff), Rhat*turnover, staff))

# add a column
spm <- transform(spm, ratio = turnover/total.rev)

# write output
write.csv(spm, "supermarkets_treated.csv", row.names = FALSE)
```

Figure 1: A script that reads, transforms and writes the `supermarkets` dataset (`supermarkets.R` in the supplementary materials).

just after reading the `supermarkets.csv` file. The function `start_log()` accepts a variable name and a logging object which will be discussed below in more detail.

```
spm <- read.csv("supermarkets.csv")

start_log(spm, cellwise$new(key = "id"))

# the rest of the script as in Figure 1.
```

The altered script is stored as `supermarkets_logged_1.R` in the supplementary materials. Now, from a running R session (interactive or in batch mode) the script must be executed as follows.

```
R> library(lumberjack)
R> out <- run_file("supermarkets_logged_1.R")
```

Alternatively one can run the script when invoking an R session as follows.

```
R -q -e 'library("lumberjack"); run_file("supermarkets_logged_1.R")'
```

The function `run_file()` has executed the script and signals that a log file was written to `smp_cellwise.csv` (the reason that `run_file()` is needed is discussed at the end of this Section).

```
R> spm_log <- read.csv("spm_cellwise.csv")
R> head(spm_log, 3)
```

```
  step                     time                      srcref
1    2 2020-05-08 15:16:12 CEST supermarkets_logged_1.R#7-7
2    2 2020-05-08 15:16:12 CEST supermarkets_logged_1.R#7-7
3    2 2020-05-08 15:16:12 CEST supermarkets_logged_1.R#7-7
                                                               expression
1 spm <- transform(spm, other.rev = ifelse(is.na(other.rev),0,other.rev))
2 spm <- transform(spm, other.rev = ifelse(is.na(other.rev),0,other.rev))
3 spm <- transform(spm, other.rev = ifelse(is.na(other.rev),0,other.rev))
    key  variable old new
1 SPM01 other.rev  NA   0
2 SPM02 other.rev  NA   0
3 SPM06 other.rev  NA   0
```

Reading the log file yields a step count, a time stamp, a source reference, the code that was executed, the key of the record where changes took place, the name of the variable, and the old and the new value. As suggested by the name of the logger (`cellwise`) it records changes cell by cell. For example, in record `SPM06` the value of variable `other.rev` was altered from `NA` to `0` by the `transform` expression shown in the third column.

When a user just adds the single `start_log()` expression, **lumberjack** makes a number of default choices. These include the point where the logging stops (after all the expressions in the R script have been executed) and where the logging information is written. In the case of the `cellwise` logger, both can be controlled by adding a line like

```
  stop_log(spm, file = "my_custom_log.csv")
```

at the point where logging should stop. An overview of logging control functions is given in Table 2. The fact that `stop_log()` accepts a `file` argument actually depends on the fact that `spm` is tracked by the `cellwise` logger: not all loggers necessarily write something to a file. The structure of loggers is discussed in more detail in Section 3 but briefly, the loggers that come with `lumberjack` are `R6` reference objects[4]. This means that the expression

```
  cellwise$new(key = "id")
```

returns a new logger, that uses variable `id` as key variable. Not all loggers need to know about a key and in fact the arguments given to `$new()` depend on the logger. An overview of loggers currently available in **lumberjack** is given in Table 2.

To demonstrate the possibility of multiple tracking, two loggers tracking the `spm` variable are specified so the top of the script in Figure 1 now looks like this.

[4] Based on the `R6` package of Chang (2019)

Table 1: Logging control.

Logger	what it does
`start_log`	Assign a logger to an R object.
`stop_log`	Stop logging and dump log, where dumping can be switched off.
`dump_log`	Dump logging info and stop logging, where stopping is optional
`run_file`	Execute a file, while logging, in a new `environment`.
`source_file`	Execute a file, while logging, in the global `environment`.
`%L>%`	Pipe operator that also triggers logging where indicated.

Table 2: Loggers in **lumberjack**.

Logger	what it does
`expression_logger`	record result of custom R expressions.
`filedump`	dump a file after each operation.
`simple`	record whether anything changed ('`logical`').
`cellwise`	record cell-by-cell changes.

```
  spm <- read.csv("supermarkets.csv")

  start_log(spm, logger = cellwise$new(key="id"))

  logger <- expression_logger$new(
               mean_staff     = mean(staff, na.rm = TRUE)
             , mean_other.rev = mean(other.rev, na.rm = TRUE)
           )
  start_log(spm, logger=logger)
  # the rest of the script...
```

The altered script is provided as `"supermarkets_logged_2.R"` in the supplementary materials. Here, the mean of variables 'staff' and 'other.rev' are tracked as the dataset is manipulated by the script. Running the file now yields two messages, one for each logger.

```
R> run_file("supermarkets_logged_2.R")
```

Below the new log file is read, yielding a complete view on how the means of 'staff' and of 'other revenue' vary as the data gets processed (the 'srcref' and 'expression' columns are suppressed for brevity).

```
R> read.csv("spm_expression.csv")[c("step", "mean_staff", "mean_other.rev")]

  step mean_staff mean_other.rev
1    1   11.53704      22.366792
2    2   11.53704       8.946717
3    3   11.53704      10.046717
4    4   11.53704      10.046717
5    5   12.07457      10.046717
6    6   12.07457      10.046717
7    7   12.07457      10.046717
```

Again, **lumberjack** chooses default places to stop logging and to dump the logging data. The user can control this by inserting `stop_log()` anywhere in the code after logging started. It is possible to stop individual loggers with the `logger` argument. For example, to stop the cellwise logger at a certain point, add the following.

```
stop_log(spm, logger = "cellwise")
```

This will dump the '`cellwise`' log for `spm` and stop using the cellwise logger, but it will continue logging with the expression logger. Note that the combination of a variable name and a logger type is sufficient to uniquely identify a logger instance: it is pointless to track the same object with the same type of logger twice, and this is therefore not allowed by **lumberjack**.

Summarizing, the interface implemented by the package consists of two main parts: an in-script specification of what and how to log, and a special function called `run_file()` to run the script. This implementation is a direct consequence of two of the design principles mentioned in the introduction: *separation of concerns* and being *agnostic*. Indeed, there are only a few ways to implement monitoring. One is to copy the mechanism that is used for process loggers such as **futile.logger**, and require users to insert explicit logging expressions at multiple places within their code. This method violates *separation of concerns* as it heavily mixes data processing code with data monitoring code. Another way is to alter the data processing functions so that they detect whether an object is being monitored, at which point they make sure that monitoring code is executed. This would violate the *agnostic* principle as it implies an explicit relation between data processing code and data monitoring code. The third way is to intercept expressions as they are executed and insert monitoring code at runtime. This is also what the tracing functions in base R do for debugging purposes. In this sense, **lumberjack** mimics the behaviour of base R tracing: it offloads the monitoring interventions to a special 'code runner' that knows what objects are monitored in which way.

2.2. Monitoring data in interactive mode

For logging in interactive R sessions, **lumberjack** defines a special 'pipe' operator, denoted `%L>%`, that can be used to chain expressions together. When used without logging it works similar (but not exactly the same) to the well known **magrittr** pipe operator of Bache and Wickham (2014): output of the left-hand-side is fed as the first argument to the function call on the right-hand-side.

```
R> spm <- read.csv("supermarkets.csv")
R> spm %L>%
+   transform(other.rev = ifelse(is.na(other.rev), 0, other.rev )) %L>%
+   transform(ratio = turnover/total.rev) %L>%
+   head(3)

     id staff turnover other.rev total.rev     ratio
1 SPM01    75       NA         0      1130        NA
2 SPM02     9     1607         0      1607 1.0000000
3 SPM03    NA     6886       -33      6919 0.9952305
```

To record what happens at each expression in the chain, a logger must be inserted and subsequently stopped.

```
R> out <- spm %L>%
+   start_log(cellwise$new(key = "id")) %L>%
+   transform(other.rev = ifelse(is.na(other.rev), 0, other.rev)) %L>%
+   transform(ratio = turnover/total.rev) %L>%
+   stop_log()

Dumped a log at cellwise.csv
```

The name of the default output file is not prepended with the name of the variable being monitored as in Section 2.1. The reason is that `start_log()` can not in all circumstances easily determine the name of the variable under scrutiny. It is also of less importance, when compared to the case presented in Section 2.1, since a chain of operations can only process a single data object.

The log can be retrieved again by reading the log file. Below, the first and last lines of the logging data are shown.

```
R> spm_log <- read.csv("cellwise.csv")
R> rbind(head(spm_log,1), tail(spm_log,1))

   step                     time srcref
1     1 2020-05-08 15:16:12 CEST     NA
91    2 2020-05-08 15:16:12 CEST     NA
                                                     expression   key
1  transform(other.rev = ifelse(is.na(other.rev), 0, other.rev)) SPM01
91                          transform(ratio = turnover/total.rev) SPM60
    variable old          new
1  other.rev  NA 0.0000000000
91     ratio  NA 0.0007087172
```

Here, the logger is created with `cellwise$new()` as usual. The 'pipe' operator fulfills the task of detecting whether data on the left-hand-side is logged. If so, it will store a copy and execute the right-hand-side with data from the left-hand-side as input to create the output. Next, the input stored earlier, the output, and some metadata is fed to the logger so it can measure the difference and finally `%L>%` returns the output. One can think of `%L>%` is a 'dressed' pipe operator that does something extra on top of passing output of one expression as input to another (i.c., making sure that the logging information is created).

3. Custom loggers

The **lumberjack** package allows users and package authors to create custom loggers. In order for the logger to work with **lumberjack** it must meet a few requirements. In short, it must be a reference object with an `$add()` method for adding entries to the log, and a `$dump()` method for dumping log data. In the rest of the Section these requirements are discussed in more detail. It is assumed that the reader is somewhat familiar with object-oriented programming in R.

Any type of reference object based on R environments may work but it is recommended to use the `R6` system of Chang (2019) or the `RefClass` system from the **methods** package (R Core Team 2019). In the current paper `R6` is used but an example using `RefClass` can be found in the 'extending **lumberjack**' vignette that is included with the package.

To create a logger for lumberjack, the new `R6` class must have an `add()` method with the following signature.

```
$add(meta, input, output)
```

The task of this method is to use the `input` and/or the `ouput` data to create logging information and add this to the log. Optionally it can use the information in `meta` to enrich the logging information. `lumberjack` puts no restrictions on the data type of `input` and `output`. It is thus possible to create loggers for any type of data. When data is logged by a custom logger, `lumberjack` will make sure that the first argument (`meta`) is passed a named '`list`' with two elements. Element `meta$expr` is the R '`expression`' that turned `input` into `output`. Element `meta$src` is the same expression represented as a '`character`' string. For example, the `add()` method of the `filedump` logger (Table 2) just increases an internal counter and writes `output` to a numbered file in a directory.

Second, the logger must have a `dump()` method with the following signature.

```
$dump()
```

It is allowed for the dump method to have extra arguments. Extra arguments passed to `stop_log()` will be passed through to the relevant `$dump()` method. For example, the `dump` method of the `cellwise` logger accepts a `file` argument to specify to what file the logging information should be exported.

In Figure 2 the '`trivial`' logger is defined. This logger only registers whether data has changed at all, but it does not register which expresssion caused the change. The final log result is therefore a simple `TRUE` (object has changed) or `FALSE` (object has not changed). Althought this logger is very simple it contains all elements necessary to define a logger.

The class definition contains one variable called `changed` with initial value `NULL`. This is the placeholder for the logging information that will be updated by the `add()` method. The `initialize` method is executed when a new object of class `trivial` is created. At initialization, `changed` is set to `FALSE`.

Now, the `add()` method ignores the `meta` argument and sets `changed` to `TRUE` when it already is `TRUE` or when `input` and `output` are not identical. The `dump()` method writes a message to screen, stating whether data has changed or not.

The code of Figure 2 is stored in a file called `trivial.R` with the supplamentary materials. Here is a demonstration of how to use it.

```
library(R6)
trivial <- R6Class("trivial",
  public = list(
    changed = NULL
  , initialize = function(){
      self$changed <- FALSE
  }
  , add = function(meta, input, output){
    self$changed <- self$changed | !identical(input, output)
  }
  , dump = function(){
    msg <- if(self$changed) "" else "not "
    cat(sprintf("The data has %schanged\n", msg))
  }
  )
)
```

Figure 2: Definition of the '`trivial`' logger using the **R6** system.

```
R> source("trivial.R")
R> spm <- read.csv("supermarkets.csv")
R> out <- spm %L>% start_log(trivial$new()) %L>% identity() %L>% dump_log()

The data has not changed

R> out <- spm %L>% start_log(trivial$new()) %L>% head(10) %L>% dump_log()

The data has changed
```

Here, `identity()` is R's identity function: it just returns it's argument unchanged. `head(10)` returns the first ten records data passed to it by `%L>%`. Observe that the logger correctly notifies the user whether the data has undergone any changes.

For some loggers it may be necessary to perform some cleanup actions when stopping. For example, a logger may need to close a connection to a database or remove temoprary files. For this reason one can optionally add a `stop()` method. If it exists, this is called (currently with no arguments) by `stop_log()` after executing the `dump()` method. A typical logger object using this construction will set up a connection object at initialisation and close the connection when stopped.

4. Implementation

The techniques used to implement functionality of this package have broader use cases then logging, and have also been documented separately in van der Loo (2020b). The main idea is to create a mechanism where one can derive information from running R code, subject to the following conditions. First, a user should not have to extensively edit their code in order to create or configure the way this information is derived (a typical counter-example is process logging where logging messages require developers to insert logging expressions throughout their code). Second, creation or manipulation of global variables, either in R's global namespace or in a package's namespace, e.g. for configuration purposes, should be avoided. And finally, the information, derived from running R code, should be transmitted through ordinary channels and not `stderr`. This means that mechanisms such as (typed) error messages are to be avoided as well. The **lumberjack** package relies on two constructions to achieve this.

The first way these objectives can be achieved is by creating a 'file runner', such as `run_file()` in the **lumberjack** package. This function parses an R script and runs the expressions one by one using R's `parse()` and `eval()`. This offers the possibility to derive information from the state of the user code before and after evaluating each expression. Since the user code is now evaluated in a custum parse-eval loop there is also no need for using exceptions to convey logging information. In order to capture user commands, such as those expressed by `start_log()` or `dump_log()`, these functions are masked by `run_file()`. That is to say, the functions are replaced by the exact same function as the one that the user is calling, except that they also write some output into an R `environment` that is only accessible from within `run_file()`. Hence, the use of a global state for configuring which variables are traced and how to trace them, is avoided. This information is only stored within the scope of `run_file()`. Furthermore, the masking of the user-facing functions only takes place while `run_file()` is doing its work, so again no changes the global environment are required.

The second way in which separation between logging and user code is achievied is through the `%L>%` operator. In this case there is no masking or custom parse and evaluation function. The idea here is that the logging object travels with the data that is tracked. The function `start_log()` returns its argument with a new logger attached as an attribute. The `%L>%` operator detects whether loggers are present. If so, a copy of the left-hand-side is stored. Next, the expression on the right-hand-side is evaluated with the left-hand-side approprately substituted. The output of this evaluation, together with the input and some metadata are fed to the attached loggers. If evaluation of the expression resulted in the removal of one or more loggers, these are reattached by `%L>%`, after which the resulting data is returned.

5. Conclusion

The `lumberjack` package allows users to monitor changes in data with minimal coding effort, both in interactive and production (batch) circumstances. Monitoring is specified by assigning a logger to an R object, thereby separating concerns between creating data processing code and data monitoring code. It is possible to track multiple R objects simultaneously and to track an R object with multiple loggers. The tracking itself is agnostic of the code used to manipulate the objects under scrutiny and can be used in combination with any (third party) R code. The way tracking takes place is flexible since it can be fully customized by creating a logging object type satisfying a small set of interface requirements.

Acknowledgements

The author is indebted by Dr. K. Olav ten Bosch for carefully reading the original manuscript.

References

Bache SM, Wickham H (2014). *magrittr: A Forward-Pipe Operator for R*. R package version 1.5, URL https://CRAN.R-project.org/package=magrittr.

Bezanson J, Edelman A, Karpinski S, Shah VB (2017). "Julia: A fresh approach to numerical computing." *SIAM review*, **59**(1), 65–98. URL https://doi.org/10.1137/141000671.

Chang W (2019). *R6: Encapsulated Classes with Reference Semantics*. R package version 2.4.0, URL https://CRAN.R-project.org/package=R6.

Daróczi G (2019). *logger: A Lightweight, Modern and Flexible Logging Utility*. R package version 0.1, URL https://CRAN.R-project.org/package=logger.

Frasca M (2019). *logging: R Logging Package*. R package version 0.10-108, URL https://CRAN.R-project.org/package=logging.

Jacobs A (2018). *rsyslog: Interface to the 'syslog' System Logger*. R package version 1.0.1, URL https://CRAN.R-project.org/package=rsyslog.

Price R (2018). *loggit: Effortless Exception Logging*. R package version 1.1.1, URL https://CRAN.R-project.org/package=loggit.

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rowe BLY (2016). *futile.logger: A Logging Utility for R*. R package version 1.4.3, URL https://CRAN.R-project.org/package=futile.logger.

van der Loo M (2020a). *lumberjack: Track Changes in Data*. R package version 1.1.3, URL https://CRAN.R-project.org/package=lumberjack.

van der Loo MPJ (2020b). "A method for deriving information from running R scripts." *The R Journal*. Accepted for publication, URL https://arxiv.org/abs/2002.07472.

van der Loo MPJ, de Jonge E (2019). "Data Validation Infrastructure for R." *Journal of Statistical Software*. Accepted for Publication, URL https://arxiv.org/abs/1912.09759.

van Rossum G, Drake FL (2009). *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA. ISBN 1441412697.

White JM, Jacobs A (2020). *log4r: A Fast and Lightweight Logging System for R, Based on 'log4j'*. R package version 0.3.2, URL https://CRAN.R-project.org/package=log4r.

Yu G (2019). *dlstats: Download Stats of R Packages*. R package version 0.1.3, URL https://CRAN.R-project.org/package=dlstats.

Affiliation:

Mark P.J. van der Loo
https://orcid.org/0000-0002-9807-4686

Research and Development
Statistics Netherlands
Henri Faasdreef 312
2492JP Den Haag, The Netherlands
E-mail: m.vanderloo@cbs.nl