Testing and debugging

Tools for Reproducible Research

Karl Broman

Biostatistics & Medical Informatics, UW–Madison

kbroman.org
github.com/kbroman
@kwbroman
Course web: kbroman.org/Tools4RR

We spend a lot of time debugging. We'd spend a lot less time if we tested our code properly.

We want to get the right answers. We can't be sure that we've done so without testing our code.

Set up a formal testing system, so that you can be confident in your code, and so that problems are identified and corrected early.

Even with a careful testing system, you'll still spend time debugging. Debugging can be frustrating, but the right tools and skills can speed the process.

"I tried it, and it worked."

This is about the limit of most programmers' testing efforts.

But: Does it still work? Can you reproduce what you did? With what variety of inputs did you try?

"It's not that we don't test our code, it's that we don't store our tests so they can be re-run automatically."

– Hadley Wickham

R Journal 3(1):5–10, 2011

This is from Hadley's paper about his `testthat` package.

Types of tests

- Check inputs
  - Stop if the inputs aren't as expected.
- Unit tests
  - For each small function: does it give the right results in specific cases?
- Integration tests
  - Check that larger multi-function tasks are working.
- Regression tests
  - Compare output to saved results, to check that things that worked continue working.

Your first line of defense should be to include checks of the inputs to a function: If they don't meet your specifications, you should issue an error or warning. But that's not really testing.

Your main effort should focus on unit tests. For each small function (and your code should be organized as a series of small functions), write small tests to check that the function gives the correct output in specific cases.

In addition, create larger integration tests to check that larger features are working. It's best to construct these as regression tests: compare the output to some saved version (e.g. by printing the results and comparing files). This way, if some change you've made leads to a change in the results, you'll notice it automatically and immediately.

Check inputs

```
winsorize <-
function(x, q=0.006)
{
  if(!is.numeric(x)) stop("x should be numeric")

  if(!is.numeric(q)) stop("q should be numeric")
  if(length(q) > 1) {
    q <- q[1]
    warning("length(q) > 1; using q[1]")
  }
  if(q < 0 || q > 1) stop("q should be in [0,1]")

  lohi <- quantile(x, c(q, 1-q), na.rm=TRUE)
  if(diff(lohi) < 0) lohi <- rev(lohi)

  x[!is.na(x) & x < lohi[1]] <- lohi[1]
  x[!is.na(x) & x > lohi[2]] <- lohi[2]
  x
}
```


The `winsorize` function in my R/broman package hadn't included any checks that the inputs were okay.

The simplest thing to do is to include some `if` statements with calls to `stop` or `warning`.

The input `x` is supposed to be a numeric vector, and `q` is supposed to be a single number between 0 and 1.

Check inputs

```
winsorize <-
function(x, q=0.006)
{
  stopifnot(is.numeric(x))
  stopifnot(is.numeric(q), length(q)==1, q>=0, q<=1)

  lohi <- quantile(x, c(q, 1-q), na.rm=TRUE)
  if(diff(lohi) < 0) lohi <- rev(lohi)

  x[!is.na(x) & x < lohi[1]] <- lohi[1]
  x[!is.na(x) & x > lohi[2]] <- lohi[2]
  x
}
```


The `stopifnot` function makes this a bit easier.

`assertthat` package

```
#' import assertthat
winsorize <-
function(x, q=0.006)
{
  if(all(is.na(x)) || is.null(x)) return(x)

  assert_that(is.numeric(x))
  assert_that(is.number(q), q>=0, q<=1)

  lohi <- quantile(x, c(q, 1-q), na.rm=TRUE)
  if(diff(lohi) < 0) lohi <- rev(lohi)

  x[!is.na(x) & x < lohi[1]] <- lohi[1]
  x[!is.na(x) & x > lohi[2]] <- lohi[2]
  x
}
```


Hadley Wickham's `assertthat` package adds some functions that simplify some of this.

How is the `assertthat` package used in practice? Look at packages which depend on it, such as `dplyr`. Download the source for `dplyr` and try `grep assert_that dplyr/R/*` and you'll see a bunch of examples if its use.

Also try `grep stopifnot dplyr/R/*` and you'll see that both are being used.

Tests in R packages

- Examples in `.Rd` files
- Vignettes
- `tests/` directory
  - `some_test.R` and `some_test.Rout.save`

`R CMD check` is your friend.

The examples and vignettes should focus on helping users to understand how to use the code. But since they get run whenever `R CMD check` is run, they also serve as crude tests that the package is working. But note that this is only testing for things that halt the code; it's not checking that the code is giving the right answers.

Things that you put in the `tests` directory can be more rigorous: these are basically regression tests. The R output (in the file `some_test.Rout`) will be compared to a saved version, if available.

If you want your package on CRAN, you'll need all of these tests and examples to be fast, as CRAN runs `R CMD check` on every package every day on multiple systems.

The `R CMD check` system is another important reason for assembling R code as a package.

An example example

```
#' @examples
#' x <- sample(c(1:10, rep(NA, 10), 21:30))
#' winsorize(x, 0.2)
```


This example doesn't provide a proper test. You'll get a note if it gives an error, but you don't get any indication about whether it's giving the right answer.

A `tests/` example

```
library(qtl)

# read data
csv <- read.cross("csv", "", "listeria.csv")

# write
write.cross(csv, "csv", filestem="junk")

# read back in
csv2 <- read.cross("csv", "", "junk.csv",
                   genotypes=c("AA", "AB", "BB",
                               "not BB", "not AA"))

# check for a change
comparecrosses(csv, csv2)

unlink("junk.csv")
```


An advantage of the `tests/` subdirectory is that you can more easily test input/output.

A useful technique here: if you have a pair of functions that are the inverse of each other (e.g., write and read), check that if you apply one and then the other, you get back to the original.

Note that `unlink("junk.csv")` deletes the file.

`testthat` package

- Expectations

```
expect_equal(10, 10 + 1e-7)
expect_identical(10, 10)
expect_equivalent(c("one"=1), 1)
expect_warning(log(-1))
expect_error(1 + "a")
```

- Tests

```
test_that("winsorize small vectors", { ... })
```

- Contexts

```
context("Group of related tests")
```

- Store tests in `tests/testthat`
- `tests/testthat.R` file containing

```
library(testthat)
test_check("mypkg")
```


The `testthat` package simplifies unit testing of code in R packages.

There are a bunch of functions for defining "expectations." Basically, for testing whether something worked as expected. (It can be good to check that something gives an error when it's supposed to give an error.)

You then define a set of tests, with a character string to explain where the problem is, if there is a problem.

You can group tests into "contexts." When the tests run, that character string will be printed, so you can see what part of the code is being tested.

Put your tests in `.R` files within `tests/testthat`. Put another file within `tests/` that will ensure that these tests are run when you do `R CMD check`.

Example `testthat` test

```
context("winsorize")

test_that("winsorize works for small vectors", {

  x <-       c(2, 3, 7, 9, 6, NA, 5, 8, NA, 0, 4, 1, 10)
  result1 <- c(2, 3, 7, 9, 6, NA, 5, 8, NA, 1, 4, 1,  9)
  result2 <- c(2, 3, 7, 8, 6, NA, 5, 8, NA, 2, 4, 2,  8)

  expect_identical(winsorize(x, 0.1), result1)
  expect_identical(winsorize(x, 0.2), result2)

})
```


These are the sort of tests you might do with the testthat package. The value of this: finally, we are checking whether the code is giving the right answer!

Ideally, you include tests like this for every function you write.

It's not really clear that the second test here is needed. If the first test is successful, what's the chance that the second will fail?

Example `testthat` test

```
test_that("winsorize works for a long vector", {

  set.seed(94745689)
  n <- 1000
  nmis <- 10
  p <- 0.05
  input <- rnorm(n)
  input[sample(1:n, nmis)] <- NA
  quL <- quantile(input, p, na.rm=TRUE)
  quH <- quantile(input, 1-p, na.rm=TRUE)

  result <- winsorize(input, p)
  middle <- !is.na(input) & input >= quL & input <= quH
  low <- !is.na(input) & input <= quL
  high <- !is.na(input) & input >= quH

  expect_identical(is.na(input), is.na(result))
  expect_identical(input[middle], result[middle])
  expect_true( all(result[low] == quL) )
  expect_true( all(result[high] == quH) )

})
```


Here's a bigger, more interesting test.

The code to test a function will generally be longer than the function itself.

Workflow

- Write tests as you're coding.
- Run `test()`
  with `devtools`, and working in your package directory
- Consider `auto_test("R", "tests")`
  automatically runs tests when any file changes
- Periodically run `R CMD check`
  also `R CMD check --as-cran`

Read Hadley's paper about `testthat`. It's pretty easy to incorporate testing into your development workflow.

It's really important to write the tests as you're coding. You're will check to see if the code works; save the test code as a formal test.

What to test?

- You can't test everything.
- Focus on the boundaries
  - (Depends on the nature of the problem)
  - Vectors of length 0 or 1
  - Things exactly matching
  - Things with no matches
- Test handling of missing data.

  `NA`, `Inf`, `-Inf`
- Automate the construction of test cases
  - Create a table of inputs and expected outputs
  - Run through the values in the table

You want your code to produce the correct output for any input, but you can't test all possible inputs, and you don't really need to.

This is an experimental design question: what is the minimal set of inputs to verify that the code is correct, with little uncertainty?

We generally focus on boundary cases, as those tend to be the places where things go wrong.

The mishandling of different kinds of missing data is also a common source of problems and so deserving of special tests.

Another example

```
test_that("running mean with constant x or position", {

  n <- 100
  x <- rnorm(n)
  pos <- rep(0, n)

  expect_equal( runningmean(pos, x, window=1), rep(mean(x), n) )
  expect_equal( runningmean(pos, x, window=1, what="median"),
                rep(median(x), n) )
  expect_equal( runningmean(pos, x, window=1, what="sd"),
                rep(sd(x), n) )

  x <- rep(0, n)
  pos <- runif(n, 0, 5)

  expect_equal( runningmean(pos, x, window=1), x)
  expect_equal( runningmean(pos, x, window=1, what="median"), x)
  expect_equal( runningmean(pos, x, window=5, what="sd"),
                rep(0, n))
})
```


Here's another example of unit tests, for a function calculating a running mean.

Writing these tests revealed a bug in the code: with constant `x`'s, the code should give SD $= 0$, but it was giving `NaN`'s due to round-off error that led to $\sqrt{\epsilon}$ for $\epsilon < 0$.

This situation can come up in practice, and this is exactly the sort of boundary case where problems tend to arise.

Debugging tools

- `cat`, `print`
- `traceback`, `browser`, `debug`
- RStudio breakpoints
- Eclipse/StatET
- gdb

I'm going to say just a little bit about debugging.

I still tend to just insert `cat` or `print` statements to isolate a problem.

R does include a number of debugging tools, and RStudio has made these even easier to use.

Eclipse/StatET is another development environment for R; it seems hard to set up.

The GNU project debugger (gdb) is useful for compiled code.

Debugging

Step 1: Reproduce the problem

Step 2: Turn it into a test

Try to create the minimal example the produces the problem. This helps both for refining your understanding of the problem and for speed in testing.

Once you've created a minimal example that produces the problem, add that to your battery of automated tests! The problem may suggest related tests to also add.

Debugging

Isolate the problem: where do things go bad?

The most common strategy I use in debugging involves a sort of divide-and-conquer: if R is crashing, figure out exactly where. If data are getting corrupted, step back to find out where it's gone from good to bad.

Debugging

Don't make the same mistake twice.

If you figure out some mistake you've made, search for all other possible instances of that mistake.

The most pernicious bugs

The code is right, but your thinking is wrong.

You were mistaken about what the code would do.

→ Write trivial programs to test your understanding.

A number of times I've combed through code looking for the problem, but there really wasn't a problem in the code: the problem was just that my understanding of the algorithm was wrong.

Or I was mistaken about some basic aspect of R.

It can be useful to write small tests, to check your understanding.

For an EM algorithm, always evaluate the likelihood function. It should be non-decreasing.

Summary

- If you don't test your code, how do you know it works?
- If you test your code, save and automate those tests.
- Check the input to each function.
- Write unit tests for each function.
- Write some larger regression tests.
- Turn bugs into tests.

Testing isn't fun. Like much of this course: somewhat painful initial investment, with great potential pay-off in the long run.