Testing statistical software I: units

Ben Klemens

Your trust in anything is an emotional state, not something that can be proven by some routine. There is a literature on proving that a given piece of code does what it claims, but that literature works at a much lower level than we do here in the world of statistical calculation, and tends to avoid floating-point math anyway. Even the best proof of code correctness doesn't check that the input to the code is in the right format, that NaNs and missing data are handled correctly, or that theorem assumptions that you were supposed to check got checked.

So, what makes us feel good about code? The next few entries will cover that question. The first will cover some basic mechanics, the second will cover brainstorming ideas for testing numeric results, and the third will cover some additional details like tolerance and regression.

For a simple statistical problem, you read in the data, run the regression, and read the output. A lot of the people who do this sort of thing all day insist that they aren't really writing code, thus making some sort of distinction between calling functions to do math and what the package authors do—which, of course, consists of calling fucntions to do math. Good habits that the package authors developed are directly applicable to load data/run regression/read output code.

Unit testing is the norm in the programming world, but not the statistical world. The idea is to break your work down into the smallest parts possible, then write a test for each part. Every added test will raise confidence in your overall results, because every step is under control.

We already have some units: load data, run regression, check output. Typically, most testing for these units happens by eyeballing. Are the ages all under 100, and the sexes all either 0/1 or "M"/"F"? This is typically verified by printing the first dozen lines to the screen and looking at it, but why not automate this? [We've written an R package for survey processing, Tea[1], that checks and corrects survey data, but it's probably overkill for most needs.]

Every platform has some variant of the `assert` function/macro, which checks whether a claim is true, and prints an error or halts if the claim fails. In R, you could write

```
stopifnot(max(age) < 100 && min(age) > 0)
```

If you have a few of these, you might as well put them into a function. Where before you may have had

```
frame <- read.csv("indata.csv")
```

now you have

```
get_data <- function(filename){
    frame <- read.csv(filename)
    stopifnot(!is.null(frame))
    stopifnot(max(frame,"age"]) < 100 && min(frame,"age"]) > 0)
    sexes <- unique(frame,"sex"])
```

[1] `http://github.com/rodri363/tea`

```
    stopifnot(length(sexes) < 3 && sexes[0] %in% c("M", "F") && sexes[1] %in% c("M", "F"))
}

frame <- get_data("indata.csv")
```

I don't know about you, but I already feel better about the second form, becaue I know reading in data is one of the most failure-prone parts of the process. People do stupid things like save the wrong data to the right file name, or misread the units, or get delimiters wrong and read in sex = `100|M|East Lansing`. The `get_data` function barks when any of these common sort of snafus happen.

But, you argue, *my statistical calculation will only be run once, so why go through all this effort?*. Except even a one-shot calculation is verified, by yourself in a month, by colleagues, by the Platonic ideal of the referee. Next week, when you get an updated data set, all that eyeballing you did before happens automatically. Next month, when you start a new project on similar data, you can cut/paste a lot of your assertions from one project to the next.

In 21st Century C[2], I spend five pages discussing what makes a good assert macro [I'm having trouble putting together a link to Google Books[3], but search for "example 2-5".] Here's what I wound up with for C[4], including a logging option, an option to continue or halt the program (very useful when debugging), or run any arbitrary code on error (`return -1` is common, but `error_action` can also free pointers, `goto` a cleanup segment, do nothing and just print the error message, &c.).

I reversed the sense of the standard `assert` and R's `stopifnot` to a `Stopif`, because the first two are a sort of double negative: if this claim is not true, then do not continue. From my own experience and anecdotal comments from colleagues, it's clearer to cut one negative: if this happens, do not continue.

All that said, adding assertions of things that seem obvious is a fast way to

- document your expectations for the data and results at this point, to yourself and other readers of your routine.
- clarify your own thinking. What do you expect that the regression results would look like, and what would indicate something so out of bounds that something is really broken? If you can give a thought-out answer those questions, then you can codify them in assertions.
- Cut time debugging when your source sends the revised data set and all the field descriptions change or you start getting ages with values like "M" and "F".
- Feel better about the validity of your code.

So even if you are the sort of person who only writes code to read in a data set and run canned procedures (would $\beta_1 = 1000$ be valid?), there is already room to include testing.

There are debates about whether to ship code with active assertions to users, but for most statistical software the question is moot because the end-user is you, the author. In this case, there is no downside to having lots of assertions, because you wrote assertions to cover those things that you the author would want to know about if they don't hold.

Next time I'll get to unit testing mathematical results that are not as easy to eyeball.

[2] `http://tinyurl.com/C-for-moderns`

[3] `http://books.google.com/books?id=_EdbrocXX9MC&printsec=frontcover`

[4] `https://github.com/b-k/21st-Century-Examples/blob/master/stopif.h`