Statistical Computing (36-350)

Lecture 9: Testing

Cosma Shalizi and Vincent Vu

28 September 2011

Agenda

- Why test?
- Testing answers vs. cross-checking
- Software testing vs. hypothesis testing
- Combining testing and programming

ABSOLUTELY ESSENTIAL READING FOR FRIDAY: Section 4.5 of the textbook

MERELY USEFUL READING: Chapter 3

Why Test Your Program?

Your code implements a method for solving a problem
You would like the solution to be correct
How do you know that you can trust it?
Answer: you test for correctness
Test both the whole program ("functional" tests) and components ("unit" tests)
distinction blurs for us

Procedure vs. Substance

Do we get *the right answer* (substance)
vs.
Do we get an answer *in the right way* (procedure)?
These go back and forth with each other:
we trust the procedure because it gives the right answer
we trust the answer because it came from a good procedure

This only *seems* like a vicious circle

We have a procedure, so we check substance

also: respect the interface

Testing for particular cases

Test cases with known answers

```
a <- runif(1)
add(2,3) == 5
add(a,0) == a
add(a,-a) == 0
cor(c(1,-1,1,1),c(-1,1,-1,1)) = -1/sqrt(3)
```

Testing by cross-checking

Compare alternate routes to the same answer

```
a <- runif(n=3,min=-10,max=10)
add(a[1],a[2]) == add(a[2],a[1])
add(add(a[1],a[2]),a[3]) == add(a[1],add(a[2],a[3]))
add(a[3]*a[1],a[3]*a[2]) == a[3]*add(a[1],a[2])
x <- runif(10,-10,10)
f <- function(x) {x^2*exp(-x^2)}
g <- function(x) {2*x*exp(-x^2) -2* x^3*exp(-x^2)}
isTRUE(all.equal(derivative(f,x), g(x)))
```

If this seems too unstatistical...

```
x <- runif(10)
a <- runif(1)
cor(x,x) == 1
cor(x,-x) == -1
cor(x,a*x) == 1
dnorm(x,mean,sd) == dnorm((x-mean)/sd,0,1)
pnorm(qnorm(p)) == p
qnorm(pnorm(x)) == x
```

of course with finite precision we don't really want to insist that these be exact! (look at the example with `all.equal`)

Software Testings vs. Hypothesis Testing

Statistical hypothesis testing: risk of false alarm (size) vs. probability of detection (power)

(type I vs. type II errors)

Software tests: no false alarms allowed (false alarm rate $= 0$)

Has to reduce power to detect errors

so code can pass all our tests and still be wrong

but we can direct the power to detect certain errors

including where the error lies

Combining Testing and Coding

- Variety of tests $\Leftrightarrow$ more power to detect errors $\Rightarrow$ more confidence when tests are passed
- $\therefore$ For each function, build a battery of tests
  Step through the tests, record which failed
- Make it easy to add tests
  Make it easy to run tests
  $\therefore$ Bundle tests together into a function, which tests another function

Testing Considerations

Tests should only involve the interface, not the internal implementation (substance, not procedure)

Tests should control inputs; may require using dummy input generators:

```
foo <- function(x,y) {
  z <- bar(x); return(baz(y,z))
}

bar <- function(x) {
  # stuff involving x
}

test.foo <- function() {
   bar <- function(x) {
     # generate a plausible value for bar(), independent of x
   }
   return(foo(121,"philomena") == "genevieve")
}
```

(cf. homework 4)

The Cycle

After making changes to a function, re-run its tests
(and those of upstream functions)
If anything's (still) broken, fix it
If not, go on your way
When you meet a new error, write a new test
Make sure tests only involve the interface

A Ratchet

When we have a version of the code which we are confident gets some cases right, keep it around (under a separate name)

Now compare new versions to the old, on those cases

Keep debugging until the new version is at least as good as the old

Test-Driven Development

Start: an idea about what the program should do
Idea is vague and unhelpful
Make it clear and useful by writing tests for success
Tests come *first*, then the program
Modify code until it passes all the tests
When you find a new error, write a new test
When you add a new capacity, write a new test
When you change your mind about the goal, change the tests
By the end, the tests specify what the program should do, and the program does it

Awkward Cases

Boundary cases, "at the edge" of something, or non-standard inputs
What should these be?

```
add(x,NA)     # NA, presumably
add("a","b") # NA, or error message?
divide(10,0) # Inf, presumably
divide(0,0)  # NA?
var(1)       # NA? error?
cor(c(1,-1,1,-1),c(-1,1,NA,1))   # NA? -1? -1 with a warning?
cor(c(1,-1,1,-1),c(-1,1,"z",1))  # NA? -1? -1 with a warning?
cor(c(1,-1),c(-1,1,-1,1))        # NA? 0? -1?
```

Pinning down awkward cases helps specify function

Pitfalls

- Writing tests takes time
- Running tests takes time
- Tests have to be debugged themselves
- Tests can provide a false sense of security

Advanced Tool: `RUnit`

Writing many tests for many functions is very repetitive
Repetitive tasks should be automated through functions
The `RUnit` package on CRAN gives tools and functions to simplify writing unit tests
Useful but optional; read the "Vignette" first, before the manual or documentation

Summary

- Trusting software means testing it for correctness, both of substance and of procedure
- Software testing is an extreme form of hypothesis testing: no false positives allowed, so any power to detect errors has to be very focused
- $\therefore$ Write and use lots of tests; add to them as we find new errors
- Cycle between writing code and testing it

Next time: functions as objects