Statistical Computing (36-350)
Lecture 7: Testing

Cosma Shalizi

18 September 2013
Agenda

- Why test?
- Testing answers vs. cross-checking
- Software testing vs. hypothesis testing
- Combining testing and programming
Your code implements a method for solving a problem
You would like the solution to be correct
Question: How do you know that you can trust it?
Why Test Your Program?

Your code implements a method for solving a problem
You would like the solution to be correct
Question: 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)
Your code implements a method for solving a problem
You would like the solution to be correct
Question: 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
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
Programming means *making* a procedure, so we check substance
also: respect the interface
Testing for particular cases

Test cases with known answers

\[ a \leftarrow \text{runif}(1) \]
\[ \text{add}(2,3) == 5 \]
\[ \text{add}(a,0) == a \]
\[ \text{add}(a,-a) == 0 \]
\[ \text{cor}(c(1,-1,1,1),c(-1,1,-1,1)) = -1/\sqrt{3} \]
Testing by cross-checking

Compare alternate routes to the same answer

```r
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...

```r
x <- runif(10)
a <- runif(1)
cor(x,x) == 1
cor(x,-x) == -1
cor(x,a*x) == 1
all(pnorm(0,mean=0,sd=x) == 0.5)
pnorm(x,mean,sd) == pnorm((x-mean)/sd,0,1)
all(pnorm(x,0,1) == 1-pnorm(-x,0,1))
pnorm(qnorm(p)) == p
qnorm(pnorm(x)) == x
```
If this seems too unstatistical...

\[
x \leftarrow \text{runif}(10) \\
a \leftarrow \text{runif}(1) \\
\text{cor}(x,x) == 1 \\
\text{cor}(x,-x) == -1 \\
\text{cor}(x,a*x) == 1 \\
\text{all} (\text{pnorm}(0,\text{mean}=0,\text{sd}=x) == 0.5) \\
\text{pnorm}(x,\text{mean},\text{sd}) == \text{pnorm}((x-\text{mean})/\text{sd},0,1) \\
\text{all} (\text{pnorm}(x,0,1) == 1-\text{pnorm}(-x,0,1)) \\
\text{pnorm} (\text{qnorm}(p)) == p \\
\text{qnorm} (\text{pnorm}(x)) == x
\]

With finite precision we don’t really want to insist that these be exact! (look at the example earlier with all.equal)
Statistical hypothesis testing: risk of false alarm (size) vs. probability of detection (power)

(type I vs. type II errors)
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)
Must reduce power to detect errors
∴ code can pass all our tests and still be wrong
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)
Must reduce power to detect errors
∴ 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 (if we test small pieces)
Variety of tests ⇔ more power to detect errors ⇒ more confidence when tests are passed

∴ 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
∴ 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 stubs/dummy input generators:

```r
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")
}
```
The Cycle

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

Software engineers sometimes call this “regression testing”, but they don’t mean statistical regressions.
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
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
- There are costs to knowing about problems (people get upset, responsibility to fix things, etc.)
Pitfalls

- Writing tests takes time
Pitfalls

- Writing tests takes time
- Running tests takes time
Pitfalls

- Writing tests takes time
- Running tests takes time
- Tests have to be debugged themselves
Pitfalls

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

There are costs to knowing about problems (people get upset, responsibility to fix things, etc.)
Pitfalls

- Writing tests takes time
- Running tests takes time
- Tests have to be debugged themselves
- Tests can provide a false sense of security
- There are costs to knowing about problems (people get upset, responsibility to fix things, etc.)
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: debugging