Lecture 11: Testing and Design

Statistical Computing, 36-350

Wednesday October 21, 2015

Last time: basic debugging

Basic debugging tricks:

- Notifications and alerts that you can add
- Localizing issues and changing input parameters
- Error handling in code

Today: testing and design

Better success through design!

- Some testing principles
- Top-down design of programs
- Re-factoring existing code to a better design
- Example: Jackknife

Procedure versus substance

Our two goals:

- Do we get *the right answer* (substance)?
- Do we get an answer *in the right way* (procedure)?

Both are important (don't forget about the second!)

Hypothesis testing versus software testing

Statistical hypothesis testing: risk of false alarm (size) and probability of detection (power). This balances type I and type II errors

Software testing: no false alarms allowed. This is going to reduce our 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

- So write and use lots of tests
- And cycle between writing code and testing it

Cross-checking answers

When we have a version of the code which we are confident gets some cases right, we should 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
- We'll see an example of this shortly

Pre-design: abstraction

- Abstraction means hiding details and specifics, dealing in generalities and common patterns
- We have talked about lots of examples of this already (variables, data structures, functions)
- The point of abstraction: program in ways which don't use people as bad computers
- Economics says: rely on *comparative* advantage
  - Computers: good at tracking arbitrary details, applying rigid rules
  - People: good at thinking, meaning, discovering patterns
- So we should spend our time on the big picture, organize our programs with this in mind

Top-down design

1. Start with the big-picture view of the problem
2. Break the problem into a few big parts
3. Figure out how to fit the parts together
4. Repeat this for each part

The big-picture view

Step 1: thnk about the big picture

- Resources: what information is available as part of the problem?
  - Usually arguments to a function
- Requirements: what information do we want as part of the solution?
  - Usually return values
- What do we have to do to transform the problem statement into a solution?

Breaking into parts

Step 2: divide-and-conquer mentality

- Try to break the calculation into a *few* parts (say, 5 or less)
  - Bad: write 500 lines of code, chop it into five blocks of 100 lines
  - Good: each part is an independent calculation, using separate data

- Advantages of the good way:
  - More comprehensible to human beings
  - Easier to improve and extend (respect interfaces)
  - Easier to debug and test

Put the parts together

Step 3: assuming you've written each part, how would you put them together?

- Write top-level code for the function which puts those steps together:

```
# Not actual code
big.job = function(lots.of.arguments) {
  intermediate.result = first.step(some.of.the.args)
  final.result = second.step(intermediate.result,rest.of.the.args)
  return(final.result)
}
```

- Note: you can actually go and do this very early on! The sub-functions don't actually have to be written at this point (only when you run the code eventually)

What about those sub-functions?

Step 4+: you don't actually have a working program yet, but you have a good setup

- Recursion: because each sub-function solves a single well-defined problem, we can solve it by top-down design
- The step one level up tells you what the arguments are, and what the return value must be
- The step one level up doesn't care how you turn inputs to output
- Stop when we hit a sub-problem we can solve in a few steps with *built-in* functions

Thinking algorithmically

- Top-down design only works if you understand
  - the problem, and
  - a systematic method for solving the problem
- Therefore it forces you to think **algorithmically**
- First guesses about how to break down the problem are often wrong
  - but functional approach contains effects of changes
  - so don't be afraid to change the design

Refactoring

Even if we didn't start in top-down design mode, once we have some code, and it (more or less) works, re-write it to emphasize commonalities:

- Parallel and transparent naming
- Grouping related values into objects
- Common or parallel sub-tasks become shared functions
- Common or parallel over-all tasks become general functions

Re-factoring tends to make code look more like the result of top-down design. *This is no accident!*

Extended example: the jackknife

- Have an estimator $\hat{\theta}$ of parameter θ. Want the standard error of our estimate, $se_{\hat{\theta}}$
- The jackknife approximation:
  - omit case i, get estimate $\hat{\theta}_{(-i)}$
- Take the variance of all the $\hat{\theta}_{(-i)}$, multiply by $(n-1)^2/n$
- Then $se_{\hat{\theta}}$ is given by square root of this

Jackknife for the mean

```
mean.jackknife = function(vec) {
  n = length(vec)
  jackknife.ests = vector(length=n)
  for (i in 1:n) {
    jackknife.ests[i] = mean(vec[-i])
  }
  variance.of.ests = var(jackknife.ests)
  jackknife.var = ((n-1)^2/n)*variance.of.ests
  jackknife.stderr = sqrt(jackknife.var)
  return(jackknife.stderr)
}
```

```
some.normals = rnorm(100,mean=7,sd=5)
mean(some.normals)
```

```
## [1] 6.560279
```

```
se.formula = sd(some.normals)/sqrt(length(some.normals))
se.jackknife = mean.jackknife(some.normals)
max(abs(se.formula-se.jackknife))
```

```
## [1] 3.330669e-16
```

Gamma parameters

Recall our friend the method of moments estimator:

```
gamma.est = function(the.data) {
  m = mean(the.data)
  v = var(the.data)
  a = m^2/v
  s = v/m
  return(c(a=a,s=s))
}
```

Jackknife for gamma parameters

```
gamma.jackknife = function(vec) {
  n = length(vec)
  jackknife.ests = matrix(NA,nrow=2,ncol=n)
  rownames(jackknife.ests) = c("a","s")
  for (i in 1:n) {
    fit = gamma.est(vec[-i])
    jackknife.ests["a",i] = fit["a"]
    jackknife.ests["s",i] = fit["s"]
  }
  variance.of.ests = apply(jackknife.ests,1,var)
  jackknife.vars = ((n-1)^2/n)*variance.of.ests
  jackknife.stderrs = sqrt(jackknife.vars)
  return(jackknife.stderrs)
}
```

```
data("cats",package="MASS")
gamma.est(cats$Hwt)
```

```
##          a          s
## 19.0653121  0.5575862
```

```
gamma.jackknife(cats$Hwt)
```

```
##          a          s
## 2.74062279 0.07829436
```

Jackknife for linear regression coefficients

```
jackknife.lm = function(df,formula,p) {
  n = nrow(df)
  jackknife.ests = matrix(0,nrow=p,ncol=n)
  for (i in 1:n) {
    new.coefs = lm(as.formula(formula),data=df[-i,])$coefficients
    jackknife.ests[,i] = new.coefs
  }
  variance.of.ests = apply(jackknife.ests,1,var)
  jackknife.var = ((n-1)^2/n)*variance.of.ests
  jackknife.stderr = sqrt(jackknife.var)
  return(jackknife.stderr)
}
```

```
cats.lm = lm(Hwt~Bwt,data=cats)
coefficients(cats.lm)
```

```
## (Intercept)         Bwt
##  -0.3566624   4.0340627
```

```
# "Official" standard errors
sqrt(diag(vcov(cats.lm)))
```

```
## (Intercept)         Bwt
##   0.6922770   0.2502615
```

```
jackknife.lm(df=cats,formula="Hwt~Bwt",p=2)
```

```
## [1] 0.8314142 0.3166847
```

Refactoring the jackknife

- Omitting one point or row is a common sub-task
- The general pattern:

```
figure out the size of the data
for each case
   repeat some estimation, after omittingthat case
   collect all estimates in a vector as you go
take variances across cases
scale up variances
take the square roots
```

- Refactor by extracting the common "omit one" operation
- Refactor by defining a general "jackknife" operation

The common operation

Let's define a function for the common operation of omitting one case

```
omit.case = function(dat,i) {
  dims = dim(dat)
  if (is.null(dims) || (length(dims)==1)) {
    return(dat[-i])
  } else {
    return(dat[-i,])
  }
}
```

(Exercise: modify so it also handles higher-dimensional arrays)

The general operation

Let's write a function for the general jackknife workflow

```
jackknife = function(estimator,dat) {
  n = ifelse(is.null(dim(dat)),length(dat),nrow(dat))
  omit.and.est = function(i) { estimator(omit.case(dat,i)) }
  jackknife.ests = matrix(sapply(1:n, omit.and.est), ncol=n)
  variance.of.ests = apply(jackknife.ests,1,var)
  jackknife.var = ((n-1)^2/n)*variance.of.ests
  jackknife.stderr = sqrt(jackknife.var)
  return(jackknife.stderr)
}
```

Could allow other arguments to `estimator`, spin off finding `n` as its own function, etc.

It works!

```
jackknife(estimator=mean,dat=some.normals)
```

```
## [1] 0.5189194
```

```
max(abs(jackknife(estimator=mean,dat=some.normals) -
          mean.jackknife(some.normals)))
```

```
## [1] 0
```

```
max(abs(jackknife(estimator=gamma.est,dat=cats$Hwt) -
          gamma.jackknife(cats$Hwt)))
```

```
## [1] 0
```

```
est.coefs = function(dat) {
  return(lm(Hwt~Bwt,data=dat)$coefficients)
}
est.coefs(cats)
```

```
## (Intercept)         Bwt
##  -0.3566624   4.0340627
```

```
max(abs(est.coefs(cats) - coefficients(cats.lm)))
```

```
## [1] 0
```

```
jackknife(estimator=est.coefs,dat=cats)
```

```
## [1] 0.8314142 0.3166847
```

```
max(abs((jackknife(estimator=est.coefs,dat=cats) -
          jackknife.lm(df=cats,formula="Hwt~Bwt",p=2))))
```

```
## [1] 0
```

Refactoring + testing

Note: we've been just cross-checking our code the whole time against our old code, to make sure it works

Summary

- Top-down design is a recursive heuristic for coding
  - Split your problem into a few sub-problems; write code tying their solutions together
  - If any sub-problems still need solving, go write their functions
- Leads to multiple short functions, each solving a limited problem
- Disciplines you to think algorithmically
- Once you have working code, re-factor it to make it look more like it came from a top-down design
  - Factor out similar or repeated sub-operations
  - Factor out common over-all operations