Re-writing Your Code

36-402, Advanced Data Analysis

Based on discussion in lecture, 8 February 2011

Suppose we want to find a standard error for the median of a Gaussian distribution. We know, somehow, that the mean of the Gaussian is 3, the standard deviation is 2, and the sample size is one hundred. If we do

```
x <- rnorm(n=100,mean=3,sd=2)
```

we'll get a draw from that distribution in `x`. If we do

```
x <- rnorm(n=100,mean=3,sd=2)
median(x)
```

we'll calculate the median on one random draw. Following the general idea of bootstrapping we can approximate the standard error of the median by repeating this many times and taking the standard deviation. We'll do this by explicitly iterating, so we need to set up a vector to store our intermediate results first.

```
medians <- vector(length=100)
for (i in 1:100) {
  x <- rnorm(n=100,mean=3,sd=2)
  medians[i] <- median(x)
}
se.in.median <- sd(medians)
```

Well, how do we know that 100 replicates is enough to get a good approximation? We'd need to run this a couple of times, typing it in or at least pasting it in many times. Instead, we can write a function which just gives everything we've done a single name. (I'll add comments as I go on.)

```
# Inputs: None; everything is hard-coded
# Output: the standard error in the median
find.se.in.median <- function() {
  # Set up a vector to store the simulated medians
  medians <- vector(length=100)
  # Do the simulation 100 times
  for (i in 1:100) {
    x <- rnorm(n=100,mean=3,sd=2) # Simulate
```

```
    medians[i] <- median(x) # Calculate the median of the simulation
  }
  se.in.median <- sd(medians) # Take standard deviation
  return(se.in.median)
}
```

If we decide that 100 replicates isn't enough and we want 1000, we need to change this function. We could just change the first two appearances of "100" to "1000", but we have to catch all of them; we have to remember that the 100 in `rnorm` is there for a different reason and leave it alone; and if we later decide that actually 500 replicates would be enough, we have to do everything all over again.

It is easier, safer, clearer and more flexible to abstract a little and add an argument to the function, which is the number of replicates. I'll add comments as I go.

```
# Inputs: Number of bootstrap replicates B
# Output: the standard error in the median
find.se.in.median <- function(B) {
  # Set up a vector to store the simulated medians
  medians <- vector(length=B)
  # Do the simulation B times
  for (i in 1:B) {
    x <- rnorm(n=100,mean=3,sd=2) # Simulate
    medians[i] <- median(x) # Calculate median of the simulation
  }
  se.in.median <- sd(medians) # Take standard deviation
  return(se.in.median)
}
```

Now suppose we want to find the standard error of the median for an exponential distribution with rate 2 and sample size 37. We could write another function,

```
find.se.in.median.exp <- function(B) {
  # Set up a vector to store the simulated medians
  medians <- vector(length=B)
  # Do the simulation B times
  for (i in 1:B) {
    x <- rexp(n=37,rate=2) # Simulate
    medians[i] <- median(x) # Calculate median of the simulation
  }
  se.in.median <- sd(medians) # Take standard deviation
  return(se.in.median)
}
```

but it is wasteful to define two functions which do almost the same job. It's not just inelegant; it invites mistakes, it's harder to read (imagine coming back to

this in two weeks — was there a big reason why we had two separate functions here?), and it's harder to improve. We need to abstract a bit more.

We *could* put in some kind of switch which would simulate from either of these two distributions, maybe like this:

```
# Inputs: number of replicates (B)
  # flag for whether to use a normal or an exponential (use.norm)
# Output: The standard error in the median
find.se.in.median <- function(B,use.norm=TRUE) {
  medians <- vector(length=B)
  for (i in 1:B) {
    if (use.norm) {
      x <- rnorm(100,3,2)
    } else {
      x <- rexp(37,2)
    }
    medians[i] <- median(x)
  }
  se.in.median <- sd(medians)
  return(se.in.median)
}
```

but why just these two? If we wanted any other distribution whatsoever, plainly all we'd have to do is change how `x` is simulated. So we really want to be able to *give* a simulator to the function as an argument.

Fortunately, in R you can give one function as an argument to another, so we'd do something like this.

```
# Inputs: Number of replicates (B)
  # Simulator function (simulator)
# Presumes: simulator is a no-argument function which produce a vector of
  # numbers
# Output: The standard error in the media
find.se.in.median <- function(B,simulator) {
  median <- vector(length=B)
  for (i in 1:B) {
    x <- simulator()
    medians[i[ <- median(x)
  }
  se.in.median <- sd(medians)
  return(se.in.medians)
}
```

Now to repeat our original calculations, we define a simulator function:

```
# Inputs: None
# Output: ten draws from the mean 3, s.d. 2 Gaussian
simulator.1 <- function() {
```

```
  return(rnorm(10,3,2))
}
```

If we now call

```
find.se.in.median(B=100,simulator=simulator.1)
```

then every time `find.se.in.median` goes through the `for` loop, it will call `simulator.1`, which in turn will produce the right random numbers. If we also define

```
# Inputs: None
# Output: 37 draws from the rate 2 exponential
simulator.2 <- function() {
  return(rexp(37,2))
}
```

then to find the standard error in the median of *this*, we just call

```
find.se.in.median(B=100,simulator=simulator.2)
```

This same approach works if we want to sample from a much more complicated distribution. If we fit a locally-linear kernel regression to the Old Faithful data, and want a standard error in the median of the predicted waiting times, with noise coming from resampling cases, we would do something like this for the simulator

```
# Inputs: None
# Output: The fitted waiting times of a bootstrapped kernel smooth from the
  # geyser data
simulator.3 <- function() {
  if (!exists("geyser")) {
    require(MASS)
    data(geyser)
  }
  n <- nrow(geyser)
  resampled.rows <- sample(1:n,size=n,replace=TRUE)
  geyser.r <- geyser[resampled.rows,]
  fit <- npreg(waiting~duration,data=geyser.r,regtype="ll")
  waiting.times <- npreg$mean
  return(waiting.times)
}
```

and then this for to find the standard error in the median:

```
find.se.in.median(B=100,simulator=simulator.3)
```

By breaking up the task this way, if we encounter errors or just general trouble when we run that last command, it is easier to localize the problem. We can check whether `find.se.in.median` seems to work properly with other

simulator functions. (For instance, we might right a "simulator" that either does `rep(10,1)` or `rep(10,-1)` with equal probability, since then we can work out what the standard error of the median ought to be.) We can also check whether `simulator.3` is working properly, and finally whether there is some issue with putting them together, say that the output from the simulator is not quite in a format that `find.se.in.median` can handle. If we just have one big ball of code, it is much harder to read, to understand, to debug, and to improe.

To turn to that last point, one of the things R does poorly is explicit iteration with `for` loops. (This was a design choice, but the reasons for it are not our concern here.) It's generally better to replace such loops with "vectorized" functions, which do the iteration using fast code outside of R. One of these, especially for this situation, is the function `replicate`. We can re-write `find.se.in.median` using it:

```
# Inputs: number of replicates (B)
  # Simulator function (simulator)
# Presumes: simulator is a no-argument function which produces a vector of
  # numbers
# Outputs: Standard error in the median of the output of simulator
find.se.in.median <- function(B,simulator) {
  medians <- replicate(B,median(simulator()))
  se.in.median <- sd(medians)
  return(se.in.median)
}
```

Again: shorter, faster, and easier to understand (if you know what `replicate` does). Also, because we are telling this what simulation function to use, and writing those functions separately, we do not have to change any of our simulators. They don't *care* how `find.se.in.median` works. In fact, they don't care that there is any such function — they could be used as components in many other functions which can also process their outputs. So long as these *interfaces* are maintained, the inner workings of the functions are irrelevant to each other.

Suppose for instance that we want not the standard error of the median, but the interquartile range of the median — the median is after all a "robust", outlier-resistant measure of the central tendency, and the IQR is likewise a robust measure of dispersion. This is now easy:

```
# Inputs: number of replicates (B)
  # Simulator function (simulator)
# Presumes: simulator is a no-argument function which produces a vector of
  # numbers
# Outputs: Interquartile range of the median of the output of simulator
find.iqr.of.median <- function(B,simulator) {
  medians <- replicate(B,median(simulator()))
  iqr.of.median <- IQR(medians)
  return(iqr.of.median)
}
```

Or for that matter the good old standard error of the mean:

```
# Inputs: number of replicates (B)
  # Simulator function (simulator)
# Presumes: simulator is a no-argument function which produces a vector of
  # numbers
# Outputs: Standard error of the mean of the output of simulator
find.se.of.mean <- function(B,simulator) {
  means <- replicate(B,mean(simulator()))
  se.of.mean <- sd(means)
  return(se.of.mean)
}
```

These last few examples suggest that we could abstract even further, by swapping in and out different estimators (like `median` and `mean`) and different summarizing functions (like `se` or `IQR`).

```
# Inputs: number of replicates (B)
  # Simulator function (simulator)
  # Estimator function (estimator)
  # Sample summarizer function (summarizer)
# Presumes: simulator is a no-argument function which produces a vector of
  # numbers
  # estimator is a function that takes a vector of numbers and produces one
  # output
  # summarizer takes a vector of outputs from estimator
# Outputs: Summary of the simulated distribution of estimates
summarize.sampling.dist.of.estimates <- function(B,simulator,estimator,
                                                 summarizer) {
  estimates <- replicate(B,estimator(simulator()))
  return(summarizer(estimates))
}
```

The name is too long, of course, so we should replace it with something catchier:

```
bootstrap <- function(B,simulator,estimator,summarizer) {
  estimates <- replicate(B,estimator(simulator()))
  return(summarizer(estimates))
}
```

Our very first example is equivalent to

```
bootstrap(B=100,simulator=simulator.1,estimator=median,summarizer=sd)
```

`bootstrap` is just two lines: one simulates and re-estimates, the other summarizes the re-estimates. This is the essence of what we are trying to do, and is logically distinct from the details of particular simulators, estimators and summaries.

We started with a particular special case and generalized it. The alternative route is to start with a very general framework — here, writing `bootstrap` — and then figure out what lower-level functions we would need to make it work in a the case at hand, writing them if necessary. (We need to write a simulator, but someone's already written `median` for us.) Getting the first stage right involves a certain amount of reflection on how to solve the problem — it's rather like the strategy of doing a "show that" math problem by starting from the desired conclusion and working backwards.

It is still somewhat clunky to have to write a new function every time we want to change the settings in the simulation, but this has gone on long enough.