Mostly programming

Thomas Lumley

Biostatistics

2004-10-28
Functions

We saw simple functions last week.

function(x) c(mean = mean(x), stddev = sd(x))

Functions are more important in R than in other statistical packages and more important than in many programming languages.

Many operations that would be built-in for other packages are done by applying simple functions.
Example: ROC curve

Recall the example at the end of last week: plotting the sensitivity and specificity of a continuous variable as a predictor of a binary variable in an ROC curve.

ROC <- function(test, disease){
  cutpoints <- c(-Inf, sort(unique(test)), Inf)
  sensitivity <- sapply(cutpoints,
                        function(result) mean(test>result & disease)/mean(disease))
  specificity <- sapply(cutpoints,
                        function(result) mean(test<=result & !disease)/mean(!disease))
  plot(sensitivity, 1-specificity, type="l")
  abline(0,1,lty=2)
  return(list(sens=sensitivity, spec=specificity))
}
Example: ROC curve

> x<-rnorm(100,mean=0)
> y<-rnorm(100, mean=1)
> isx<-rep(c(TRUE,FALSE),each=100)
> ROC(c(x,y), isx)
$sens
[1] 1.00 0.99 0.98 0.97 0.96 0.95 0.94 0.93 0.93 0.93 0.92 0.91 0.90 0.89 0.88 0.87 0.86 0.86 0.85 0.85
[21] 0.85 0.84 0.83 0.82 0.81 0.80 0.79 0.78 0.77 0.76 0.75 0.74 0.73 0.72 0.71 0.70 0.70 0.69 0.69
...
Example: ROC curve
Notes

- `sort` sorts a vector, so `sort(unique(test))` are the ordered observed values. `-Inf` and `Inf` are added to ensure that the curve gets to (0,0) and (1,1).

- `disease` is a logical variable (or treated as one). `!disease` means "not disease"

- Variables created inside the function are local

- In R, variables that are visible where a function is defined (eg `test` and `disease`) will be visible inside the function. This isn’t true in S-PLUS, where this ROC function won’t work. Read 3.3.1 and 7.12 in the R FAQ if you are curious.

In S-PLUS we would have to write
sensitivity<-sapply(cutpoints,
    function(result,test, disease)
        mean(test>result & disease)/mean(disease),
        test=test,
        disease=disease)

making this a less attractive approach.

- return() is optional. Recall that every expression in R has some value: the value of the last expression will be returned.

- rep() repeats things. Two most common versions are rep(something, times) and rep(somethings, each=times), but there are more complex versions.
Theoretical note

In principle, the use of user-written functions and second-order functions such as `apply()` and `by()` makes it possible never to change the value of a variable.

Variables can then be thought of as names for values, as in math; rather than storage for values, as in C or Fortran.

The extremist form of this position is called "functional programming". It is a useful idea in moderation.

Along these lines, note that Stata distinguishes `generate` and `replace` for creating and modifying variables.
Historical and cultural note

There have always been multiple versions of the assignment operator available in R and S, not always the same ones.

- In the Old Days, R and S-PLUS allowed `<-` and `_.` The underscore actually printed as a left arrow on some Bell Labs terminals.

- In S-PLUS since 5.0 and R since 1.4.0 `=` has been allowed as an alternative to `<-`.

- In R since 1.8.0 the `_` has been removed as an assignment operator and is now an ordinary character that can be used in variable names (as in Stata)
In R, `=` can’t be used in some places (where you probably wouldn’t have meant to do an assignment), so that

```r
a = 4
if(a = 5) b = 4
print(a)
```

gives 5 on S-PLUS and a syntax error in R.

I use `<-`, but there’s nothing wrong with using `=` if you prefer. Do get used to leaving spaces around it.

Don’t use `_`, even in S-PLUS where it is legal. You can’t imagine how much some people hate it.
Example: computing the median

Suppose we wanted to write a function to compute the median. A simple algorithm is to sort the data and take the middle element.

ourmedian <- function(x){
    n<-length(x)
    sort(x)[(n+1)/2]
}

For even sample sizes we might prefer the average of the two middle values

ourmedian <- function(x){
    n<-length(x)
    if (n %% 2==1) ## odd
        sort(x)[(n+1)/2]
    else {  ## even
        middletwo <- sort(x)[(n/2)+0:1]
        mean(middletwo)
    }
}
We need to handle missing values

ourmedian <- function(x, na.rm=FALSE){
  if(any(is.na(x))) {
    if(na.rm)
      x<-x[!is.na(x)]
    else
      return(NA)
  }
  n<-length(x)
  if (n %% 2==1) ## odd
    sort(x)[(n+1)/2]
  else { ## even
    middletwo <- sort(x)[(n/2)+0:1]
    mean(middletwo)
  }
}
We might also want to

- Check that $x$ is numeric, so that a median makes sense
- Check that $n$ is not 0

The built-in function also takes advantage of an option to `sort()` that stops sorting when specific indices (eg $(n+1)/2$) are correct. This is faster for large vectors (by 1sec=50% for $n = 10^6$).
Median survival

Slightly more complex is a function that computes median (or other quantile) of survival from the Kaplan–Meier estimator. Here we try a top-down approach

```r
mediansurv<-function(time, event, quantile=0.5){
  km <- kaplanmeier(time, event)
  findquantile(km, quantile)
}
```
Kaplan-Meier estimator

Recall that this is the product over times of the fraction of people surviving. If all the times are different, this is easy

```r
kaplanmeier <- function(time, event) {
  if(any(duplicated(time)))
    stop("I can’t cope: some times are equal")

  index <- order(time)
  time <- time[index]
  event <- event[index]
  n <- length(event)
  list(time = time, surv = cumprod(1 - event / (n:1)))
}
```
The function almost works when times are tied: instead of multiplying by, say,

\[
\left(1 - \frac{4}{50}\right)
\]

when 4 die out of 50 it will multiply by

\[
\left(1 - \frac{1}{50}\right)\left(1 - \frac{1}{49}\right)\left(1 - \frac{1}{48}\right)\left(1 - \frac{1}{47}\right)
\]

which is a reasonable estimator, but not the KM estimator.

We need to get the distinct times at which deaths occur, the number of deaths at each of these times, and the number at risk at each time: we need a table of event by time.
kaplanmeier<-function(time,event){
    tab <- table(time,event)
    n<-length(time)
    ndead<-tab[,2]
    ngone<-tab[,1]+tab[,2]
    nalive<-n-cumsum(ngone)+ngone
    list(time=sort(unique(time)),
         surv=cumprod(1-ndead/nalive)
    )
}
• It still doesn't work if everyone is censored (ok) or if no-one is censored (not ok), since the table will have only one column. Cases like this are a real pain in programming. A useful trick is to make a factor, so R knows there should be two levels

```r
tab <- table(time, factor(event, levels=c(0,1)))
```

• We adopted the convention that censorings come after deaths if they are recorded at the same time. You could do it the other way.

• `n-cumsum(ngone)+ngone` could also be done with indices:

```r
ngone <- c(0, tab[,1]+tab[,2])[-(n+1)]
nalive <- n-ngone
```
Finding the quantile

To find the median, we find where the probability first goes below 0.5.

```r
findquantile <- function(km, quantile) {
  below <- which(km$surv < quantile)
  firstbelow <- min(below)
  km$time[min(below)]
}
```

- What happens if the KM estimator never falls below 0.5? What should happen?
- As you saw in your assignment, this could be written as a single line, rather than a function.
Capturing output

To send text output to a file

sink("filename")

and to cancel it

sink()

- Error messages are not diverted.
- Use sink("filename",split=TRUE) to send output to the file and to the screen
To capture output in a variable, use `capture.output()`

```r
> output <- capture.output(example(by))
> length(output)
[1] 107
> output[1]
[1] ""
> output[2]
[1] "by> require(stats)"
> output[3]
[1] "[1] TRUE"
```
Capturing pretty output

Having chunks of output in typewriter font in the middle of the document is convenient but you may want something prettier.

The `xtable()` function in the `xtable` package will produce LaTeX or HTML tables from matrices or from statistical model output. The HTML can be saved to a file and read into eg Word or Powerpoint.
Integrating output and documentation

Sweave is an adaptation to statistics of a concept called literate programming. The idea is to write textual documentation and chunks of code in a single file, and then process this file to produce pretty documentation or extract the code.

An Sweave file is made up of documentation chunks (in \LaTeX) and code chunks (in R). When it is processed, each chunk is run through R. The text chunks and the output of the code are combined into a single file.

The most important advantage of Sweave is that you can be sure that the output in the document corresponds to the code in the document. It is also a convenient way to work, at least if you know \LaTeX.

In addition to the example file you have, the R Daycamp notes are produced with Sweave.
Integrating output and documentation

More documentation for Sweave is in the R Newsletter, Vol 3 No 2 and Vol 2 No 3, on the http://www.r-project.org website.