Mostly programming

Thomas Lumley

Biostatistics

2006-10-26

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

```
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

```
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

  ```
  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:

  ```
  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.

```
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?

Standard errors of quantiles

The homework asked for standard errors of the median by simulation. Other options are asymptotic theory and numerical integration.

Asymptotic theory says that the standard error is approximately $1/2\sqrt{n}f(m)$ where n is the sample size, m is the median, and $f()$ is the pdf.

For numerical integration we need the pdf of the median: the pdf of the rth order statistic in a sample of size n is

$$f_{r,n}(x) = \binom{n}{r} F(x)^{r-1} f(x)(1 - F(x))^{n-r}$$

Standard errors of quantiles

We can write this formula as a function

```
dorder <- function(n, r, pfun, dfun) {
    con <- round(exp(lgamma(n + 1) - lgamma(r) - lgamma(n - r + 1)))
    function(x) {
        con * pfun(x)^(r - 1) * (1 - pfun(x))^(n - r) * dfun(x)
    }
  }
```

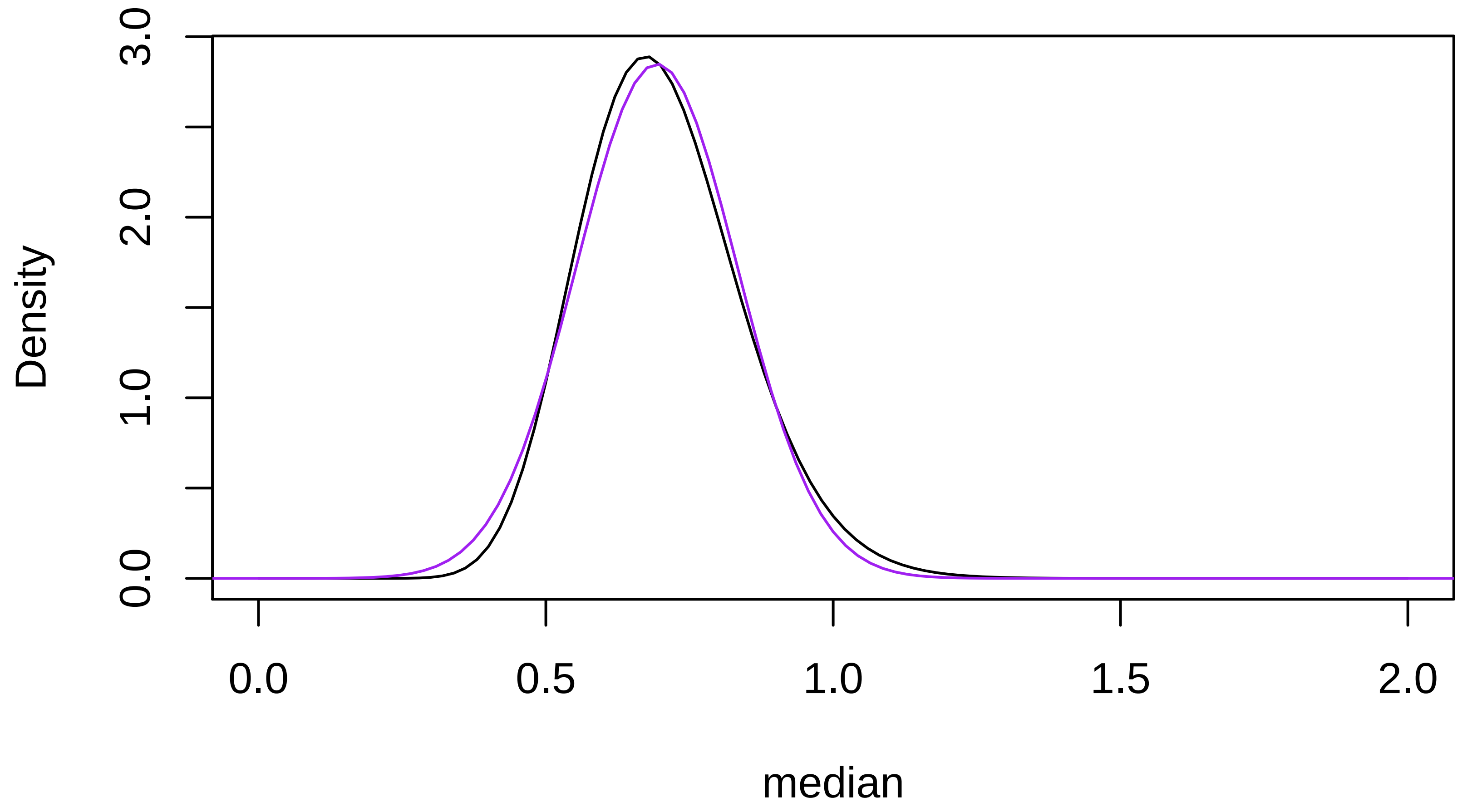
In addition to the n and r arguments, which are numbers, the function takes two functions as arguments, $F()$ and $f()$. These might be built-in functions such as `pnorm()` and `dnorm()` or functions that you write

Standard errors of quantiles

For example, the median of sample of size 51 from an Exponential(1) distribution compared to the asymptotic Normal distribution.

```
dmed51exp<-dorder(51,26,pexp,dexp)
> dmed51exp
function(x) {
        con * pfun(x)^(r - 1) * (1 - pfun(x))^(n - r) * dfun(x)
    }
<environment: 0x193b844>
> curve(dmed51exp,from=0,to=2,ylab="Density",xlab="median")
> curve(dnorm(x,m=log(2),s=1/(2*sqrt(51)*dexp(log(2)))),
        add=TRUE,col="purple")
```

Standard errors of quantiles



Integration

We can integrate functions of this pdf to find the moments of the median. The standard error is the square root of the variance of the median.

```
> moment1<-integrate(function(x) x*dmed51exp(x),
                     lower=-Inf,upper=Inf)
> moment2<-integrate(function(x) x*x*dmed51exp(x),
                     lower=-Inf,upper=Inf)
> moment1
0.702855 with absolute error < 6.7e-06
> moment2
0.513799 with absolute error < 6.9e-05
> sqrt(moment2$value -moment1$value^2)
[1] 0.1406904
```

Note that the median is not unbiased: the true median is $\log 2 \approx 0.6931472$

Integration

Now we can compare this to the simulation and asymptotic standard errors

```
> sd(replicate(10000,median(rexp(51))))
[1] 0.1406540
> 1/(2*sqrt(51)*dexp(log(2)))
[1] 0.140028
```

Maximization

The `optimize()` function maximizes or minimizes a function of one variable, `optim()` handles many variables. [A related function is `uniroot()`, which solves an equation in one variable.]

```
> optimize(dmed51exp,c(0,2),maximum=TRUE)
$maximum
[1] 0.6737122
$objective
[1] 2.891226
```

The mode of the density is not (quite) at the true median either

```
> log(2)
[1] 0.6931472
> dmed51exp(log(2))
[1] 2.863017
```

Matrix operations

You have seen `matrix()` for creating a matrix.

Arithmetic operations on matrices are elementwise: `*`, `/`, `+`, `-`, `&`, `|`

Matrix multiplication is `%*%`

Transposition is `t()`, but also note `crossprod` for X^TY and `tcrossprod` for XY^T, which are more efficient than transposing and multiplying

`solve(A,b)` solves the system of linear equations $Ax = b$. Without `b` it returns A^{-1}

`eigen()` computes eigenthingies, `qr()` and `svd()` do the QR and singular value decomposition.

Matrix operations

The standard linear regression formula

$$\widehat{\beta} = (X^T X)^{-1} X^T Y$$

can be written as

```
solve(t(X) %*% X) %*% t(X) %*% Y
solve(crossprod(X)) %*% crossprod(X,Y)
solve(X,Y)
```

The last version is not only faster, it is more accurate (which used to matter more in the days before 16 decimal place hardware)

Wheel, reinventing of

The built-in functions for integration, maximization, matrix operations, etc, are more convenient than writing your own.

They also work better. For example `help(optimize)` says

```
The method used is a combination of golden section search and
successive parabolic interpolation.  Convergence is never much
slower than that for a Fibonacci search.  If 'f' has a continuous
second derivative which is positive at the minimum (which is not
at 'lower' or 'upper'), then convergence is superlinear, and
usually of the order of about 1.324.
```

which is probably not what you would have implemented.

From the 10 Commandments for C Programmers: Thou shalt study thy libraries and strive not to reinvent them without cause, that thy code may be short and readable and thy days pleasant and productive.

Debugging and optimization

Premature optimization is the root of all evil

Donald Knuth

The first task when programming is to get the program correct, which is easier if it is written more simply and clearly.

Some clarity optimizations make the code faster, eg operating on whole vectors rather than elements. Some have no real impact, eg using `*apply` functions. Some make the code slower, like adding names to vectors.

When the code works correctly, the next step is to find out which parts, if any, are too slow, and then speed them up. This requires measurement, rather than guessing.

Timing

- `proc.time()` returns the current time. Save it before a task and subtract from the value after a task.

- `system.time()` to time the evaluation of `expression`

- `Rprof(filename)` turns on the profiler, and `Rprof(NULL)` turns it off. The profiler writes a list of the current functions being run to `filename` many times per second. `summaryRprof(filename)` summarizes this to report how much time is spent in each function.

Remember that a 1000-fold speedup in a function that uses 10% of the time is less helpful than a 30% speedup in a function that uses 50% of the time.

Profiler

An example from code I wrote earlier this month: Hamiltonian MCMC for source apportionent. The code includes multiplication of large matrices, and also constraints to make the parameters non-zero, involving $\log\max(\theta, 0)$

```
> summaryRprof("spokane.prof")
$by.self
                   self.time self.pct total.time total.pct
"pmax"                 63.04     38.5      94.24      57.6
"/"                    12.82      7.8      12.82       7.8
"-"                     9.58      5.9       9.58       5.9
"cbind"                 9.08      5.6      10.72       6.6
"%*%"                   8.06      4.9       8.06       4.9
"&"                     7.48      4.6       7.48       4.6
"log"                   7.36      4.5      44.38      27.1
"*"                     5.62      3.4       5.62       3.4
```

Profiler

```
"crossprod"                 5.12      3.1       5.12      3.1
"<"                         4.74      2.9       4.74      2.9
"is.na"                     4.32      2.6       4.32      2.6
"leapfrog"                  4.28      2.6     151.28     92.5
"tcrossprod"                4.16      2.5       4.16      2.5
"^"                         3.36      2.1       3.36      2.1
...
```

`pmax(x,y)` computes the elementwise maximum of of `x` and `y`, but I was using it just for $y = 0$.

Rewriting as $x[x < 0] < -0$ and a few other minor changes gave:

Profiler

```
$by.self
              self.time self.pct total.time total.pct
"/"              116.50     15.7     116.50      15.7
"-"               93.80     12.6      93.80      12.6
"%*%"             78.14     10.5      78.14      10.5
"log"             75.60     10.2      75.60      10.2
"penal"           51.30      6.9     156.22      21.1
"*"               47.80      6.4      47.80       6.4
"tcrossprod"      43.96      5.9      43.96       5.9
"leapfrog"        40.00      5.4     654.92      88.3
"crossprod"       37.80      5.1      37.80       5.1
"^"               30.86      4.2      30.86       4.2
```

and roughly doubled the speed.

Memory

- `gc()` will report maximum allocation since the last call to `gc(reset=TRUE)`.
- `gcinfo(TRUE)` asks for a report on memory use every time the garbage collector is run.
- In R 2.4.0 there are additional memory profiling tools.

Debugging

- `traceback()` shows where S was at the last error: what function it was in, where this was called from, and so on back to your top-level command.
- `options(error=dump.frames)` saves the entire state of your program when an error occurs. `debugger()` then lets you start the debugger to inspect any function that was being run. `options(error=recover)` starts the debugger as soon as an error occurs.
- `browser()` starts the debugger at this point in your code.
- `options(warn=2)` turns warnings into errors.
- `debug(fname)` starts the debugger when function `fname()` is called.

The debugger in R gives you an interactive command prompt inside your function.

Faster code

- Operations on whole vectors are fast.
- Matrix operations may be faster even than naive C code
- Functions that have few options and little error checking are faster: eg `sum(x)/length(x)` may be faster than `mean(x)`
- Allocating memory all at once is faster than incremental allocation: `x<-numeric(10000); x[i]<-f(i)` rather than `x<-c(x, f(i))`
- Data frames are much slower than matrices (especially large ones).
- Running out of memory makes code much slower, especially under Windows.

If none of this works, coding a small part of the program in C may make it hundreds of times faster.