Data Manipulation and Functions

Thomas Lumley

Biostatistics

2006-10-19

Merging and matching

The data for an analysis often do not come in a single file. Combining multiple files is necessary.

If two data sets have the same individuals in the same order, they can simply be pasted together side by side.

```
## CHS baseline data
baseline <- read.spss("I:/DISTRIB/BASEBOTH.SAV", to.data.frame=TRUE)
## Events data (eg death, heart attack, ...)
events <- read.spss("I:/SAVEFILES/EVSUM04.SAV", to.data.frame=TRUE)

if (!all(baseline$IDNO==events$IDNO)) {
   stop("PANIC: They don't match!")
} else {
   alldata <- cbind(baseline, events[,c("TTODTH","DEATH",
                      "TTOMI","INCMI")])
}
```

Merging: order

The data might need to be sorted first

```
index1 <- order(baseline$IDNO)
baseline <- baseline[index1,]
index2 <- order(events$IDNO)
events <- events[index2,]
if (!all(baseline$IDNO==events$IDNO)) {
   stop("PANIC: They still don't match!")
} else {
   alldata <- cbind(baseline, events[,c("TTODTH","DEATH",
                    "TTOMI","INCMI")])
}
```

Note that `order(baseline$IDNO)` gives a subset of row numbers containing all the rows but in a different (increasing) order.

Merging: merge

Or there might be different rows in the two data sets

- Some people are missing from one or other data set (eg baseline and year 5 visits)

- Some people have multiple records in one data set (eg baseline data and all hospitalisations

The merge function can do an database outer join, giving a data set that has all the possible matches between a row in one and a row in the other

Merging: merge

```
combined <- merge(baseline, hospvisits, by="IDNO", all=TRUE)
```

- `by=IDNO` says that the IDNO variable indicates individuals who should be matched.
- `all=TRUE` says that even people with no records in the `hospvisits` data set should be kept in the merged version.

How does it work: match

You could imagine a dumb algorithm for merging

```
for(row in firstdataset){
    for(otherrow in seconddataset){
     if (row$IDNO==otherrow$IDNO)
       ##add the row to the result
 }
}
```

More efficiently, the match function gives indices to match one variable to another

```
> match(c("B","I","O","S","T","A","T"),LETTERS)
[1]  2  9 15 19 20  1 20
> letters[match(c("B","I","O","S","T","A","T"),LETTERS)]
[1] "b" "i" "o" "s" "t" "a" "t"
```

Reshaping

Sometimes data sets are the wrong shape. Data with multiple observations of similar quantities can be in long form (multiple records per person) or wide form (multiple variables per person).

Example: The SeattleSNPs genetic variation discovery resource supplies data in a format

```
 SNP   sample al1 al2
000095 D001 C T
000095 D002 T T
000095 D003 T T
```

so that data for a single person is broken across many lines. To convert this to one line per person

```
> data<-read.table("http://pga.gs.washington.edu/data/il6
               /ilkn6.prettybase.txt",
               col.names=c("SNP","sample","allele1","allele2"))
> dim(data)
[1] 2303    4
> wideData<-reshape(data, direction="wide", idvar="sample",
                        timevar="SNP")
> dim(wideData)
[1] 47 99
> names(wideData)
 [1] "sample"       "allele1.95"   "allele2.95"   "allele1.205"
 [5] "allele2.205"  "allele1.276"  "allele2.276"  "allele1.321"
 [9] "allele2.321"  "allele1.657"  "allele2.657"  "allele1.1086"
...
```

- `direction="wide"` says we are going from long to wide format

- `idvar="sample"` says that sample identifies the rows in wide format

- `timevar="SNP"` says that SNP identifies which rows go into the same column in wide form (for repeated measurements over time it would be the time variable)

There is a similar reshape command in Stata. S-PLUS does not have reshape(); converting the R version would be an interesting exercise.

Broken down by age and sex

A common request for Table 1 or Table 2 in a medical paper is to compute means and standard deviations, percentages, or frequency tables of many variables broken down by groups (eg case/control status, age and sex, exposure,...).

That is, we need to apply a simple computation to subsets of the data, and apply it to many variables. One useful function is by(), another is tapply(), which is very similar.

```
> by(airquality$Ozone, list(month=airquality$Month),
       mean, na.rm=TRUE)
month: 5
[1] 23.61538
------------------------------------------------------
month: 6
[1] 29.44444
------------------------------------------------------
month: 7
[1] 59.11538
------------------------------------------------------
month: 8
[1] 59.96154
------------------------------------------------------
month: 9
[1] 31.44828
```

Notes

- The first argument is the variable to be analyzed.
- The second argument is a list of variable defining subsets. In this case, a single variable, but we could do `list(month=airquality$Month, toohot=airquality$Temp>85)` to get a breakdown by month and temperature
- The third argument is the analysis function to use on each subset
- Any other arguments (na.rm=TRUE) are also given to the analysis function
- The result is really a vector (with a single grouping variable) or array (with multiple grouping variables). It prints differently.

Confusing digression: str()

How do I know it is an array? Because str() summarises the internal structure of a variable.

```
> a<- by(airquality$Ozone, list(month=airquality$Month,
                                toohot=airquality$Temp>85),
                                mean, na.rm=TRUE)
> str(a)
 by [1:5, 1:2] 23.6 22.1 49.3 40.9 22.0 ...
 - attr(*, "dimnames")=List of 2
  ..$ month : chr [1:5] "5" "6" "7" "8" ...
  ..$ toohot: chr [1:2] "FALSE" "TRUE"
 - attr(*, "call")= language by.data.frame(data =
     as.data.frame(data), INDICES = INDICES,
     FUN = FUN, na.rm = TRUE)
 - attr(*, "class")= chr "by"
```

One function, many variables

Last week we saw colMeans for finding the mean of each column of a matrix.

There is a general function, apply() for doing something to rows or columns of a matrix (or slices of a higher-dimensional array).

```
> apply(psa[,1:8],2,mean,na.rm=TRUE)
        id      nadir      pretx         ps        bss      grade
 25.500000  16.360000 670.751163  80.833333   2.520833   2.146341
     grade        age    obstime
  2.146341  67.440000  28.460000
```

This is just a slower version of `colMeans`, but the same can be done with other functions such as `sd`, `IQR`, `min`,...

apply

- the first argument is an array or matrix or dataframe
- the third argument is the analysis function
- the second argument says which margins to keep (1=rows, 2=columns, ...), so 2 means that the result should keep the columns: apply the function to each column.
- any other arguments are given to the analysis function

There is a widespread belief that apply() is faster than a for() loop over the columns. This is a useful belief, since it encourages people to use apply(), but it is completely untrue.

New functions

Suppose you want the mean and standard deviation for each variable. One solution is to apply a new function. Watch carefully,...

```
> apply(psa[,1:8], 2, function(x) c(mean=mean(x,na.rm=TRUE),
                                    stddev=sd(x,na.rm=TRUE)))
             id   nadir     pretx       ps       bss     grade
mean   25.50000 16.3600  670.7512 80.83333 2.5208333 2.1463415
stddev 14.57738 39.2462 1287.6384 11.07678 0.6838434 0.7924953
             age  obstime
mean   67.440000 28.46000
stddev  5.771711 18.39056
```

New function

```
function(x) c(mean=mean(x,na.rm=TRUE),
               stddev=sd(x,na.rm=TRUE))
```

translates as: "If you give me a vector, which I will call `x`, I will mean it and sd it and give you the results"

We could give this function a name and then refer to it by name

```
mean.and.sd <- function(x) c(mean=mean(x,na.rm=TRUE),
                              stddev=sd(x,na.rm=TRUE))
apply(psa[,1:8], 2, mean.and.sd)
```

which would save typing if we used the function many times. Note that giving the function a name is not necessary, any more than giving 2 a name.

by() revisited

Now we know how to write simple functions we can use by() more generally

```
> by(psa[,1:8], list(remission=psa$inrem),
     function(subset) round(apply(subset, 2, mean.and.sd), 2))
remission: no
          id nadir   pretx    ps  bss grade   age obstime
mean   31.03 22.52  725.99 79.71 2.71  2.11 67.17   21.75
stddev 11.34 44.91 1362.34 10.29 0.52  0.83  5.62   15.45
-----------------------------------------------------
remission: yes
          id nadir   pretx    ps  bss grade   age obstime
mean   11.29  0.53  488.45 83.57 2.07  2.23 68.14   45.71
stddev 12.36  0.74 1044.14 12.77 0.83  0.73  6.30   13.67
```

Notes

```
function(subset) round(apply(subset, 2, mean.and.sd), 2)
```

translates as “If you give me a data frame, which I will call `subset`, I will apply the `mean.and.sd` function to each variable, round to 2 decimal places, and give you the results”

Functions

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

Plotting the sensitivity and specificity of a continuous variable T as a predictor of a binary variable D gives an ROC curve.

$$\begin{aligned} \text{sensitivity} &= P[T > c|D = 1] \\ \text{specificity} &= P[T \leq c|D = 0] \end{aligned}$$

Compute this for all c and plot sensitivity vs $1-$specificity.

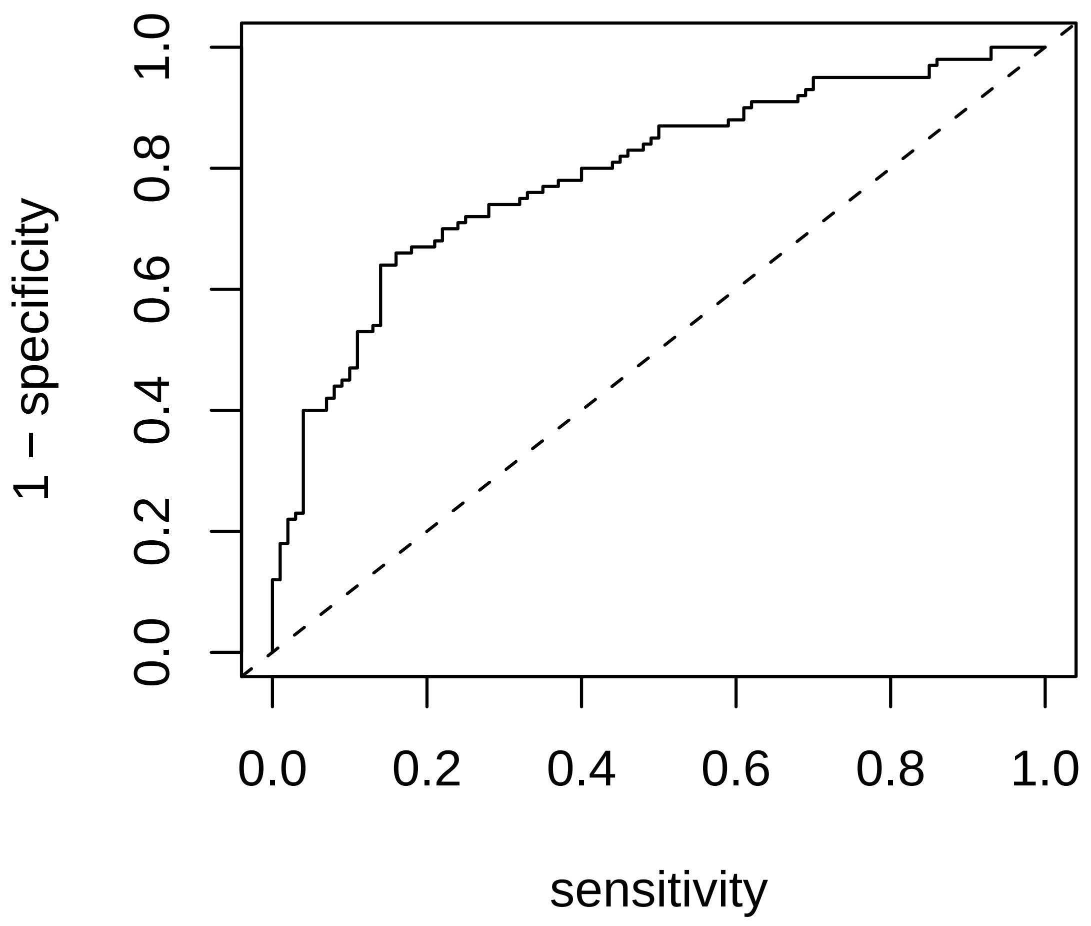
Example: ROC curve

```
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.
 [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.
...
```

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

Notes

```
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 – R is not an ideal language for pure functional programming.

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

```
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.