Summer Institute in Statistical Genetics Module 6: Computing for Statistical Genetics

Thomas Lumley
Ken Rice

3. Data Manipulation

Auckland, December 2008

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...

Reshaping

```
> 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"
...
```

Reshaping

- `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)

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. This is a *loop*.

One useful function is `by()` , another is `tapply()` , which is very similar (but harder to remember).

Broken down `by()` age and sex

```
> 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"
```

Back to loops — `apply()`

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
```

In this case there is a special, faster, function `colMeans` , but the `apply` can be used with other functions such as `sd` , `IQR` , `min` ,...

What does `apply()` need?

- 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 not true.

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 – new object

```
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. The {curly brackets} are optional for a function with just one expression, but necessary for longer functions

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"