1 Introduction

2 Introduction

The doBy package contains a variety of utility functions. This working document describes some of these functions. The package originally grew out of a need to calculate groupwise summary statistics (much in the spirit of `PROC SUMMARY` of the SAS system), but today the package contains many different utilities.

The doBy package (and this document as a .pdf file) is available from
`http://cran.r-project.org/web/packages/doBy/index.html`
The package is loaded with:

```
library(doBy)
```

3 Data used for illustration

The description of the `doBy` package is based on the following datasets.

CO2 data The `CO2` data frame comes from an experiment on the cold tolerance of the grass species *Echinochloa crus-galli*. To limit the amount of output we modify names and levels of variables as follows

```
data(CO2)
CO2 <- transform(CO2, Treat=Treatment, Treatment=NULL)
levels(CO2$Treat) <- c("nchil","chil")
levels(CO2$Type)  <- c("Que","Mis")
CO2 <- subset(CO2, Plant %in% c("Qn1", "Qc1", "Mn1", "Mc1"))
```

Airquality data The `airquality` dataset contains air quality measurements in New York, May to September 1973. The months are coded as $5, \dots, 9$. To limit the output we only consider data for two months:

```
airquality <- subset(airquality, Month %in% c(5,6))
```

Dietox data The `dietox` data are provided in the `doBy` package and result from a study of the effect of adding vitamin E and/or copper to the feed of slaughter pigs.

4 Working with groupwise data

4.1 The `summaryBy` function

The `summaryBy` function is used for calculating quantities like "the mean and variance of x and y for each combination of two factors A and B". Examples are based on the `CO2` data.

4.1.1 Basic usage

For example, the mean and variance of `uptake` and `conc` for each value of `Plant` is obtained by:

```
myfun1 <- function(x){c(m=mean(x), v=var(x))}
summaryBy(conc+uptake~Plant, data=CO2,
 FUN=myfun1)
```

```
  Plant conc.m conc.v uptake.m uptake.v
1   Qn1    435 100950    33.23    67.48
2   Qc1    435 100950    29.97    69.47
3   Mn1    435 100950    26.40    75.59
4   Mc1    435 100950    18.00    16.96
```

Defining the function to return named values as above is the recommended use of `summaryBy`. Note that the values returned by the function has been named as `m` and `v`.

If the result of the function(s) are not named, then the names in the output data in general become less intuitive:

```
myfun2 <- function(x){c(mean(x), var(x))}
summaryBy(conc+uptake~Plant, data=CO2,FUN=myfun2)
```

```
  Plant conc.FUN1 conc.FUN2 uptake.FUN1 uptake.FUN2
1   Qn1       435    100950       33.23       67.48
2   Qc1       435    100950       29.97       69.47
3   Mn1       435    100950       26.40       75.59
4   Mc1       435    100950       18.00       16.96
```

4.1.2 Using predefined functions

It is possible use a vector of predefined functions. A typical usage will be by invoking a list of predefined functions:

```
summaryBy(uptake~Plant, data=CO2, FUN=c(mean,var,median))
```

```
  Plant uptake.mean uptake.var uptake.median
1   Qn1       33.23      67.48          35.3
2   Qc1       29.97      69.47          32.5
3   Mn1       26.40      75.59          30.0
4   Mc1       18.00      16.96          18.9
```

Slightly more elaborate is

```
mymed <- function(x)c(med=median(x))
summaryBy(uptake~Plant, data=CO2, FUN=c(mean,var,mymed))
```

```
  Plant uptake.mean uptake.var uptake.mymed
1   Qn1       33.23      67.48         35.3
2   Qc1       29.97      69.47         32.5
3   Mn1       26.40      75.59         30.0
4   Mc1       18.00      16.96         18.9
```

The naming of the output variables determined from what the functions returns. The names of the last two columns above are imposed by `summaryBy` because `myfun2` does not return named values.

4.1.3 Copying variables out with the `id` argument

To get the value of the `Type` and `Treat` in the first row of the groups (defined by the values of `Plant`) copied to the output dataframe we use the `id` argument: as:

```
summaryBy(conc+uptake~Plant, data=CO2, FUN=myfun1, id=~Type+Treat)
```

```
  Plant conc.m conc.v uptake.m uptake.v Type Treat
1   Qn1    435 100950    33.23    67.48  Que nchil
2   Qc1    435 100950    29.97    69.47  Que  chil
3   Mn1    435 100950    26.40    75.59  Mis nchil
4   Mc1    435 100950    18.00    16.96  Mis  chil
```

4.1.4 Statistics on functions of data

We may want to calculate the mean and variance for the logarithm of `uptake`, for `uptake`+`conc` (not likely to be a useful statistic) as well as for `uptake` and `conc`. This can be achieved as:

```
 summaryBy(log(uptake)+I(conc+uptake)+ conc+uptake~Plant, data=CO2,
  FUN=myfun1)
```

```
  Plant log(uptake).m log(uptake).v conc + uptake.m conc + uptake.v conc.m
1   Qn1         3.467       0.10168           468.2           104747    435
2   Qc1         3.356       0.11873           465.0           105297    435
3   Mn1         3.209       0.17928           461.4           105642    435
4   Mc1         2.864       0.06874           453.0           103157    435
  conc.v uptake.m uptake.v
1 100950    33.23    67.48
2 100950    29.97    69.47
3 100950    26.40    75.59
4 100950    18.00    16.96
```

If one does not want output variables to contain parentheses then setting `p2d=TRUE` causes the parentheses to be replaced by dots ("`.`").

```
 summaryBy(log(uptake)+I(conc+uptake)~Plant, data=CO2, p2d=TRUE,
  FUN=myfun1)
```

```
  Plant log.uptake..m log.uptake..v conc + uptake.m conc + uptake.v
1   Qn1         3.467       0.10168           468.2           104747
2   Qc1         3.356       0.11873           465.0           105297
3   Mn1         3.209       0.17928           461.4           105642
4   Mc1         2.864       0.06874           453.0           103157
```

4.1.5 Using '.' on the left hand side of a formula

It is possible to use the dot (".") on the left hand side of the formula. The dot means "all numerical variables which do not appear elsewhere" (i.e. on the right hand side of the formula and in the `id` statement):

```
 summaryBy(log(uptake)+I(conc+uptake)+. ~Plant, data=CO2,
  FUN=myfun1)
```

```
  Plant log(uptake).m log(uptake).v conc + uptake.m conc + uptake.v conc.m
1   Qn1         3.467       0.10168           468.2           104747    435
2   Qc1         3.356       0.11873           465.0           105297    435
3   Mn1         3.209       0.17928           461.4           105642    435
4   Mc1         2.864       0.06874           453.0           103157    435
  conc.v uptake.m uptake.v
1 100950    33.23    67.48
2 100950    29.97    69.47
3 100950    26.40    75.59
4 100950    18.00    16.96
```

4.1.6 Using '.' on the right hand side of a formula

The dot (".") can also be used on the right hand side of the formula where it refers to "all non–numerical variables which are not specified elsewhere":

```
 summaryBy(log(uptake) ~Plant+., data=CO2,
  FUN=myfun1)
  Plant Type Treat log(uptake).m log(uptake).v
1   Qn1  Que nchil         3.467       0.10168
2   Qc1  Que  chil         3.356       0.11873
3   Mn1  Mis nchil         3.209       0.17928
4   Mc1  Mis  chil         2.864       0.06874
```

4.1.7 Using '1' on the right hand side of the formula

Using 1 on the right hand side means no grouping:

```
summaryBy(log(uptake) ~ 1, data=CO2,
 FUN=myfun1)
```

```
  log(uptake).m log(uptake).v
1         3.224        0.1577
```

4.1.8 Preserving names of variables using keep.names

If the function applied to data only returns one value, it is possible to force that the summary variables retain the original names by setting `keep.names=TRUE`. A typical use of this could be

```
summaryBy(conc+uptake+log(uptake)~Plant,
 data=CO2, FUN=mean, id=~Type+Treat, keep.names=TRUE)
```

```
  Plant conc uptake log(uptake) Type Treat
1   Qn1  435  33.23       3.467  Que nchil
2   Qc1  435  29.97       3.356  Que  chil
3   Mn1  435  26.40       3.209  Mis nchil
4   Mc1  435  18.00       2.864  Mis  chil
```

4.2 The orderBy function

Ordering (or sorting) a data frame is possible with the `orderBy` function. Suppose we want to order the rows of the the `airquality` data by `Temp` and by `Month` (within `Temp`). This can be achieved by:

```
x<-orderBy(~Temp+Month, data=airquality)
```

The first lines of the result are:

```
head(x)
```

```
   Ozone Solar.R Wind Temp Month Day
5     NA      NA 14.3   56     5   5
18     6      78 18.4   57     5  18
25    NA      66 16.6   57     5  25
27    NA      NA  8.0   57     5  27
15    18      65 13.2   58     5  15
26    NA     266 14.9   58     5  26
```

If we want the ordering to be by decreasing values of one of the variables, we change the sign, e.g.

```
x<-orderBy(~-Temp+Month, data=airquality)
head(x)
```

```
   Ozone Solar.R Wind Temp Month Day
42    NA     259 10.9   93     6  11
43    NA     250  9.2   92     6  12
40    71     291 13.8   90     6   9
39    NA     273  6.9   87     6   8
41    39     323 11.5   87     6  10
36    NA     220  8.6   85     6   5
```

4.3 The `splitBy` function

Suppose we want to split the `airquality` data into a list of dataframes, e.g. one dataframe for each month. This can be achieved by:

```
x<-splitBy(~Month, data=airquality)
x
```

```
  listentry Month
1         5     5
2         6     6
```

Hence for month 5, the relevant entry-name in the list is '5' and this part of data can be extracted as

```
x[['5']]
```

Information about the grouping is stored as a dataframe in an attribute called `groupid` and can be retrieved with:

```
attr(x,"groupid")
```

```
  Month
1     5
2     6
```

4.4 The `sampleBy` function

Suppose we want a random sample of 50 % of the observations from a dataframe. This can be achieved with:

```
sampleBy(~1, frac=0.5, data=airquality)
```

Suppose instead that we want a systematic sample of every fifth observation within each month. This is achieved with:

```
sampleBy(~Month, frac=0.2, data=airquality,systematic=T)
```

4.5 The `subsetBy` function

Suppose we want to select those rows within each month for which the the wind speed is larger than the mean wind speed (within the month). This is achieved by:

```
subsetBy(~Month, subset=Wind>mean(Wind), data=airquality)
```

Note that the statement `Wind>mean(Wind)` is evaluated within each month.

4.6 The `transformBy` function

The `transformBy` function is analogous to the `transform` function except that it works within groups. For example:

```
transformBy(~Month, data=airquality, minW=min(Wind), maxW=max(Wind),
     chg=sum(range(Wind)*c(-1,1)))
```

4.7 The `lapplyBy` function

This `lapplyBy` function is a wrapper for first splitting data into a list according to the formula (using splitBy) and then applying a function to each element of the list (using apply).

Suppose we want to calculate the weekwise feed efficiency of the pigs in the `dietox` data, i.e. weight gain divided by feed intake.

```
data(dietox)
dietox <- orderBy(~Pig+Time, data=dietox)
v<-lapplyBy(~Pig, data=dietox, function(d) c(NA, diff(d$Weight)/diff(d$Feed)))
dietox$FE <- unlist(v)
```

Technically, the above is the same as

```
dietox <- orderBy(~Pig+Time, data=dietox)
wdata <- splitBy(~Pig, data=dietox)
v <- lapply(wdata, function(d) c(NA, diff(d$Weight)/diff(d$Feed)))
dietox$FE <- unlist(v)
```

5 Miscellaneous

5.1 The `esticon` function

Consider a linear model which explains `Ozone` as a linear function of `Month` and `Wind`:

```
data(airquality)
airquality <- transform(airquality, Month=factor(Month))
m<-lm(Ozone~Month*Wind, data=airquality)
coefficients(m)
```

```
(Intercept)      Month6      Month7      Month8      Month9        Wind
     50.748     -41.793      68.296      82.211      23.439      -2.368
Month6:Wind Month7:Wind Month8:Wind Month9:Wind
      4.051      -4.663      -6.154      -1.874
```

When a parameter vector β of (systematic) effects have been estimated, interest is often in a particular estimable function, i.e. linear combination $\lambda^\top\beta$ and/or testing the hypothesis $H_0 : \lambda^\top\beta = \beta_0$ where λ is a specific vector defined by the user.

Suppose for example we want to calculate the expected difference in ozone between consequtive months at wind speed 10 mph (which is about the average wind speed over the whole period).

The `esticon` function provides a way of doing so. We can specify several λ vectors at the same time. For example

```
Lambda <- rbind(
   c(0,-1,0,0,0,0,-10,0,0,0),
   c(0,1,-1,0,0,0,10,-10,0,0),
   c(0,0,1,-1,0,0,0,10,-10,0),
   c(0,0,0,1,-1,0,0,0,10,-10)
   )
```

```
 esticon(m, Lambda)
```

```
  beta0 Estimate Std.Error t.value  DF Pr(>|t|)   Lower  Upper
1     0   1.2871    10.238  0.1257 106  0.90019 -19.010 21.585
2     0 -22.9503    10.310 -2.2259 106  0.02814 -43.392 -2.509
3     0   0.9954     7.094  0.1403 106  0.88867 -13.069 15.060
4     0  15.9651     6.560  2.4337 106  0.01662   2.959 28.971
```

In other cases, interest is in testing a hypothesis of a contrast $H_0 : \Lambda\beta = \beta_0$ where Λ is a matrix. For example a test of no interaction between `Month` and `Wind` can be made by testing jointly that the last four parameters in `m` are zero (observe that the test is a Wald test):

```
 Lambda <- rbind(
    c(0,0,0,0,0,0,1,0,0,0),
    c(0,0,0,0,0,0,0,1,0,0),
    c(0,0,0,0,0,0,0,0,1,0),
    c(0,0,0,0,0,0,0,0,0,1)
    )
```

```
 esticon(m, Lambda, joint.test=T)
```

```
  X2.stat DF Pr(>|X^2|)
1   22.11  4  0.0001906
```

For a linear normal model, one would typically prefer to do a likelihood ratio test instead. However, for generalized estimating equations of glm–type (as dealt with in the packages geepack and gee) there is no likelihood. In this case `esticon` function provides an operational alternative.

Observe that another function for calculating contrasts as above is the `contrast` function in the Design package but it applies to a narrower range of models than `esticon` does.

5.2 The `firstobs()` / `lastobs()` function

To obtain the indices of the first/last occurences of an item in a vector do:

```
 x <- c(1,1,1,2,2,2,1,1,1,3)
 firstobs(x)
```

```
[1]  1  4 10
```

```
 lastobs(x)
```

```
[1]  6  9 10
```

The same can be done on a data frame, e.g.

```
 firstobs(~Plant, data=CO2)
```

```
[1]  1  8 15 22
```

```
 lastobs(~Plant, data=CO2)
```

```
[1]  7 14 21 28
```

5.3 The `which.maxn()` and `which.minn()` functions

The location of the n largest / smallest entries in a numeric vector can be obtained with

```
 x <- c(1:4,0:5,11,NA,NA)
 which.maxn(x,3)
```

```
[1] 11 10  4
```

```
 which.minn(x,5)
```

```
[1] 5 1 6 2 7
```

5.4 Subsequences - `subSeq()`

Find (sub) sequences in a vector:

```
 x <- c(1,1,2,2,2,1,1,3,3,3,3,1,1,1)
 subSeq(x)
```

```
  first last slength midpoint value
1     1    2       2        2     1
2     3    5       3        4     2
3     6    7       2        7     1
4     8   11       4       10     3
5    12   14       3       13     1
```

```
 subSeq(x, item=1)
```

```
  first last slength midpoint value
1     1    2       2        2     1
2     6    7       2        7     1
3    12   14       3       13     1
```

```
 subSeq(letters[x])
```

```
  first last slength midpoint value
1     1    2       2        2     a
2     3    5       3        4     b
3     6    7       2        7     a
4     8   11       4       10     c
5    12   14       3       13     a
```

```
 subSeq(letters[x],item="a")
```

```
  first last slength midpoint value
1     1    2       2        2     a
2     6    7       2        7     a
3    12   14       3       13     a
```

5.5 Recoding values of a vector - `recodeVar()`

```
 x <- c("dec","jan","feb","mar","apr","may")
 src1 <- list(c("dec","jan","feb"), c("mar","apr","may"))
 tgt1 <- list("winter","spring")
 recodeVar(x,src=src1,tgt=tgt1)
```

```
[1] "winter" "winter" "winter" "spring" "spring" "spring"
```

5.6 Renaming columns of a dataframe or matrix – `renameCol()`

```
 head(renameCol(CO2, 1:2, c("kk","ll")))
```

```
   kk  ll conc uptake Treat
1 Qn1 Que   95   16.0 nchil
2 Qn1 Que  175   30.4 nchil
3 Qn1 Que  250   34.8 nchil
4 Qn1 Que  350   37.2 nchil
5 Qn1 Que  500   35.3 nchil
6 Qn1 Que  675   39.2 nchil
```

```
head(renameCol(CO2, c("Plant","Type"), c("kk","ll")))
```

```
   kk  ll conc uptake Treat
1 Qn1 Que   95   16.0 nchil
2 Qn1 Que  175   30.4 nchil
3 Qn1 Que  250   34.8 nchil
4 Qn1 Que  350   37.2 nchil
5 Qn1 Que  500   35.3 nchil
6 Qn1 Que  675   39.2 nchil
```

5.7 Time since an event - `timeSinceEvent()`

Consider the vector

```
#yvar <- c(0,0,0,1,0,0,0,0,0,1,0,0,0,0,1,0,1,0,0,0,0,0,0,0,1,1,0,0,0,0,0)
yvar <- c(0,0,0,1,0,0,0,0,0,1,0,0,0,1,1,0,0,0,0,0)
```

Imagine that "1" indicates an event of some kind which takes place at a certain time point. By default time points are assumed equidistant but for illustration we define time time variable

```
#tvar <- seq_along(yvar) + c(0.1,0.2,0.3)
tvar <- seq_along(yvar) + c(0.1,0.2)
```

Now we find time since event as

```
tse<- timeSinceEvent(yvar,tvar)
```

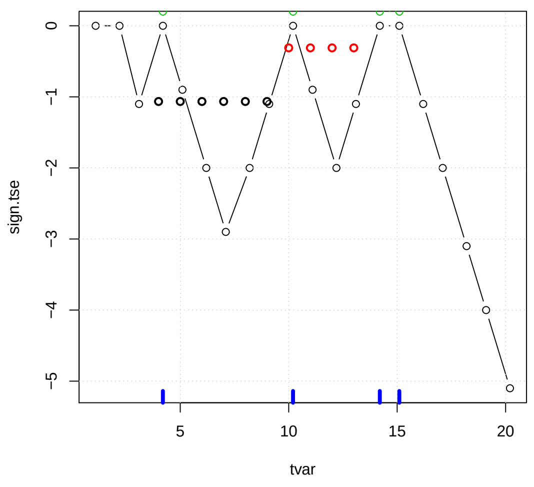
```
   yvar tvar abs.tse sign.tse ewin run tae  tbe
1     0  1.1     3.1      0.0    1  NA  NA -3.1
2     0  2.2     2.0      0.0    1  NA  NA -2.0
3     0  3.1     1.1     -1.1    1  NA  NA -1.1
4     1  4.2     0.0      0.0    1   1 0.0  0.0
5     0  5.1     0.9     -0.9    1   1 0.9 -5.1
6     0  6.2     2.0     -2.0    1   1 2.0 -4.0
7     0  7.1     2.9     -2.9    1   1 2.9 -3.1
8     0  8.2     2.0     -2.0    1   1 4.0 -2.0
9     0  9.1     1.1     -1.1    1   1 4.9 -1.1
10    1 10.2     0.0      0.0    1   2 0.0  0.0
11    0 11.1     0.9     -0.9    1   2 0.9 -3.1
12    0 12.2     2.0     -2.0    1   2 2.0 -2.0
13    0 13.1     1.1     -1.1    1   2 2.9 -1.1
14    1 14.2     0.0      0.0    1   3 0.0  0.0
15    1 15.1     0.0      0.0    1   4 0.0  0.0
16    0 16.2     1.1     -1.1    1   4 1.1   NA
17    0 17.1     2.0     -2.0    1   4 2.0   NA
18    0 18.2     3.1     -3.1    1   4 3.1   NA
19    0 19.1     4.0     -4.0    1   4 4.0   NA
20    0 20.2     5.1     -5.1    1   4 5.1   NA
```

The output reads as follows:

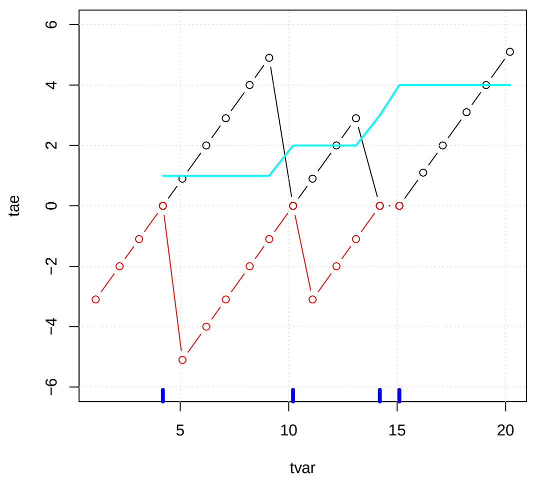
- `abs.tse`: Absolute time since (nearest) event.
- `sign.tse`: Signed time since (nearest) event.
- `ewin`: Event window: Gives a symmetric window around each event.

- `run`: The value of `run` is set to 1 when the first event occurs and is increased by 1 at each subsequent event.
- `tae`: Time after event.
- `tbe`: Time before event.

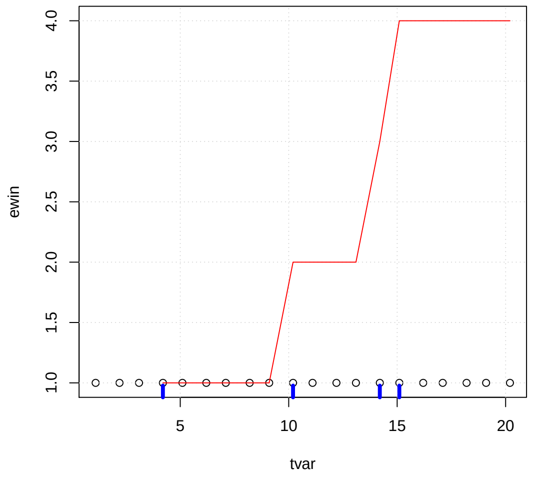
```
plot(sign.tse~tvar, data=tse, type="b")
grid()
rug(tse$tvar[tse$yvar==1], col='blue',lwd=4)
points(scale(tse$run), col=tse$run, lwd=2)
lines(abs.tse+.2~tvar, data=tse, type="b",col=3)
```



```
plot(tae~tvar, data=tse, ylim=c(-6,6),type="b")
grid()
lines(tbe~tvar, data=tse, type="b", col='red')
rug(tse$tvar[tse$yvar==1], col='blue',lwd=4)
lines(run~tvar, data=tse, col='cyan',lwd=2)
```



```
plot(ewin~tvar, data=tse,ylim=c(1,4))
rug(tse$tvar[tse$yvar==1], col='blue',lwd=4)
grid()
lines(run~tvar, data=tse,col='red')
```



We may now find times for which time since an event is at most 1 as

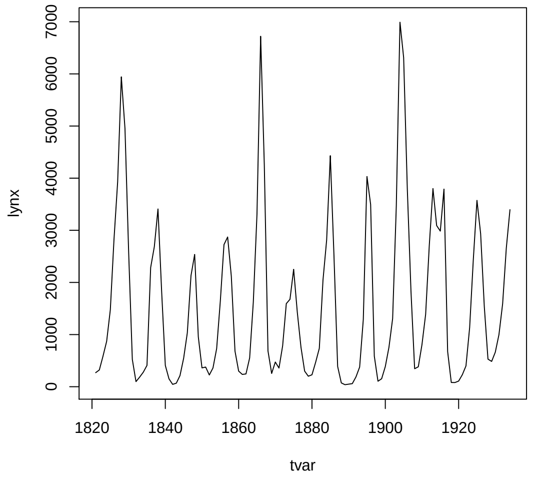
```
tse$tvar[tse$abs<=1]
```

```
[1]  4.2  5.1 10.2 11.1 14.2 15.1
```

5.8 Example: Using `subSeq()` and `timeSinceEvent()`

Consider the `lynx` data:

```
lynx <- as.numeric(lynx)
tvar <- 1821:1934
plot(tvar,lynx,type='l')
```



Suppose we want to estimate the cycle lengths. One way of doing this is as follows:

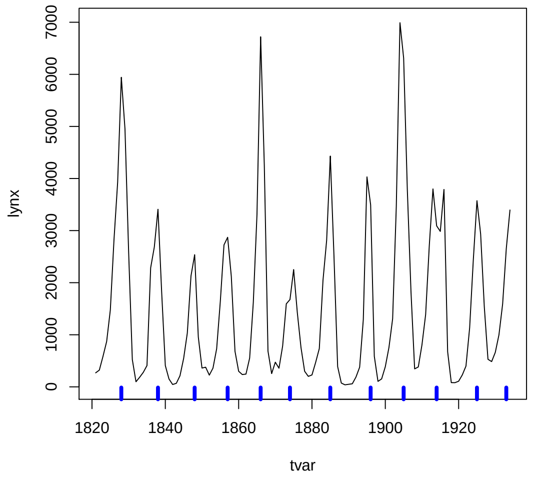
```
yyy <- lynx>mean(lynx)
head(yyy)
```

```
[1] FALSE FALSE FALSE FALSE FALSE  TRUE
```

```
sss <- subSeq(yyy,TRUE)
sss
```

```
   first last slength midpoint value
1      6   10       5        8  TRUE
2     16   19       4       18  TRUE
3     27   28       2       28  TRUE
4     35   38       4       37  TRUE
5     44   47       4       46  TRUE
6     53   55       3       54  TRUE
7     63   66       4       65  TRUE
8     75   76       2       76  TRUE
9     83   87       5       85  TRUE
10    92   96       5       94  TRUE
11   104  106       3      105  TRUE
12   112  114       3      113  TRUE
```

```
plot(tvar,lynx,type='l')
rug(tvar[sss$midpoint],col='blue',lwd=4)
```



Create the 'event vector'

```
yvar <- rep(0,length(lynx))
yvar[sss$midpoint] <- 1
str(yvar)
```

```
num [1:114] 0 0 0 0 0 0 0 1 0 0 ...
```

```
tse <- timeSinceEvent(yvar,tvar)
head(tse,20)
```

```
  yvar tvar abs.tse sign.tse ewin run tae tbe
1    0 1821       7        7    1  NA  NA  -7
```

```
2      0 1822       6         6    1  NA  NA  -6
3      0 1823       5         5    1  NA  NA  -5
4      0 1824       4         4    1  NA  NA  -4
5      0 1825       3         3    1  NA  NA  -3
6      0 1826       2         2    1  NA  NA  -2
7      0 1827       1         1    1  NA  NA  -1
8      1 1828       0         0    2   1   0   0
9      0 1829       1         1    2   1   1  -9
10     0 1830       2         2    2   1   2  -8
11     0 1831       3         3    2   1   3  -7
12     0 1832       4         4    2   1   4  -6
13     0 1833       5         5    2   1   5  -5
14     0 1834       4         4    2   1   6  -4
15     0 1835       3         3    2   1   7  -3
16     0 1836       2         2    2   1   8  -2
17     0 1837       1         1    2   1   9  -1
18     1 1838       0         0    3   2   0   0
19     0 1839       1         1    3   2   1  -9
20     0 1840       2         2    3   2   2  -8
```

We get two different (not that different) estimates of period lengths:

```
len1 <- tapply(tse$ewin, tse$ewin, length)
```

```
 1  2  3  4  5  6  7  8  9 10 11 12 13
 7 10 10  9  9  8 11 11  9  9 11  8  2
```

```
len2 <- tapply(tse$run, tse$run, length)
```

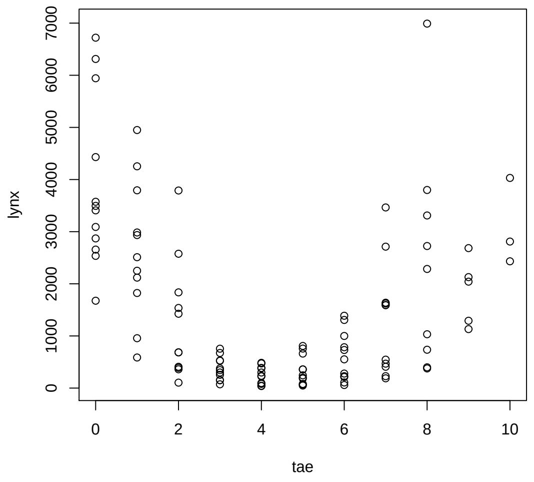
```
 1  2  3  4  5  6  7  8  9 10 11 12
10 10  9  9  8 11 11  9  9 11  8  2
```

```
c(median(len1),median(len2),mean(len1),mean(len2))
```

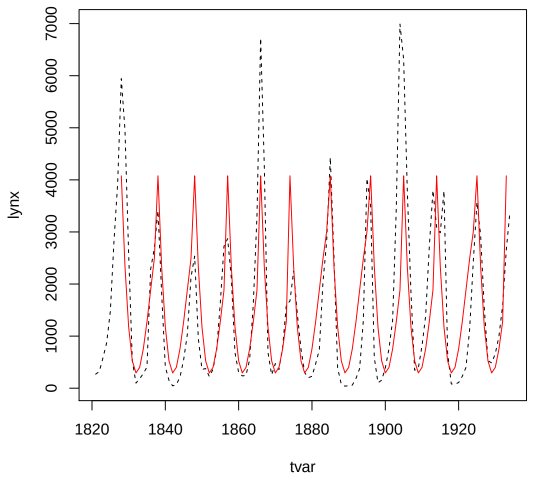
```
[1] 9.000 9.000 8.769 8.917
```

We can overlay the cycles as:

```
tse$lynx <- lynx
tse2 <- na.omit(tse)
plot(lynx~tae, data=tse2)
```



```
plot(tvar,lynx,type='l',lty=2)
mm <- lm(lynx~tae+I(tae^2)+I(tae^3), data=tse2)
lines(fitted(mm)~tvar, data=tse2, col='red')
```



6 Acknowledgements

Credit is due to Dennis Chabot, Gabor Grothendieck, Paul Murrell, Jim Robison-Cox and Erik Jørgensen for reporting various bugs and making various suggestions to the functionality in the doBy package.

7 A simulated dataset

Consider these data:

```
library(doBy)
dd <- expand.grid(A=factor(1:3),B=factor(1:3),C=factor(1:2))
dd$y <- rnorm(nrow(dd))
dd$x <- rnorm(nrow(dd))^2
dd$z <- rnorm(nrow(dd))
head(dd,10)
```

```
   A B C        y        x        z
1  1 1 1 -0.58082 2.651835  1.19674
2  2 1 1  0.08894 0.068831 -0.29895
3  3 1 1 -0.16451 0.161791 -0.60577
4  1 2 1 -0.68817 0.523081  0.20935
5  2 2 1 -0.40846 0.206348  1.38148
6  3 2 1  1.41766 0.003133  0.07697
7  1 3 1 -1.16384 0.032023 -1.28683
8  2 3 1  1.89204 4.626368  0.47590
9  3 3 1 -0.89801 0.065836  0.06439
10 1 1 2  0.21287 0.899207  0.34100
```

Consider the additive model

$$y_i = \beta_0 + \beta^1_{A(i)} + \beta^2_{B(i)} + \beta^3_{C(i)} + e_i \tag{1}$$

where $e_i \sim N(0, \sigma^2)$. We fit this model:

```
mm <- lm(y~A+B+C, data=dd)
coef(mm)
```

```
(Intercept)          A2          A3          B2          B3          C2 
    -0.1952      0.2270     -0.2229      0.6040     -0.1909     -0.1381 
```

Notice that the parameters corresponding to the factor levels `A1`, `B1` and `C2` are set to zero to ensure identifiability of the remaining parameters.

8 Linear functions of parameters, contrasts

For a regression model with parameters $\beta = (\beta^1, \beta^2, \dots, \beta^P)$ we shall refer to a weighted sum of the form

$$\sum_j w_j \beta^j$$

as a contrast. Notice that it is common in the litterature to require that $sum_j w_j = 0$ for the sum $\sum_j w_j \beta^j$ to be called a contrast but we do not follow this tradition here.

The effect of changing the factor A from `A2` to `A3` can be found as

```
w <- c(0,-1,1,0,0,0)
sum(coef(mm)*w)
```

```
[1] -0.4499
```

The `esticon()` function provides this estimate, the standard error etc. as follows:

```
esticon(mm, w)
```

```
  beta0 Estimate Std.Error t.value DF Pr(>|t|)  Lower Upper
1     0  -0.4499    0.5902 -0.7623 12   0.4606 -1.736 0.836
```

9 Population means

Population means (sometimes also called marginal means) are in some sciences much used for reporting marginal effects (to be described below). Population means are known as lsmeans in SAS jargon. Population means is a special kind of contrasts as defined in Section 8.

The model (1) is a model for the conditional mean $\mathbb{E}(y|A, B, C)$. Sometimes one is interested in quantities like $\mathbb{E}(y|A)$. This quantity can not formally be found unless B and C are random variables such that we may find $\mathbb{E}(y|A)$ by integration.

However, suppose that A is a treatment of main interest, B is a blocking factor and C represents days on which the experiment was carried out. Then it is tempting to average $\mathbb{E}(y|A, B, C)$ over B and C (average over block and day) and think of this average as $\mathbb{E}(y|A)$.

9.1 A brute–force calculation

The population mean for $A = 1$ is

$$\beta^0 + \beta^1_{A1} + \frac{1}{3}(\beta^2_{B1} + \beta^2_{B2} + \beta^2_{B3}) + \frac{1}{2}(\beta^3_{C1} + \beta^3_{C2}) \quad (2)$$

Recall that the parameters corresponding to the factor levels `A1`, `B1` and `C2` are set to zero to ensure identifiability of the remaining parameters. Therefore we may also write the population mean for $A = 1$ as

$$\beta^0 + \frac{1}{3}(\beta^2_{B2} + \beta^2_{B3}) + \frac{1}{2}(\beta^3_{C2}) \quad (3)$$

This quantity can be estimated as:

```
 w <- c(1, 0, 0, 1/3, 1/3, 1/2)
 coef(mm)*w
```

```
(Intercept)          A2          A3          B2          B3          C2 
   -0.19517     0.00000     0.00000     0.20132    -0.06363    -0.06906 
```

```
 sum(coef(mm)*w)
```

```
[1] -0.1265
```

We may find the population mean for all three levels of A as

```
 W <- matrix(c(1, 0, 0, 1/3, 1/3, 1/2,
               1, 1, 0, 1/3, 1/3, 1/2,
               1, 0, 1, 1/3, 1/3, 1/2),nr=3, byrow=TRUE)
 W
```

```
     [,1] [,2] [,3]   [,4]   [,5] [,6]
[1,]    1    0    0 0.3333 0.3333  0.5
[2,]    1    1    0 0.3333 0.3333  0.5
[3,]    1    0    1 0.3333 0.3333  0.5
```

```
 W %*% coef(mm)
```

```
        [,1]
[1,] -0.1265
[2,]  0.1004
[3,] -0.3494
```

Notice that the matrix W is based on that the first level of A is set as the reference level. If the reference level is changed then so must W be.

9.2 Using `esticon()`

Given that one has specified W, the `esticon()` function in the `doBy` package be used for the calculations above and the function also provides standard errors, confidence limits etc:

```
 esticon(mm, W)
```

```
  beta0 Estimate Std.Error t.value DF Pr(>|t|)   Lower  Upper
1     0  -0.1265    0.4173 -0.3032 12   0.7669 -1.0358 0.7827
2     0   0.1004    0.4173  0.2406 12   0.8139 -0.8088 1.0097
3     0  -0.3494    0.4173 -0.8374 12   0.4188 -1.2587 0.5598
```

10 Using `popMatrix()` and `popMeans()`

Writing the matrix W is somewhat tedious and hence error prone. In addition, there is a potential risk of getting the wrong answer if the the reference level of a factor has been changed. The `popMatrix()` function provides an automated way of generating such matrices. The above `W` matrix is constructed by

```
 pma <- popMatrix(mm,effect='A')
 summary(pma)
```

```
     (Intercept) A2 A3     B2     B3  C2
[1,]           1  0  0 0.3333 0.3333 0.5
[2,]           1  1  0 0.3333 0.3333 0.5
[3,]           1  0  1 0.3333 0.3333 0.5
grid:
'data.frame':        3 obs. of  1 variable:
 $ A: chr  "1" "2" "3"
at:
 NULL
```

The popMeans() function is simply a wrapper around first a call to popMatrix() followed by a call to (by default) esticon():

```
pme <- popMeans(mm, effect='A')
pme
```

```
  beta0 Estimate Std.Error t.value DF Pr(>|t|)   Lower  Upper A
1     0  -0.1265    0.4173 -0.3032 12   0.7669 -1.0358 0.7827 1
2     0   0.1004    0.4173  0.2406 12   0.8139 -0.8088 1.0097 2
3     0  -0.3494    0.4173 -0.8374 12   0.4188 -1.2587 0.5598 3
```

More details about how the matrix was constructed is provided by the summary() function:

```
summary(pme)
```

```
  beta0 Estimate Std.Error t.value DF Pr(>|t|)   Lower  Upper A
1     0  -0.1265    0.4173 -0.3032 12   0.7669 -1.0358 0.7827 1
2     0   0.1004    0.4173  0.2406 12   0.8139 -0.8088 1.0097 2
3     0  -0.3494    0.4173 -0.8374 12   0.4188 -1.2587 0.5598 3
Call:
NULL
Contrast matrix:
Length  Class   Mode
     0   NULL   NULL
```

The effect argument requires to calculate the population means for each level of A aggregating across the levels of the other variables in the data.

Likewise we may do:

```
popMatrix(mm,effect=c('A','C'))
```

```
     (Intercept) A2 A3     B2     B3 C2
[1,]           1  0  0 0.3333 0.3333  0
[2,]           1  1  0 0.3333 0.3333  0
[3,]           1  0  1 0.3333 0.3333  0
[4,]           1  0  0 0.3333 0.3333  1
[5,]           1  1  0 0.3333 0.3333  1
[6,]           1  0  1 0.3333 0.3333  1
```

This gives the matrix for calculating the estimate for each combination of A and C when averaging over B. Consequently

```
popMeans(mm)
```

```
  beta0 Estimate Std.Error t.value DF Pr(>|t|)   Lower  Upper
1     0  -0.1252    0.2409 -0.5196 12   0.6128 -0.6501 0.3998
```

gives the “total average”.

10.1 Using the `at` argument

We may be interested in finding the population means at all levels of A but only at $C = 1$. This is obtained by using the `at` argument:

```
popMatrix(mm,effect='A', at=list(C='1'))
```

```
     (Intercept) A2 A3     B2     B3 C2
[1,]           1  0  0 0.3333 0.3333  0
[2,]           1  1  0 0.3333 0.3333  0
[3,]           1  0  1 0.3333 0.3333  0
```

Notice here that average is only taken over B. Another way of creating the population means at all levels of (A, C) is therefore

```
popMatrix(mm,effect='A', at=list(C=c('1','2')))
```

```
     (Intercept) A2 A3     B2     B3 C2
[1,]           1  0  0 0.3333 0.3333  0
[2,]           1  1  0 0.3333 0.3333  0
[3,]           1  0  1 0.3333 0.3333  0
[4,]           1  0  0 0.3333 0.3333  1
[5,]           1  1  0 0.3333 0.3333  1
[6,]           1  0  1 0.3333 0.3333  1
```

We may have several variables in the `at` argument:

```
popMatrix(mm,effect='A', at=list(C=c('1','2'), B='1'))
```

```
     (Intercept) A2 A3 B2 B3 C2
[1,]           1  0  0  0  0  0
[2,]           1  1  0  0  0  0
[3,]           1  0  1  0  0  0
[4,]           1  0  0  0  0  1
[5,]           1  1  0  0  0  1
[6,]           1  0  1  0  0  1
```

10.2 Ambiguous specification when using the `effect` and `at` arguments

There is room for an ambiguous specification if a variable appears in both the `effect` and the `at` argument, such as

```
popMatrix(mm,effect=c('A','C'), at=list(C='1'))
```

```
     (Intercept) A2 A3     B2     B3 C2
[1,]           1  0  0 0.3333 0.3333  0
[2,]           1  1  0 0.3333 0.3333  0
[3,]           1  0  1 0.3333 0.3333  0
```

This ambiguity is due to the fact that the `effect` argument asks for the populations means at all levels of the variables but the `at` chooses only specific levels.

This ambiguity is resolved as follows: Any variable in the `at` argument is removed from the `effect` argument such as the statement above is equivalent to

```
popMatrix(mm,effect='A', at=list(C='1'))
```

10.3 Using covariates

Next consider the model where a covariate is included:

```
mm2 <- lm(y~A+B+C+C:x, data=dd)
coef(mm2)
```

```
(Intercept)          A2          A3          B2          B3          C2
   -0.56007     0.15366     0.04488     0.57520    -0.44357    -0.03179
       C1:x        C2:x
    0.42509     0.16052
```

In this case we get

```
popMatrix(mm2,effect='A', at=list(C='1'))
```

```
     (Intercept) A2 A3     B2     B3 C2  C1:x C2:x
[1,]           1  0  0 0.3333 0.3333  0 1.359    0
[2,]           1  1  0 0.3333 0.3333  0 1.359    0
[3,]           1  0  1 0.3333 0.3333  0 1.359    0
```

Above, x has been replaced by its average and that is the general rule for models including covariates. However we may use the `at` argument to ask for calculation of the population mean at some user-specified value of x, say 12:

```
popMatrix(mm2,effect='A', at=list(C='1',x=12))
```

```
     (Intercept) A2 A3     B2     B3 C2 C1:x C2:x
[1,]           1  0  0 0.3333 0.3333  0   12    0
[2,]           1  1  0 0.3333 0.3333  0   12    0
[3,]           1  0  1 0.3333 0.3333  0   12    0
```

10.4 Using transformed covariates

Next consider the model where a transformation of a covariate is included:

```
mm3 <- lm(y~A+B+C+C:log(x), data=dd)
coef(mm3)
```

```
(Intercept)          A2          A3          B2          B3          C2
   -0.10605     0.26942    -0.08983     0.42560    -0.31605    -0.17396
  C1:log(x)   C2:log(x)
    0.02590     0.20493
```

In this case we can not use `popMatrix()` (and hence `popMeans()` directly. Instead we have first to generate a new variable, say `log.x`, with `log.x`= log(x), in the data and then proceed as

```
dd <- transform(dd, log.x = log(x))
mm3 <- lm(y~A+B+C+C:log.x, data=dd)
popMatrix(mm3,effect='A', at=list(C='1'))
```

```
     (Intercept) A2 A3     B2     B3 C2 C1:log.x C2:log.x
[1,]           1  0  0 0.3333 0.3333  0  -0.9228        0
[2,]           1  1  0 0.3333 0.3333  0  -0.9228        0
[3,]           1  0  1 0.3333 0.3333  0  -0.9228        0
```

11 The `engine` argument of `popMeans()`

The `popMatrix()`is a function to generate a linear tranformation matrix of the model parameters with emphasis on constructing such matrices for population means. `popMeans()` invokes by default the `esticon()` function on this linear transformation matrix for calculating parameter estimates and confidecne intervals. A similar function to `esticon()` is the `glht` function of the `multcomp` package.

The `glht()` function can be chosen via the `engine` argument of `popMeans()`:

```
  library(multcomp)
 g<-popMeans(mm,effect='A', at=list(C='1'),engine="glht")
 g

         General Linear Hypotheses

Linear Hypotheses:
       Estimate
1 == 0  -0.0575
2 == 0   0.1695
3 == 0  -0.2804
```

This allows to apply the methods available on the `glht` object like

```
 summary(g,test=univariate())

         Simultaneous Tests for General Linear Hypotheses

Fit: lm(formula = y ~ A + B + C, data = dd)

Linear Hypotheses:
       Estimate Std. Error t value Pr(>|t|)
1 == 0  -0.0575     0.4819   -0.12     0.91
2 == 0   0.1695     0.4819    0.35     0.73
3 == 0  -0.2804     0.4819   -0.58     0.57
(Univariate p values reported)

 confint(g,calpha=univariate_calpha())

         Simultaneous Confidence Intervals

Fit: lm(formula = y ~ A + B + C, data = dd)

Quantile = 2.179
95% confidence level


Linear Hypotheses:
       Estimate lwr     upr
1 == 0 -0.0575  -1.1074  0.9924
2 == 0  0.1695  -0.8804  1.2194
3 == 0 -0.2804  -1.3303  0.7695
```

which yield the same results as the `esticon()` function.

By default the functions will adjust the tests and confidence intervals for multiplicity

```
 summary(g)
```

```
         Simultaneous Tests for General Linear Hypotheses

Fit: lm(formula = y ~ A + B + C, data = dd)

Linear Hypotheses:
       Estimate Std. Error t value Pr(>|t|)
1 == 0  -0.0575     0.4819   -0.12     1.00
2 == 0   0.1695     0.4819    0.35     0.98
3 == 0  -0.2804     0.4819   -0.58     0.91
(Adjusted p values reported -- single-step method)
```

```
 confint(g)
```

```
         Simultaneous Confidence Intervals

Fit: lm(formula = y ~ A + B + C, data = dd)

Quantile = 2.734
95% family-wise confidence level


Linear Hypotheses:
       Estimate lwr     upr
1 == 0 -0.0575  -1.3750  1.2600
2 == 0  0.1695  -1.1480  1.4870
3 == 0 -0.2804  -1.5979  1.0371
```

12 Discussion

13 Acknowledgements

This work was supported by the Danish National Advanced Technology Foundation through the ILSORM project.