Groupwise computations and other utilities in the `doBy` package

Søren Højsgaard

doBy version 4.6.11 as of 2021-07-13

Contents

1 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.

2 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.

3 Groupwise computations

3.1 The `summaryBy` function

The `summaryBy` function is used for calculating quantities like "the mean and variance of numerical variables x and y for each combination of two factors A and B". Notice: A functionality similar to `summaryBy` is provided by `aggregate()` from base R.

```
myfun1 <- function(x){c(m=mean(x), s=sd(x))}
summaryBy(cbind(conc, uptake, lu=log(uptake)) ~ Plant,
          data=CO2, FUN=myfun1)
```

```
##   Plant conc.m conc.s uptake.m uptake.s  lu.m   lu.s
## 1   Qn1    435  317.7    33.23    8.215 3.467 0.3189
## 2   Qc1    435  317.7    29.97    8.335 3.356 0.3446
## 3   Mn1    435  317.7    26.40    8.694 3.209 0.4234
## 4   Mc1    435  317.7    18.00    4.119 2.864 0.2622
```

A simpler call is

```
summaryBy(conc ~ Plant, data=CO2, FUN=mean)
```

Instead of formula we may specify a list containing the left hand side and the right hand side of a formula[1] but that is possible only for variables already in the dataframe:

```
## Will fail because of log(uptake)
## summaryBy(list(c("conc", "uptake", "log(uptake)"), "Plant"),
##           data=CO2, FUN=myfun1)
## Works
summaryBy(list(c("conc", "uptake"), "Plant"),
          data=CO2, FUN=myfun1)
```

3.2 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:

```
x1 <- orderBy(~ Temp + Month, data=airquality)
head(x1)
```

```
##    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 can do

[1]This is a feature of `summaryBy` and it does not work with `aggregate`.

```
x2 <- orderBy(~ - Temp + Month, data=airquality)
```

An alternative form is:

```
x3 <- orderBy(c("Temp", "Month"), data=airquality)
x4 <- orderBy(c("-Temp", "Month"), data=airquality)
```

3.3 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)
lapply(x, head, 4)

## $'5'
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
##
## $'6'
##    Ozone Solar.R Wind Temp Month Day
## 32    NA     286  8.6   78     6   1
## 33    NA     287  9.7   74     6   2
## 34    NA     242 16.1   67     6   3
## 35    NA     186  9.2   84     6   4

attributes(x)

## $names
## [1] "5" "6"
##
## $groupid
##   Month
## 1     5
## 2     6
##
## $idxvec
## $idxvec$'5'
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31
##
```

```
## $idxvec$'6'
##  [1] 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
## [26] 57 58 59 60 61
##
##
## $grps
##  [1] "5" "5" "5" "5" "5" "5" "5" "5" "5" "5" "5" "5" "5" "5" "5" "5" "5" "5" "5"
## [20] "5" "5" "5" "5" "5" "5" "5" "5" "5" "5" "5" "5" "6" "6" "6" "6" "6" "6" "6"
## [39] "6" "6" "6" "6" "6" "6" "6" "6" "6" "6" "6" "6" "6" "6" "6" "6" "6" "6" "6"
## [58] "6" "6" "6" "6"
##
## $class
## [1] "splitByData" "list"
```

An alternative call is

```
splitBy("Month", data=airquality)
```

3.4 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:

```
x <- subsetBy(~Month, subset=Wind > mean(Wind), data=airquality)
head(x)
```

```
##      Ozone Solar.R Wind Temp Month Day
## 5.3     12     149 12.6   74     5   3
## 5.5     NA      NA 14.3   56     5   5
## 5.6     28      NA 14.9   66     5   6
## 5.8     19      99 13.8   59     5   8
## 5.9      8      19 20.1   61     5   9
## 5.15    18      65 13.2   58     5  15
```

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

3.5 The `transformBy` function

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

```
x <- transformBy(~Month, data=airquality,
                 minW=min(Wind), maxW=max(Wind),
                  chg = diff(range(Wind)))
head(x)
```

```
##   Ozone Solar.R Wind Temp Month Day minW maxW  chg
## 1    41     190  7.4   67     5   1  5.7 20.1 14.4
## 2    36     118  8.0   72     5   2  5.7 20.1 14.4
## 3    12     149 12.6   74     5   3  5.7 20.1 14.4
## 4    18     313 11.5   62     5   4  5.7 20.1 14.4
## 5    NA      NA 14.3   56     5   5  5.7 20.1 14.4
## 6    28      NA 14.9   66     5   6  5.7 20.1 14.4
```

Alternative forms:

```
x <- transformBy("Month", data=airquality,
                  minW=min(Wind), maxW=max(Wind),
                  chg = diff(range(Wind)))
```

4 Miscellaneous utilities

4.1 `restrict_fun()`: Restrict a functions domain

The `restrict_fun` function can restrict the domain of a function. There are two approaches: 1) Store the restricted arguments in an auxillary environment and 2) substitute the restricted arguments into the function.

4.1.1 Using an auxillary environment

```
f1  <- function(a, b=2, c=4){a + b + c}
f1_ <- restrict_fun(f1, list(a=1, b=7))
class(f1_)
```

```
## [1] "scaffold"
```

```
f1_
```

```
## function (c = 4)
## {
##     args <- arg_getter()
##     do.call(fun, args)
## }
## <environment: 0x55a1b38d8648>
```

```
f1_()
```

```
## [1] 12
```

The restricted values are stored in an extra environment in the scaffold function:

```
get_restrictions(f1_)
```

```
## $a
## [1] 1
##
## $b
## [1] 7
```

```
## attr(f1_, "arg_env")£args ## Same result
```

The original function is stored in the scaffold functions environment:

```
get_fun(f1_)
```

```
## function(a, b=2, c=4){a + b + c}
```

```
## environment(f1_)£fun ## Same result
```

Similarly

```
rnorm5 <- restrict_fun(rnorm, list(n=5))
rnorm5()
```

```
## [1]  2.0005  1.3873  1.7214 -0.6100 -0.4435
```

4.1.2 Substitute restricted values into function

With substitution, it is clear what is happening:

```
f1s_ <- restrict_fun_sub(f1, list(a=1, b=7))
f1s_
```

```
## function (c = 4)
## {
##     1 + 7 + c
## }
```

```
f1s_()
```

```
## [1] 12
```

However, absurdities can arise:

```
f2  <- function(a) {a <- a + 1; a}
## Notice that the following is absurd
```

```
f2s_ <- restrict_fun_sub(f2, list(a = 10))
f2s_

## function ()
## {
##     10 <- 10 + 1
##     10
## }

# do not run: f2s_()
try(f2s_())

## Error in 10 <- 10 + 1 : invalid (do_set) left-hand side to assignment

## Using the environment approch, the result makes sense
f2_ <- restrict_fun(f2, list(a = 10))
f2_

## function ()
## {
##     args <- arg_getter()
##     do.call(fun, args)
## }
## <environment: 0x55a1b712db38>

f2_()

## [1] 11
```

4.1.3 Example: Benchmarking

Consider a simple task: Adding integers from 1 to n. A naive implementation is

```
sum2n <- function(n) {
  s <- 0
  for (i in 1:n) s <- s + i
  s
}
sum2n(10)

## [1] 55
```

We can benchmark timing for different values of n as

```
library(microbenchmark)
microbenchmark(
  sum2n(10), sum2n(100), sum2n(1000), sum2n(10000),
  times=5
)

## Unit: nanoseconds
##           expr    min     lq   mean median     uq    max neval cld
##      sum2n(10)    601    705   1664    720    951   5343     5 a
##     sum2n(100)   2355   2384   2416   2387   2473   2479     5 a
##    sum2n(1000)  18163  18277  24695  18287  22113  46635     5  b
##   sum2n(10000) 173900 174516 184543 176696 188509 209092     5   c
```

It is tedious (and hence error prone) to write these function calls. Instead we can do:

```
n.vec  <- c(10, 100, 1000, 10000)
fn.list <- lapply(n.vec, function(a.) restrict_fun(sum2n, list(n=a.)))
fn.list %>% length

## [1] 4
```

Each element is a function (a scaffold object, to be precise) and we can evaluate all functions as:

```
fn.list[[1]]

## function ()
## {
##     args <- arg_getter()
##     do.call(fun, args)
## }
## <environment: 0x55a1b5be69a8>

sapply(fn.list, function(f) do.call(f, list()))

## [1]       55     5050   500500 50005000
```

To use the list of functions in connection with microbenchmark, we can do the following (which is eqully tedious):

```
microbenchmark(
  fn.list[[1]](), fn.list[[2]](), fn.list[[3]](), fn.list[[4]](),
  times=5
)
```

This can be automatized as follows: We bquote all functions

```
dobq <- function(fnlist){
   lapply(fnlist, function(g) bquote(.(g)()))
}
cl.list <- dobq(fn.list)
cl.list[[1]]

## (function ()
## {
##     args <- arg_getter()
##     do.call(fun, args)
## })()
```

All calls can be evaluated as

```
sapply(cl.list, eval)

## [1]       55     5050   500500 50005000
```

To use microbenchmark we must name the elements of the list:

```
names(cl.list) <- n.vec
microbenchmark(
  list=cl.list,
  times=5
)

## Unit: microseconds
##   expr     min      lq    mean  median      uq     max neval cld
##     10   4.261   4.651   4.964   4.663   4.886   6.358     5 a
##    100   6.057   6.104   6.914   6.109   6.110  10.192     5 a
##   1000  21.851  21.891  22.346  21.939  22.107  23.944     5  b
##  10000 177.588 177.594 183.618 177.826 180.636 204.448     5   c
```

4.2 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
```

4.3 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
```

4.4 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
```

4.5 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"
```

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

```
head(renameCol(CO2, 1:2, c("plant_", "type_")))

##   plant_ type_ 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("plant_", "type_")))

##   plant_ type_ 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
```

4.7 Time since an event - `timeSinceEvent()`

Consider the vector

```
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)
```

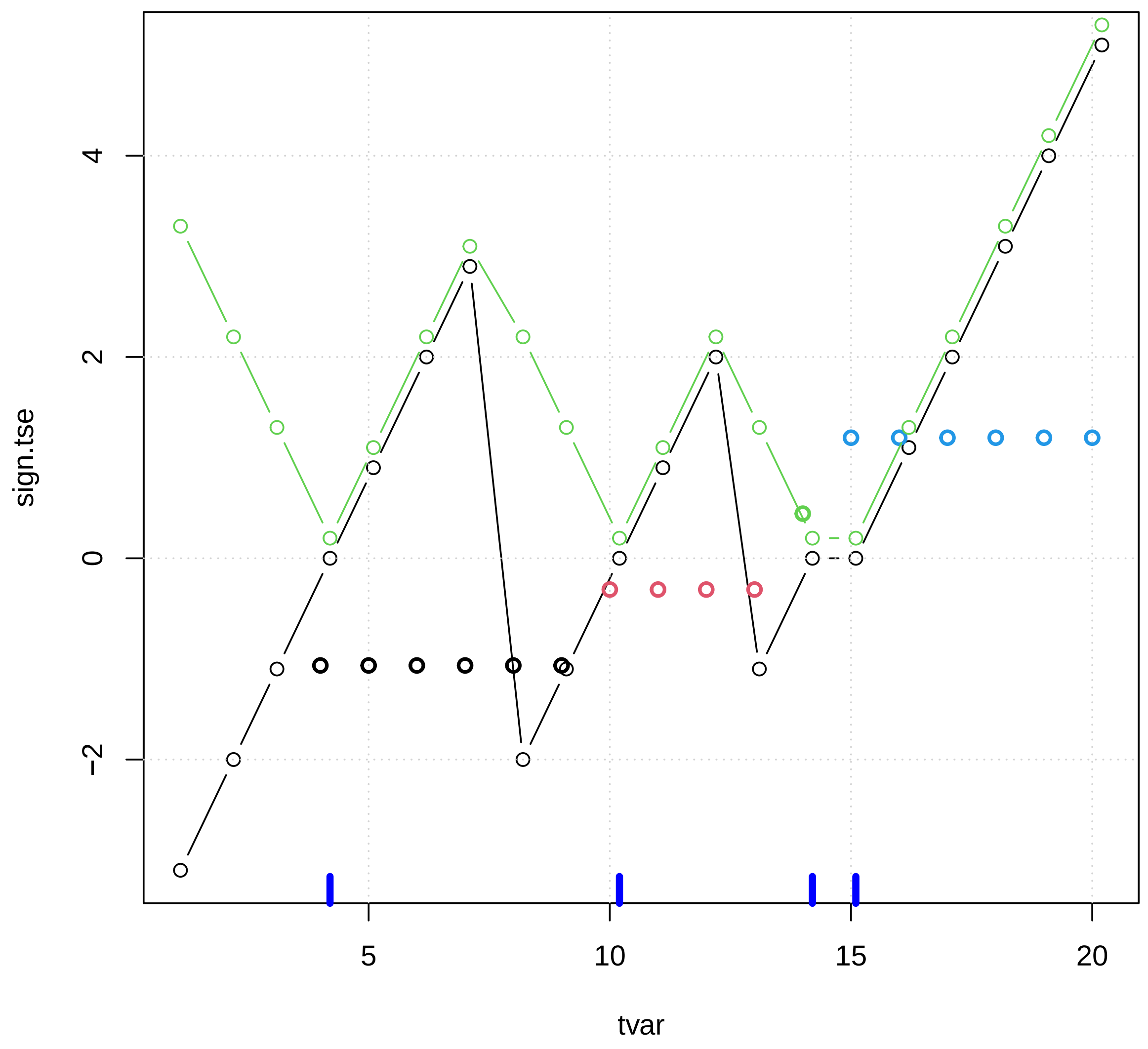
Now we find time since event as

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

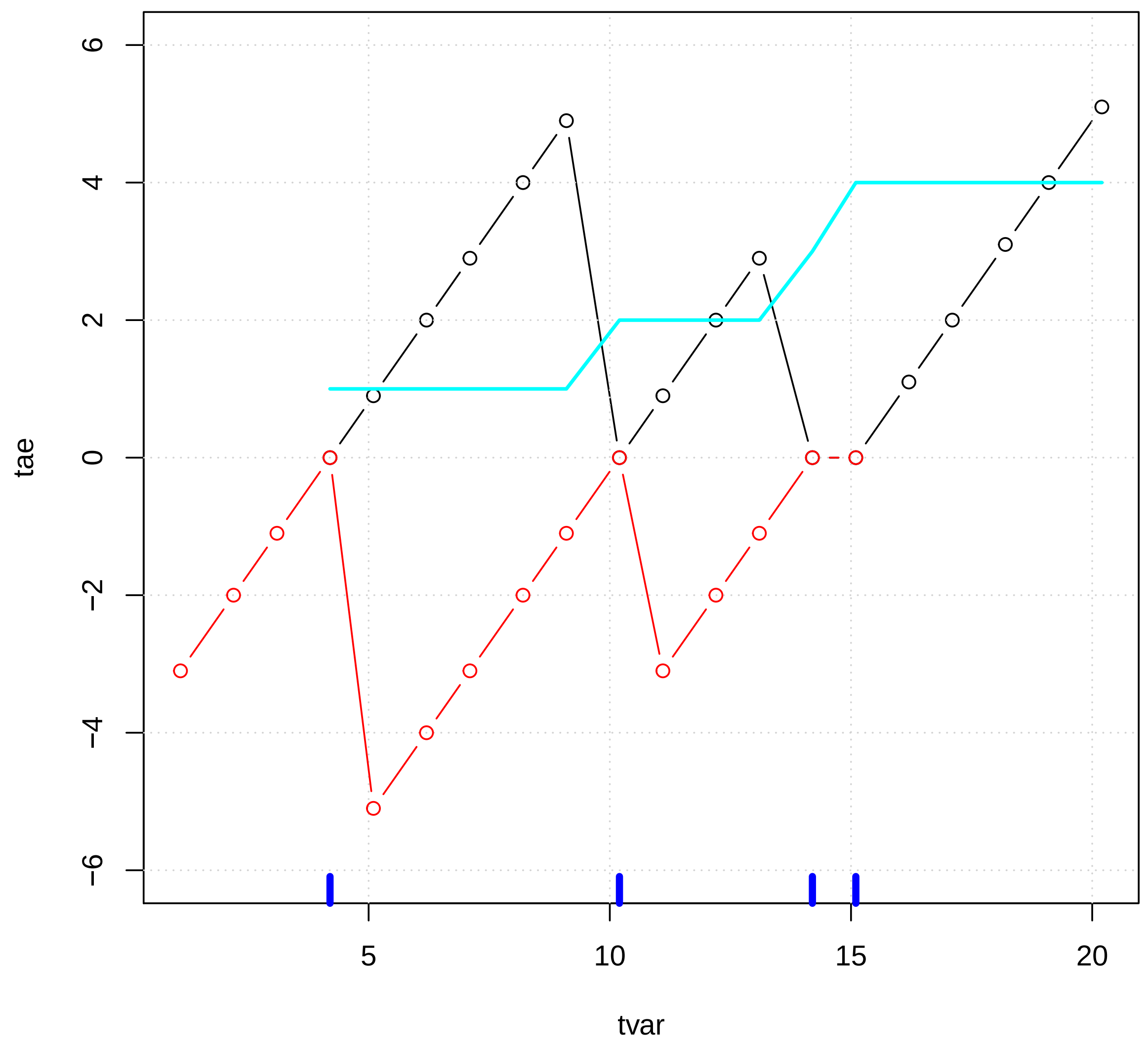
The output reads as follows:

- `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.

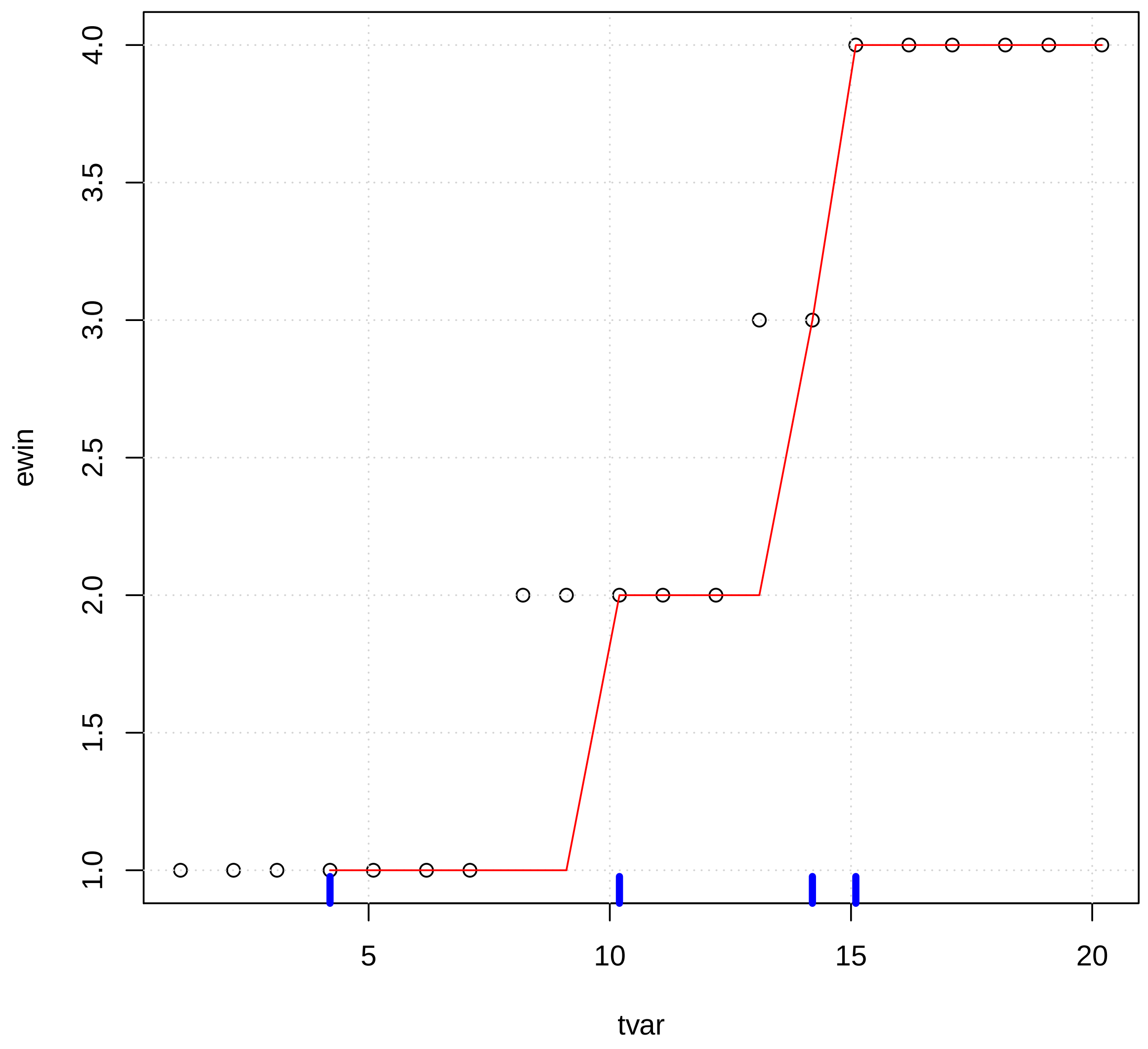
```
plot(sign.tse ~ tvar, data=tse, type="b")
grid()
rug(tse$tvar[tse$yvar == 1], col="blue",lwd=4)
points(scale(tse$run), col=tse$run, lwd=2)
lines(abs.tse + .2 ~ tvar, data=tse, type="b",col=3)
```



```
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")
```



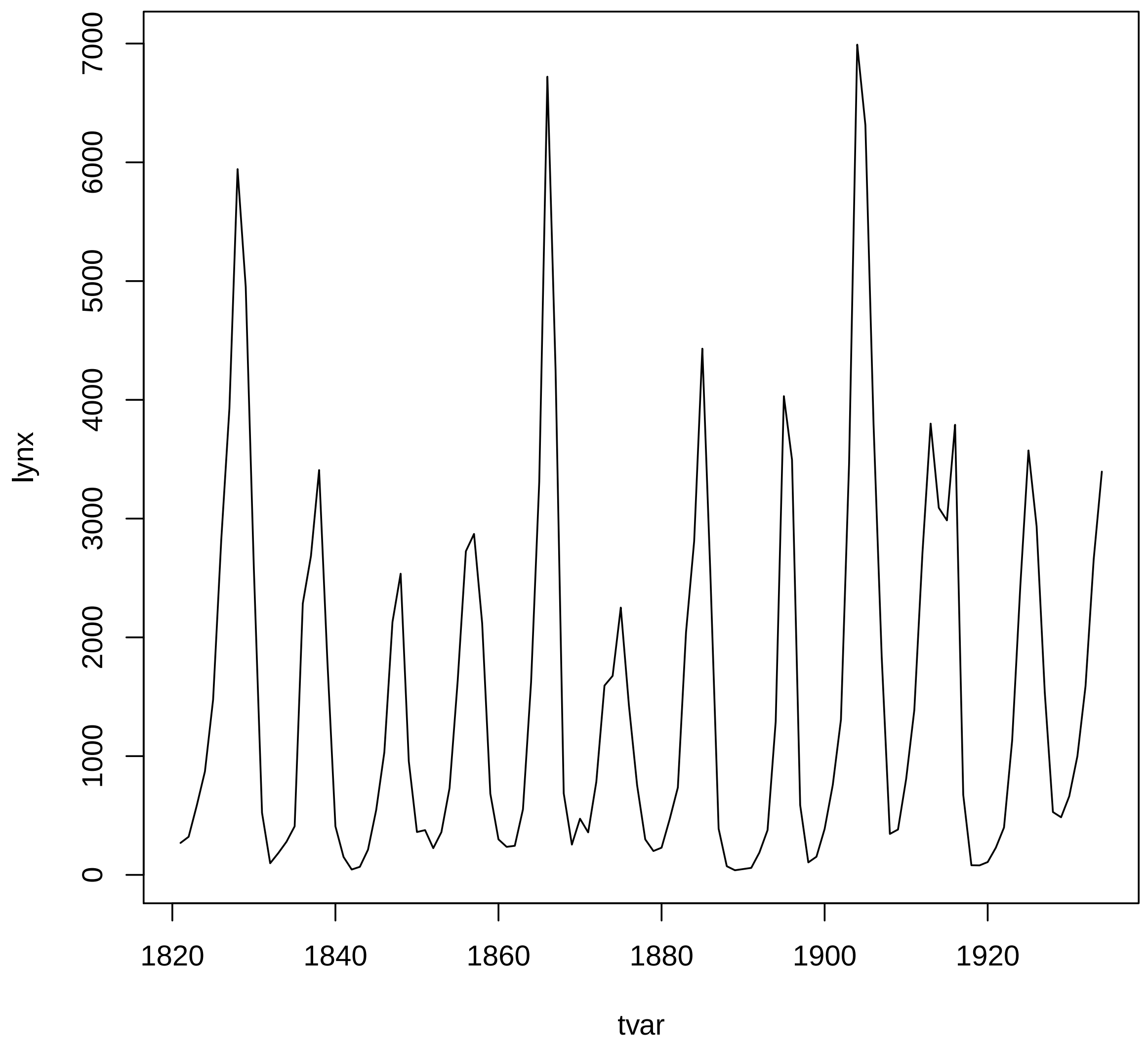
We may now find times for which time since an event is at most 1 as

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

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

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

Consider the `lynx` data:

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



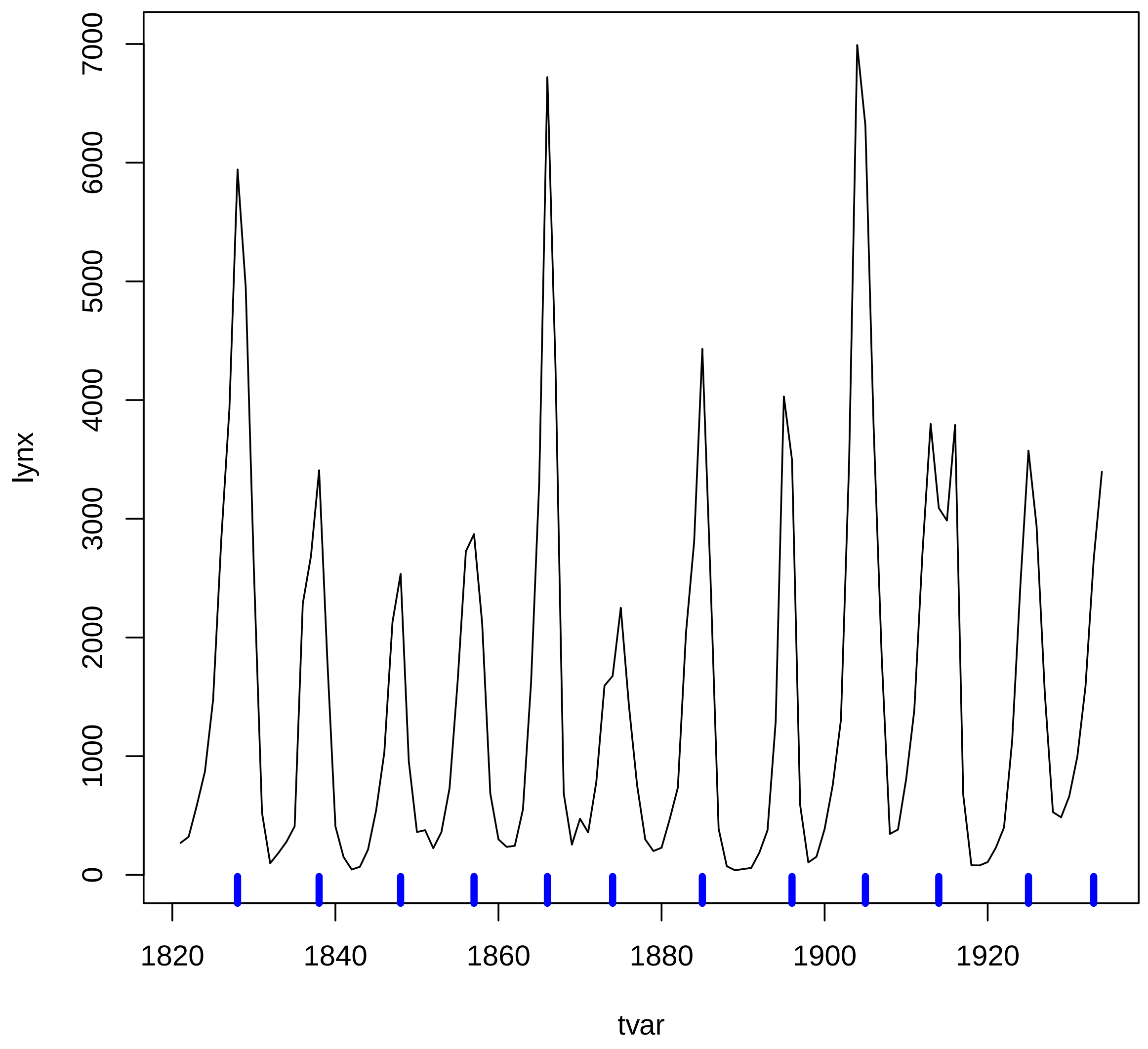
Suppose we want to estimate the cycle lengths. One way of doing this is as follows:

```r
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)
```

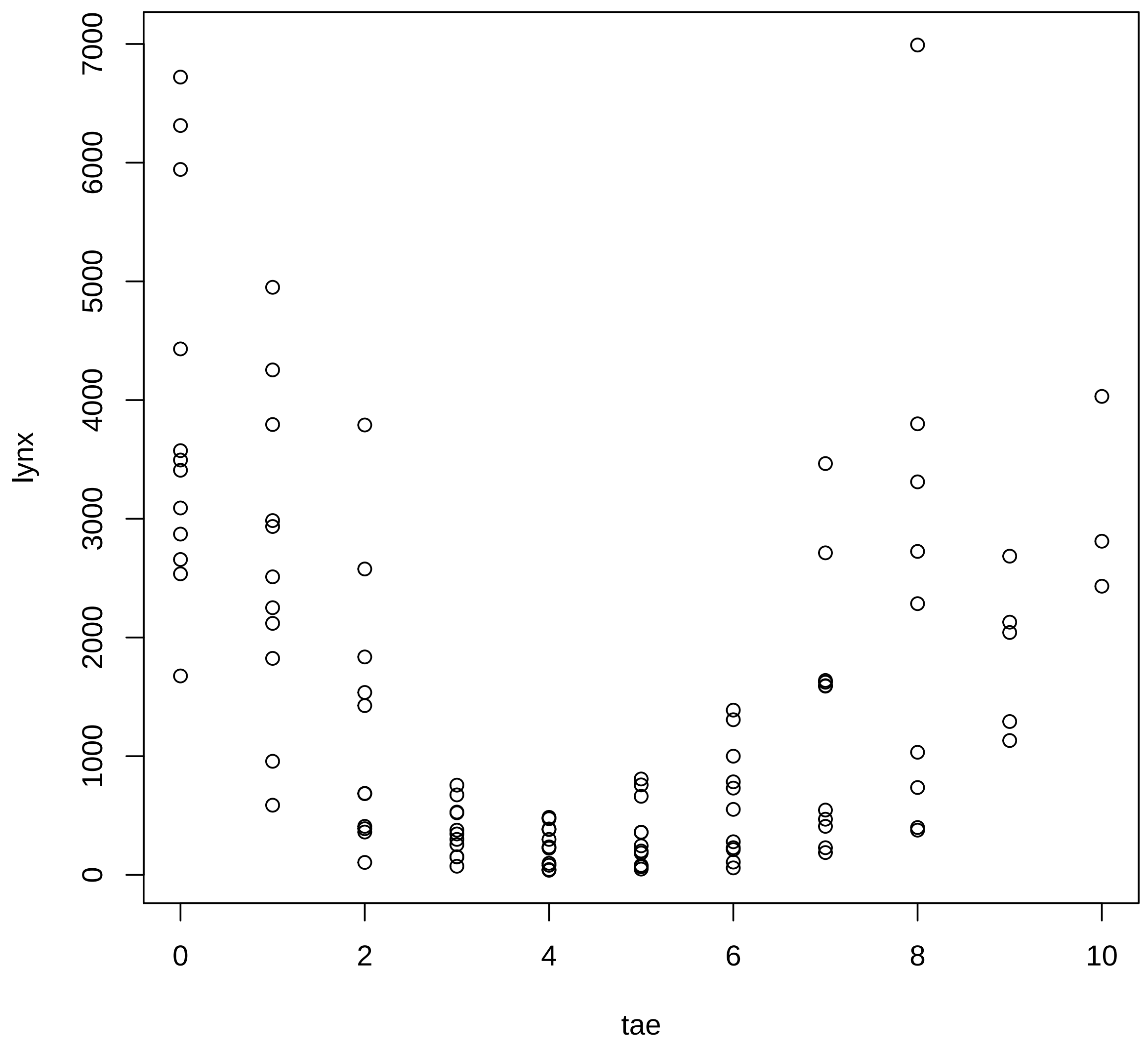
```
##    yvar tvar abs.tse sign.tse ewin run tae tbe
## 1     0 1821       7       -7    1  NA  NA  -7
## 2     0 1822       6       -6    1  NA  NA  -6
## 3     0 1823       5       -5    1  NA  NA  -5
## 4     0 1824       4       -4    1  NA  NA  -4
## 5     0 1825       3       -3    1  NA  NA  -3
## 6     0 1826       2       -2    1  NA  NA  -2
## 7     0 1827       1       -1    1  NA  NA  -1
## 8     1 1828       0        0    1   1   0   0
## 9     0 1829       1        1    1   1   1  -9
## 10    0 1830       2        2    1   1   2  -8
## 11    0 1831       3        3    1   1   3  -7
## 12    0 1832       4        4    1   1   4  -6
## 13    0 1833       5        5    1   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    2   2   0   0
## 19    0 1839       1        1    2   2   1  -9
## 20    0 1840       2        2    2   2   2  -8
```

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

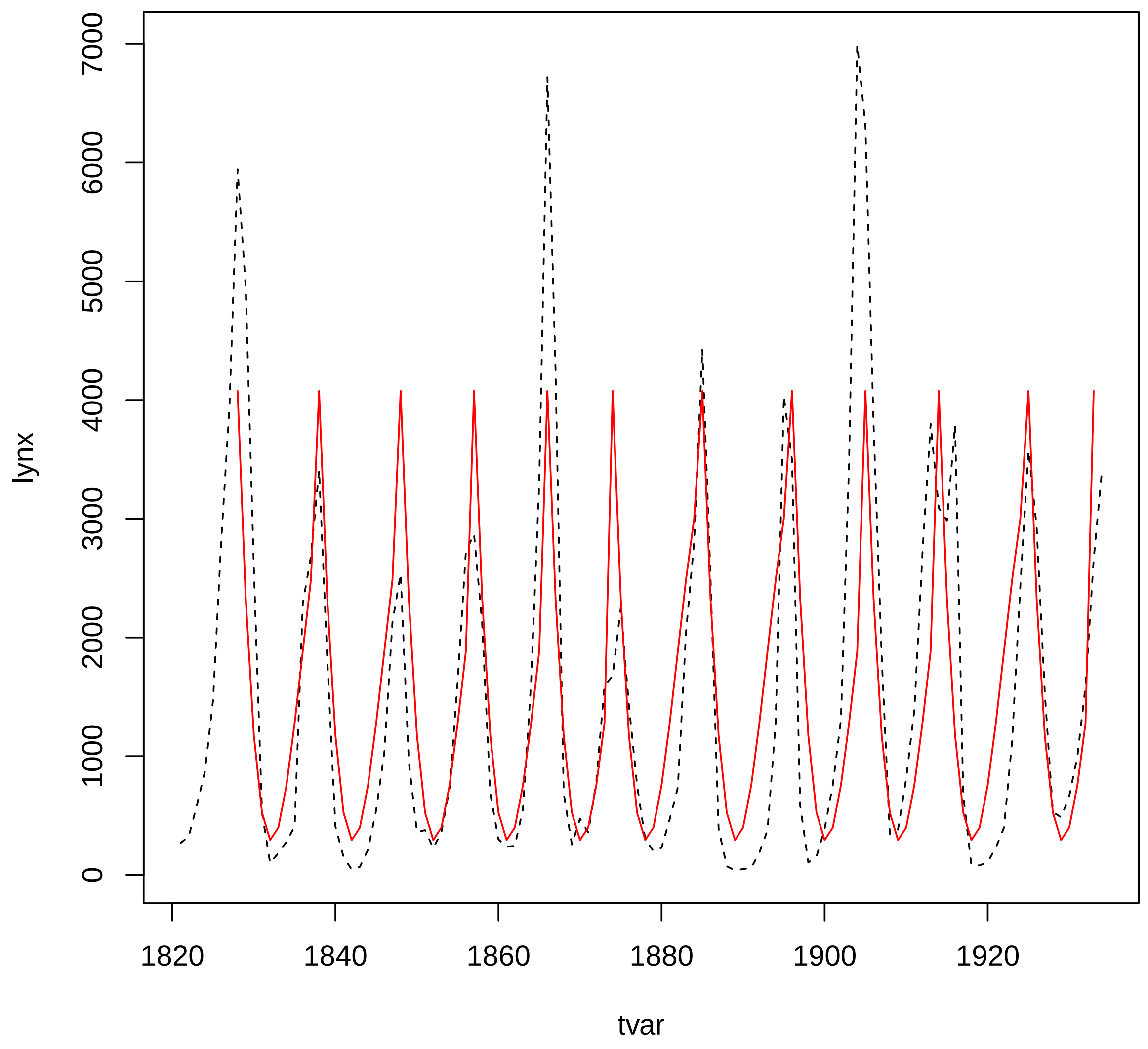
```
len1 <- tapply(tse$ewin, tse$ewin, length)
len2 <- tapply(tse$run, tse$run, length)
c(median(len1), median(len2), mean(len1), mean(len2))

## [1] 9.500 9.000 9.500 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")
```



5 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.