

The doBy package for data handling, linear estimates and LS-means

by Søren Højsgaard

Abstract The **doBy** is one of several general utility packages on CRAN. We illustrate two main features of the package: The ability to making groupwise computations and the ability to compute linear estimates, contrasts and least-squares means.

Introduction

The **doBy** package (Højsgaard and Halekoh, 2020) grew out of a need to calculate groupwise summary statistics (much in the spirit of PROC SUMMARY of the SAS system, (SAS Institute Inc., 2020)). The package first appeared on CRAN, <https://cran-r-project.org>, in 2006. The name **doBy** comes from the need to **do** some computations on data which is stratified **By** the value of some variables. Today the package contains many additional utilities. In this paper we focus 1) on the “doing by”-functions and 2) on functions related to linear estimates and contrasts (in particular LS-means).

Related functionality

When it comes to data handling, **doBy** is nowhere nearly as powerful as more contemporary packages, such as those in the **tidyverse** eco system, (Wickham et al., 2019). The aggregate function in base R provides functionality similar to **doBy**s `summaryBy` function. Another package to be mentioned in this connection is **data.table**, Dowle and Srinivasan (2019). On the other hand, **doBy** is based on classical data structures that are unlikely to undergo sudden changes. There is one exception to this, though: The data handling functions work on tibble’s, from **tibble** Müller and Wickham (2020). In relation to linear estimates, the **multcomp** package (Hothorn et al., 2008) deserves mention, and the **lsmeans** package (Lenth, 2016) provides facilities for computing LS-means.

It can be hypothesized that the data handling functions in **doBy** remain appealing to a group of users because of their simplicity.

A working dataset - the C02 data

The C02 data frame comes from an experiment on the cold tolerance of the grass species *Echinochloa crus-galli*. Type is a factor with levels Quebec or Mississippi giving the origin of the plant. Treatment is a factor levels nonchilled or chilled. Data is balanced with respect to these two factors. However, illustrated certain points we exclude a few rows of data to make data imbalanced. To limit the amount of output we modify names and levels of variables as follows

```
data(C02)
C02 <- within(C02, {
  Treat = Treatment; Treatment = NULL
  levels(Treat) = c("nchil", "chil"); levels(Type) = c("Que", "Mis")
})
C02 <- subset(C02, Plant %in% c("Qn1", "Qc1", "Mn1", "Mc1"))
C02 <- C02[-(1:3),]
xtabs(~Treat+Type, data=C02)

#>      Type
#> Treat  Que Mis
#> nchil   4   7
#>  chil   7   7

head(C02, 4)

#>   Plant Type conc uptake Treat
#> 4  Qn1  Que  350   37.2 nchil
#> 5  Qn1  Que  500   35.3 nchil
#> 6  Qn1  Que  675   39.2 nchil
#> 7  Qn1  Que 1000   39.7 nchil
```

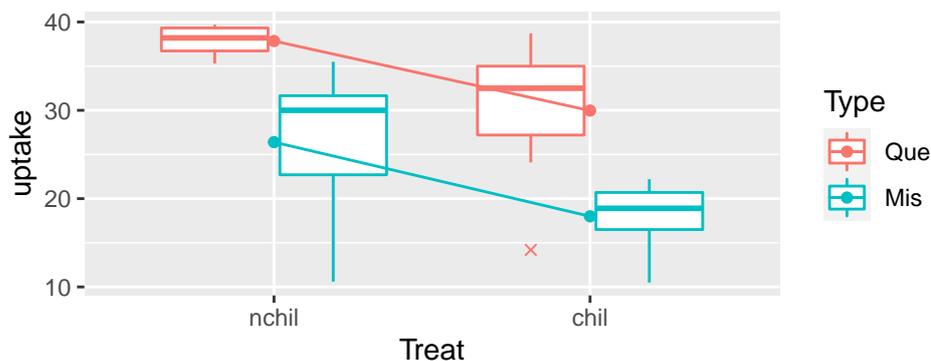


Figure 1: Interaction plot for the CO2 data. Boxplot outliers are crosses. The plot suggests additivity between Treat and Type.

Functions related to groupwise computations

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, but `summaryBy` offers additional features.

```
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 631.2 279.4 37.85 2.014 3.633 0.05375
#> 2 Qc1 435.0 317.7 29.97 8.335 3.356 0.34457
#> 3 Mn1 435.0 317.7 26.40 8.694 3.209 0.42341
#> 4 Mc1 435.0 317.7 18.00 4.119 2.864 0.26219
```

The convention is that variables that do not appear in the dataframe (e.g. `log(uptake)`) must be named (here as `lu`). Various shortcuts are available, e.g. the following, where left hand side dot refers to *all numeric variables* while the right hand side dot refers to *all factor variables*. Writing 1 on the right hand side leads to computing over the entire dataset:

```
summaryBy(. ~ ., data=CO2, FUN=myfun1)

#> Plant Type Treat conc.m conc.s uptake.m uptake.s
#> 1 Qn1 Que nchil 631.2 279.4 37.85 2.014
#> 2 Qc1 Que chil 435.0 317.7 29.97 8.335
#> 3 Mn1 Mis nchil 435.0 317.7 26.40 8.694
#> 4 Mc1 Mis chil 435.0 317.7 18.00 4.119
```

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

```
#> conc.m conc.s uptake.m uptake.s
#> 1 466.4 301.4 26.88 9.323
```

Specifications as formulas and lists

We shall refer to all the functions for groupwise computations etc. as the *By-functions*. The convention for the *By-functions* is that a two sided formula like can be written in two ways:

```
cbind(x, y) ~ A + B
list(c("x", "y"), c("A", "B"))
```

Some *By-functions* only take a right hand sided formula as input. Such a formula can also be written in two ways:

```
~ A + B
c("A", "B")
```

The list-form / vector-form is especially useful if a function is invoked programatically. Hence the calls to `summaryBy` above can also be made as

```
summaryBy(list(c("conc", "uptake", "lu=log(uptake)"), "Plant"), data=C02, FUN=myfun1)
summaryBy(list(".", "."), data=C02, FUN=myfun1)
summaryBy(list(".", "1"), data=C02, FUN=myfun1)
```

Using the pipe operator

The `summaryBy` function has a counterpart called `summary_by`. The difference is that a formula is the first argument to the former function while a dataframe (or a tibble) is the first argument to the latter. The same applies to the other By-functions. This allows for elegant use of the pipe operator `%>%` from [magrittr](#), ([Bache and Wickham, 2014](#)):

```
C02 %>% summary_by(cbind(conc, uptake) ~ Plant + Type, FUN=myfun1) -> newdat
newdat
```

```
#>   Plant Type conc.m conc.s uptake.m uptake.s
#> 1   Qn1  Que  631.2  279.4   37.85   2.014
#> 2   Qc1  Que  435.0  317.7   29.97   8.335
#> 3   Mn1  Mis  435.0  317.7   26.40   8.694
#> 4   Mc1  Mis  435.0  317.7   18.00   4.119
```

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 C02 data by increasing values of `conc` and decreasing value of `uptake` (within `conc`):

```
x1 <- orderBy(~ conc - uptake, data=C02)
head(x1)
```

```
#>   Plant Type conc uptake Treat
#> 22   Qc1  Que   95  14.2  chil
#> 43   Mn1  Mis   95  10.6 nchil
#> 64   Mc1  Mis   95  10.5  chil
#> 23   Qc1  Que  175  24.1  chil
#> 44   Mn1  Mis  175  19.2 nchil
#> 65   Mc1  Mis  175  14.9  chil
```

The splitBy function

Suppose we want to split C02 into a list of dataframes:

```
x1 <- splitBy(~ Plant + Type, data=C02)
x1
```

```
#>   listentry Plant Type
#> 1   Qn1|Que   Qn1  Que
#> 2   Qc1|Que   Qc1  Que
#> 3   Mn1|Mis   Mn1  Mis
#> 4   Mc1|Mis   Mc1  Mis
```

The result is a list (with a few additional attributes):

```
lapply(x1, head, 2)

#> $`Qn1|Que`
#>   Plant Type conc uptake Treat
#> 4   Qn1  Que  350   37.2 nchil
#> 5   Qn1  Que  500   35.3 nchil
#>
#> $`Qc1|Que`
#>   Plant Type conc uptake Treat
#> 22   Qc1  Que   95  14.2  chil
#> 23   Qc1  Que  175  24.1  chil
```

```
#>
#> `$Mn1|Mis`
#>   Plant Type conc uptake Treat
#> 43  Mn1  Mis   95   10.6 nchil
#> 44  Mn1  Mis  175   19.2 nchil
#>
#> `$Mc1|Mis`
#>   Plant Type conc uptake Treat
#> 64  Mc1  Mis   95   10.5 chil
#> 65  Mc1  Mis  175   14.9 chil
```

The subsetBy function

Suppose we want to select those rows within each treatment for which the uptake is larger than 75% quantile of uptake (within the treatment). This is achieved by:

```
x2 <- subsetBy(~ Treat, subset=uptake > quantile(uptake, prob=0.75), data=C02)
head(x2, 4)
```

```
#>       Plant Type conc uptake Treat
#> nchil.4  Qn1  Que  350   37.2 nchil
#> nchil.6  Qn1  Que  675   39.2 nchil
#> nchil.7  Qn1  Que 1000   39.7 nchil
#> chil.25  Qc1  Que  350   34.6  chil
```

The transformBy function

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

```
x3 <- transformBy(~ Treat, data=C02,
                  minU=min(uptake), maxU=max(uptake), range=diff(range(uptake)))
head(x3, 4)
```

```
#>   Plant Type conc uptake Treat minU maxU range
#> 1  Qn1  Que  350   37.2 nchil 10.6 39.7 29.1
#> 2  Qn1  Que  500   35.3 nchil 10.6 39.7 29.1
#> 3  Qn1  Que  675   39.2 nchil 10.6 39.7 29.1
#> 4  Qn1  Que 1000   39.7 nchil 10.6 39.7 29.1
```

The lmBy function

The lmBy function allows for fitting linear models to different strata of data (the vertical bar is used for defining groupings of data):

```
m <- lmBy(uptake ~ conc | Treat, data=C02)
coef(m)
```

```
#>      (Intercept)      conc
#> nchil      19.32 0.02221
#> chil       17.02 0.01602
```

The result is a list with a few additional attributes and the list can be processed further as e.g.

```
lapply(m, function(z) coef(summary(z)))

#> $nchil
#>      Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 19.31969   3.692936   5.232 0.0005408
#> conc         0.02221   0.006318   3.515 0.0065698
#>
#> $chil
#>      Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 17.01814   3.668315   4.639 0.0005709
#> conc         0.01602   0.006986   2.293 0.0407168
```

Functions related linear estimates and contrasts

A linear function of a p -dimensional parameter vector β has the form

$$C = L\beta$$

where L is a $q \times p$ matrix which we call the *Linear Estimate Matrix* or simply *LE-matrix*. The corresponding linear estimate is $\hat{C} = L\hat{\beta}$. A linear hypothesis has the form $H_0 : L\beta = m$ for some q dimensional vector m .

In the following we describe what is essentially simple ways of generating such L -matrices.

Computing linear estimates

First we focus on an additive model.

```
co2.add <- lm(uptake ~ Treat + Type, data=CO2)
```

Consider computing the estimated uptake for each treatment for plants originating from Mississippi: One option: Construct the LE-matrix L directly and then compute $L\hat{\beta}$:

```
L <- matrix(c(1, 0, 1,
              1, 1, 1), nrow=2, byrow=T); L

#>      [,1] [,2] [,3]
#> [1,]    1    0    1
#> [2,]    1    1    1

beta <- coef(co2.add); beta

#> (Intercept)  Treated  TypeMis
#>      38.04      -8.18     -11.75

L %% beta

#>      [,1]
#> [1,] 26.29
#> [2,] 18.11
```

However, this approach does not produce standard errors, confidence intervals etc. Once L has been constructed, such quantities can be constructed using `linest` (short for *linear estimate*) and the older but very similar `esticon` function (short for *estimate contrast*)

```
c1 <- linest(co2.add, L)
coef(c1)

#> estimate std.error statistic df p.value
#> 1 26.29 2.247 11.70 22 6.440e-11
#> 2 18.11 2.247 8.06 22 5.209e-08

confint(c1)

#> 0.025 0.975
#> 1 21.63 30.95
#> 2 13.45 22.77

c1 <- esticon(co2.add, L)
c1

#> estimate std.error statistic p.value beta0 df
#> [1,] 2.63e+01 2.25e+00 1.17e+01 6.44e-11 0.00e+00 22
#> [2,] 1.81e+01 2.25e+00 8.06e+00 5.21e-08 0.00e+00 22
```

Another option is to invoke `glht` (short for *general linear hypothesis*) from the `multcomp` package:

```
library(multcomp)
mc <- glht(co2.add, linfct=L)
summary(mc)
```

```
#>
#> Simultaneous Tests for General Linear Hypotheses
#>
#> Fit: lm(formula = uptake ~ Treat + Type, data = CO2)
#>
#> Linear Hypotheses:
#>      Estimate Std. Error t value Pr(>|t|)
#> 1 == 0    26.29      2.25  11.70 1.3e-10
#> 2 == 0    18.11      2.25   8.06 1.0e-07
#> (Adjusted p values reported -- single-step method)
```

In **doBy** there are facilities for computing L automatically and for supplying $L\hat{\beta}$ with standard errors etc.

```
L <- LE_matrix(co2.add, effect = "Treat", at=list(Type="Mis")); L

#>      (Intercept) Treated TypeMis
#> [1,]          1          0          1
#> [2,]          1          1          1
```

Least-squares means (LS-means)

A related question is: What is the estimated uptake for each treatment if we ignore the type (i.e. origin of the plants)? One option would be to fit a linear model without Type as explanatory variable:

```
co2.0 <- update(co2.add, . ~ . - Type)
L0 <- LE_matrix(co2.0, effect="Treat"); L0

#>      (Intercept) Treated
#> [1,]          1          0
#> [2,]          1          1

linest(co2.0, L=L0)

#> Coefficients:
#>      estimate std.error statistic    df p.value
#> [1,]    30.56     2.68    11.40 23.00      0
#> [2,]    23.99     2.38    10.09 23.00      0
```

An alternative would be to keep the focus on the original model but compute the estimated uptake for each treatment for an *average location*. That would correspond to giving weight 1/2 to each of the two locations. However, as one of the parameters is already set to zero to obtain identifiability, we obtain the LE-matrix L as

```
L1 <- matrix(c(1, 0, 0.5,
              1, 1, 0.5), nrow=2, byrow=T); L1

#>      [,1] [,2] [,3]
#> [1,]    1    0 0.5
#> [2,]    1    1 0.5

linest(co2.add, L=L1)

#> Coefficients:
#>      estimate std.error statistic    df p.value
#> [1,]    32.17     2.05    15.68 22.00      0
#> [2,]    23.99     1.79    13.41 22.00      0
```

Such a particular linear estimate is sometimes called a *least-squares mean*, an *LS-mean*, a *marginal mean* or a *population mean*. If data had been balanced, the LS-mean would be identical to the result obtained after fitting a model without Type.

The virtue of the **doBy** package is in this connection that L can be generated automatically with:

```
L1 <- LE_matrix(co2.add, effect="Treat"); L1

#>      (Intercept) Treated TypeMis
#> [1,]          1          0    0.5
#> [2,]          1          1    0.5
```

Notice: One may obtain the LS-mean directly as:

```
LSmeans(co2.add, effect="Treat")
## same as
linest(co2.add, L=LE_matrix(co2.add, effect="Treat"))
```

For a model with interactions, the LS-means are computed as above, but the L -matrix is different:

```
co2.int <- lm(uptake ~ Treat * Type, data=C02)
LE_matrix(co2.int, effect="Treat")

#>      (Intercept) Treated TypeMis Treated:TypeMis
#> [1,]           1           0           0.5           0.0
#> [2,]           1           1           0.5           0.5
```

Using (transformed) covariates

From the examples above it should follow that the key aspect of computing LS-means is the (automatic) generation of the L matrix. Therefore we shall in the following focus on form of the L matrix rather than on the computed LS-means. Covariates are fixed at their average value (unless the `at=...`-argument is used, see below). For example, `conc` is fixed at the average value:

```
co2.lm1 <- lm(uptake ~ conc + Type + Treat, data=C02)
lsm1 <- LSmeans(co2.lm1, effect="Treat")
lsm1$L

#>      (Intercept)  conc TypeMis Treated
#> [1,]           1 466.4     0.5      0
#> [2,]           1 466.4     0.5      1

lsm1a <- LSmeans(co2.lm1, effect="Treat", at=list(conc=700))
lsm1a$L

#>      (Intercept)  conc TypeMis Treated
#> [1,]           1  700     0.5      0
#> [2,]           1  700     0.5      1
```

A special issue arises in connection with transformed covariates. Consider:

```
co2.lm2 <- lm(uptake ~ conc + I(conc^2) + log(conc) + Type + Treat, data=C02)
lsm2 <- LSmeans(co2.lm2, effect="Treat")
lsm2$L

#>      (Intercept)  conc I(conc^2) log(conc) TypeMis Treated
#> [1,]           1 466.4  217529   6.145     0.5      0
#> [2,]           1 466.4  217529   6.145     0.5      1
```

Above `I(conc^2)` is the square of the average of `conc` (which is 2.1753×10^5) - not the average of the squared values of `conc` (which is 3.0476×10^5). Likewise `log(conc)` is the log of the average of `conc` (which is 6.145) - not the average of the log of `conc` (which is 5.908). To make computations based on the average value of the square of `conc` and the average of the log of `conc` do

```
co2.lm3 <- lm(uptake ~ conc + conc2 + log.conc + Type + Treat,
             data=transform(C02, conc2=conc^2, log.conc=log(conc)))
lsm3 <- LSmeans(co2.lm3, effect="Treat")
lsm3$L

#>      (Intercept)  conc  conc2 log.conc TypeMis Treated
#> [1,]           1 466.4 304758   5.908     0.5      0
#> [2,]           1 466.4 304758   5.908     0.5      1
```

Thus, if we want to evaluate the LS-means at `conc=700` then we can do:

```
lsm4 <- LSmeans(co2.lm3, effect="Treat", at=list(conc=700, conc2=700^2, log.conc=log(700)))
lsm4$L

#>      (Intercept)  conc  conc2 log.conc TypeMis Treated
#> [1,]           1  700 490000   6.551     0.5      0
#> [2,]           1  700 490000   6.551     0.5      1
```

Alternative models

The functions `esticon`, `linest`, `LSmeans` etc. are available for a range of model classes. We illustrate a few below: We may decide to treat `Type` as a random effect (here with only two levels). This leads to a *linear mixed effects model* as implemented in `lme4`, (Bates et al., 2015):

```
library(lme4)
co2.mix <- lmer(uptake ~ Treat + (1|Type), data=C02)
LSmeans(co2.mix, effect="Treat")

#> Coefficients:
#>      estimate std.error statistic    df p.value
#> [1,]    32.08     6.08     5.28  1.14  0.10
#> [2,]    23.99     5.99     4.00  1.08  0.14
```

Notice here that the parameter estimates themselves are similar to those of a linear model (had data been completely balanced, the estimates would have been identical). However, the standard errors of the the estimates are much larger under the mixed model. This is due to `Type` being treated as a random effect. Notice that the degrees of freedom by default are adjusted using a Kenward–Roger approximation (provided that `pbkrtest` package (Halekoh and Højsgaard, 2014) is installed). Adjustment of degrees of freedom is controlled with the `adjust.df` argument.

LS-means are also available in a *generalized linear model* setting as well as for *generalized estimating equations* as implemented in the `geepack` package, (Halekoh et al., 2006). In both cases the LS-means are on the scale of the linear predictor - not on the scale of the response.

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.

Bibliography

- S. M. Bache and H. Wickham. *magrittr: A Forward-Pipe Operator for R*, 2014. URL <https://CRAN.R-project.org/package=magrittr>. R package version 1.5. [p3]
- D. Bates, M. Mächler, B. Bolker, and S. Walker. Fitting linear mixed-effects models using `lme4`. *Journal of Statistical Software*, 67(1):1–48, 2015. doi: 10.18637/jss.v067.i01. [p8]
- M. Dowle and A. Srinivasan. *data.table: Extension of 'data.frame'*, 2019. URL <https://CRAN.R-project.org/package=data.table>. R package version 1.12.8. [p1]
- U. Halekoh and S. Højsgaard. A kenward-roger approximation and parametric bootstrap methods for tests in linear mixed models – the R package `pbkrtest`. *Journal of Statistical Software*, 59(9):1–30, 2014. URL <http://www.jstatsoft.org/v59/i09/>. [p8]
- U. Halekoh, S. Højsgaard, and J. Yan. The r package `geepack` for generalized estimating equations. *Journal of Statistical Software*, 15/2:1–11, 2006. [p8]
- T. Hothorn, F. Bretz, and P. Westfall. Simultaneous inference in general parametric models. *Biometrical Journal*, 50(3):346–363, 2008. [p1]
- S. Højsgaard and U. Halekoh. *doBy: Groupwise Statistics, LSmeans, Linear Contrasts, Utilities*, 2020. URL <http://people.math.aau.dk/~sorenh/software/doBy/>. R package version 4.6.6. [p1]
- R. V. Lenth. Least-squares means: The R package `lsmeans`. *Journal of Statistical Software*, 69(1):1–33, 2016. doi: 10.18637/jss.v069.i01. [p1]
- K. Müller and H. Wickham. *tibble: Simple Data Frames*, 2020. URL <https://CRAN.R-project.org/package=tibble>. R package version 3.0.1. [p1]
- SAS Institute Inc. *Base SAS 9.4 Procedures Guide, Seventh Edition*, April 2020. [p1]
- H. Wickham, M. Averick, J. Bryan, W. Chang, L. D. McGowan, R. François, G. Grolemund, A. Hayes, L. Henry, J. Hester, M. Kuhn, T. L. Pedersen, E. Miller, S. M. Bache, K. Müller, J. Ooms, D. Robinson, D. P. Seidel, V. Spinu, K. Takahashi, D. Vaughan, C. Wilke, K. Woo, and H. Yutani. Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686, 2019. doi: 10.21105/joss.01686. [p1]

Søren Højsgaard
Department of Mathematical Sciences, Aalborg University, Denmark
Skjernvej 4A
9220 Aalborg Ø, Denmark
sorenh@math.aau.dk