Basic data management in R

Antonio Gasparrini

01 November 2023

Contents

This session offers an overview of the basic methods and functions for data management in R. These basic tools are based on a small set of core functions, complemented with general methods such as labelling and indexing, relying on the programming flexibility of R. More sophisticated and efficient methods are provided in specialised packages and are not described here. In any case, the content is far from being comprehensive. The session starts with the description of common management tasks involving datasets, such as labelling, subsetting, and reordering, and then illustrates more complex operations such as appending, merging, and reshaping. The last two sections deal with managing variables and categorical data.

Labelling, subsetting, and reordering datasets

Quite often, the first step in data analysis is to prepare a final version of the dataset before running some statistical analysis. The first steps involve inspecting the data and checking for errors, among other issues, which we skip here. The next steps are usually performing basic data management tasks, which we describe in this and the following sections.

In the first examples illustrated in this section, we use the dataset ESOPH available in the DATASET package, storing the results of a case-control study of oesophageal cancer in France (see `help(esoph)`). Let's have a look:

```
head(esoph, 3)
```

```
##   agegp     alcgp    tobgp ncases ncontrols
## 1 25-34 0-39g/day 0-9g/day      0        40
## 2 25-34 0-39g/day    10-19      0        10
## 3 25-34 0-39g/day    20-29      0         6
```

A common data management task is to change the name of the variables of the dataset. This can be simply accomplished with basic *labelling* methods, for example using a replacement expression involving the function `names()`. Let's see:

```
names(esoph) <- c("age", "alcohol", "tobacco", "cases", "controls")
```

You can check that the names of the variables are now replaced. Similarly, the function `colnames()` can be used to obtain the same results, while `rownames()` can be applied to change the names of the rows.

Another basic operation is *subsetting*, applied to select specific records or variables in a dataset. This operation can be accomplished using the function `subset()`. Let's see an example where we select records in `esoph` with age corresponding to the group 55-64 and tobacco consumption (number of cigarettes a day) equal to 10-19 or 20-29. The code:

```
subset(esoph, age=="55-64" & tobacco %in% c("10-19","20-29"),
  select=c(alcohol, tobacco, cases, controls))
```

```
##      alcohol tobacco cases controls
## 48 0-39g/day   10-19     3       19
## 49 0-39g/day   20-29     3        9
## 52     40-79   10-19     6       15
## 53     40-79   20-29     4       13
## 56    80-119   10-19     8        7
## 57    80-119   20-29     3        3
## 60      120+   10-19     6        1
## 61      120+   20-29     2        1
```

The function includes a second argument `subset` (not named here) to define the selection of the rows, usually with logical statements, and a third argument `select` to keep only a reduced set of variables (in this case keeping all but `age`). Note that the latter includes expressions with unquoted variables, and can also use the operators ':' and '-' to select a range of variables or exclude some, respectively. See `help(subset)`. Interestingly, subsetting can be easily performed without specific functions, but simply relying on indexing by defining the selection of rows and columns in the operator '`[,]`'.

Another usual task is *reordering* a dataset, that is changing the order of the rows or (less often) the variables. The reordering of the rows usually involves sorting depending on the content of one or more variables. Again, this task can be performed simply using indexing and the function `order()`. As an example, we reorder esoph by tobacco and then alcohol consumption, and then age. In addition, we change the order of the variables accordingly. The code:

```
ord <- with(esoph, order(tobacco, alcohol, age))
esoph <- esoph[ord, c(3:1,4:5)]
```

You can check that the dataset has been reordered. The code above first creates an *indexing* vector using `order()`. This function accepts an arbitrary number of vectors as arguments, returning a vector with the ordering permutation of the elements of the first argument, and then applying the others to break optional ties.

Other more specific data management operations are not described here, for instance removing duplicated observations (see the functions `unique()` and `duplicated()`).

Appending, merging, and reshaping datasets

More complex data management operations are needed when the final dataset must be put together by combining multiple data sources, or when its format needs to be changed.

To illustrate some examples, we use a different dataset, specifically MTCARS available in the same package DATASET. The dataset includes technical aspects of 32 cars featured in the US magazine *Motor Trend*. Let's

have a look:

```
head(mtcars, 3)
```

```
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

The simplest case of combining data is when different sets of observations of the same variables are stored in multiple datasets. You can then combine the information by *appending* them, that is stacking them by columns. Curiously, in R there is no specific function for this task, probably because it can be easily performed using `rbind()` (note: there is a function `append()` but it only works for vectors).

A more complex computation is required when the datasets contain different sets of observations, an issue often occurring when they refer to different levels of the data, for example individual and group-level information. You can then apply a *merging* operation, that is combining the data in a way that the independent variables provided by each one are linked through the values of the common variables. For example, let's create a small hypothetical dataset with information on a certain score determined by the number of forward gears in a car:

```
gearscore <- data.frame(gear=3:6, score=c(2.1, 7.5, 8.1, 8.3))
```

We can merge this new data frame with `mtcars`, exploiting the common information on the forward gears. This can be done through the function `merge()`, using the expression:

```
merge(mtcars, gearscore, by="gear")
```

The resulting data frame (not shown here) now includes the variable `score`. The argument `by` defines the merging variable (or more than one), and can be omitted when this corresponds to common variable names (as in this case). If names are different, the arguments `by.x` and `by.y` can be used to refer to the first and second datasets, respectively. Also, you may note how by default only the common observations are kept in the final dataset (for instance, the observation with `gear` equal to 6 is not included). If you want to keep all the observations for both or one of the datasets, you can set the arguments `all`, `all.x`, or `all.y` to `TRUE`.

Another common task is to transform a dataset from *wide* to *long* formats, and vice versa. These formats are used when a dataset contains multiple observations on some variables related to groups defined by other variables. This is a common issue in longitudinal datasets, when the same individual contributes to multiple measurements sampled at different times. The process of transforming a dataset from wide to long, or in the opposite direction, is called *reshaping*.

To show an example, this time we use another dataset called SLEEP, also available in the DATASET package, which contains the results of a study on the effect of two soporific drugs in increasing the sleep time in ten patients (see `help(sleep)`). Let's have a look:

```
head(sleep, 3)
```

```
##   extra group ID
## 1   0.7     1  1
## 2  -1.6     1  2
## 3  -0.2     1  3
```

The data frame `sleep` has a long format, whereby each row refers to unique measurements and one or more variables define the groups. This dataset can be transformed in a wide format, where each row refers to a group or individual, and multiple observations are stored in different variables. This transformation can be performed using the function `reshape()`. The code:

```
sleepwide <- reshape(sleep, idvar="ID", v.names="extra", timevar="group",
  direction="wide")
head(sleepwide, 3)
```

```
##   ID extra.1 extra.2
## 1  1     0.7     1.9
## 2  2    -1.6     0.8
## 3  3    -0.2     1.1
```

The argument `idvar` defines the ID variable(s) identifying multiple records from the same group in the long format, `v.names` identifies the variables that correspond to multiple measurements, and `timevar` defines the name of the variable used to differentiate measurements within a group. The type of transformation is stated by `direction`. The function `reshape` can also transform in the opposite direction, in this case reversing the previous application using similar arguments and `direction="long"`.

Note that the usage of the `reshape()` function can be very difficult with more complex data structures, and you should consult the help page for more information (see `help(reshape)`).

Aggregating datasets

Sometimes the analysis needs to be performed at an aggregated level of information compared to the original data. In this case, a common data management procedure involves *aggregating* the dataset and creating a more compact version with information collapsed to a grouped level. In practical terms, the aggregation is done by recomputing the value of some variables with summary statistics repeated for groups defined by one or more factors.

In R, direct aggregation is rarely performed, as many functions can act and return results on different levels of aggregation, exploiting the flexibility of the language in handling and interrelating different data structures. However, it is sometimes useful to obtain an aggregated version of a dataset.

The basic function to perform an aggregation of a dataset is `aggregate()`, which can internally call any other existing R function to compute a wide range of aggregated statistics for one or more variables. The function `aggregate()` has different usages. The first syntax includes arguments `x` and `by` defining the columns of the data frame to be collapsed and a list of cross-classifying grouping factors (optionally tagged), respectively. The second syntax uses arguments `formula` and `data`, and it is used in the next examples. Check the help page using `help(aggregate)` for more examples and the alternative usage.

As an illustration, we go back to the dataset MTCARS introduced above, and we start with a simple aggregation where the data are collapsed by computing the average consumption (miles per gallon in the variable `mpg`) by engine and transmission type (variables `vs` and `am`, respectively). See `help(mtcars)` for details. The code is:

```
aggregate(mpg ~ vs + am, data=mtcars, FUN=mean)
```

```
##   vs am      mpg
## 1  0  0 15.05000
## 2  1  0 20.74286
## 3  0  1 19.75000
## 4  1  1 28.37143
```

The first argument of `aggregate()` in the expression above is a *formula*, namely an expression including the *tilde* operator '~' that defines a relationship between the variables in left and right-hand sides. Specifically, in this case, this means that an aggregated summary of `mpg` is computed by cross-classifying categories of `vs` and `am`. Formulae are common expressions in R and they are used in other contexts, for instance in regression models. The second argument data, often accompanying a formula, defines the object (commonly a data frame, or optionally a list) where the variables can be found (if not defined in the global environment). The last argument, FUN, identifies an object representing an existing R function to compute the summary statistic.

The usage of aggregate() includes an *ellipsis* argument, defined by the three-dot symbol '…', which allows including an indefinite number of other arguments to be passed internally to the aggregating function speci-

fied in FUN. This allows the definition of more complex aggregating statistics. For instance, we can compute the values of the inter-quartile range (IQR) of `mpg` by:

```
aggregate(mpg ~ vs + am, data=mtcars, FUN=quantile, probs=c(0.25, 0.75))
```

```
##   vs am mpg.25% mpg.75%
## 1  0  0  14.050  16.625
## 2  1  0  18.650  22.150
## 3  0  1  16.775  21.000
## 4  1  1  25.050  31.400
```

Here, the additional argument `probs` of the function `quantile()` is passed directly in the call to `aggregate()`, offering a compact and efficient syntax. You can notice that both statistics (25th and 75th percentiles) are added as separate variables to the aggregated dataset.

The function `aggregate()` allows aggregating multiple variables by adding them in a `cbind()` call on the left-hand of the formula (see the example on the help page). However, the aggregation can only be performed using the same function specified in `FUN`, and it is not possible to obtain different aggregating statistics for different variables. A solution is to perform multiple aggregations separately and then merge the datasets, although the process is somewhat unwieldy. Other functions in more specialised packages provide more elegant and efficient methods for data aggregation.

Creating and transforming variables

Other tasks in data management focus on the variables in the datasets. For example, it is customary to add, remove, or transform variables during an analysis. In R, such simple tasks do not require the use of dedicated functions, but can be performed using simple operators like '`$`' or '`[`' together with replacement expressions.

Let's see some examples using the dataset ESOPH introduced earlier. We first erase the saved object to get back the original variable names, which were changed in a previous section. As a first example, we create a new variable `tot` that reports the total number of subjects in each group (cases plus controls). The same result can be achieved with either the two expressions below:

```
rm(esoph)
data(esoph)
esoph$tot <- with(esoph, ncases+ncontrols)
esoph["tot"] <- with(esoph, ncases+ncontrols)
```

Note the use of the function `with()`, which allows calling directly the variables of `esoph` without referring to it with the dollar sign, providing a clearer syntax (see `?with`). You can now check that the data frame `esoph` has one additional column with the name `tot`, for example visualizing the dataset with the function `View()`. The same approach can be used to transform existing variables, simply using replacement expression to re-assign them.

The opposite task, which is removing a variable from a dataset, can be performed using the same replacement expressions above but assigning it to `NULL`. For instance, `tot` can now be removed by:

```
esoph$tot <- NULL
```

or using the indexing operator as above. You can check again and see how the variable `tot` is not included in `esoph` anymore.

If you plan to add, remove or transform several variables, you can use the function `transform()`. For example, let's include in `esoph` the total number of subjects seen above, the proportion of cases, and the odds, calculated as the ratio of cases and controls. The code is:

```
esoph <- transform(esoph,
  tot = ncases+ncontrols,
  prop = ncases/(ncases+ncontrols),
  odds = ncases/ncontrols
)
```

The function accepts as its first argument a data frame, and then an arbitrary number of tagged arguments representing new or transformed variables in the dataset (see `?transform`). In the example above, each argument has been written in a new line for clarity of code, but this is not a requirement.

Dealing with categorical data

Another task frequently required in data analysis is the categorization of continuous variables or the recoding of existing categorical variables. In R, categorical variables are stored in factors, and there are simple tools to apply transformations using such objects.

We illustrate some examples using the dataset BIRTHS from the package EPI. Differently from the DATASET package used in previous sections, EPI is neither a standard nor a recommended package, and therefore it must be installed (the first time) and then loaded. In addition, the data frame BIRTHS should be loaded as well into the session:

```
library(Epi)
data(births)
```

The first example features the creation of categorical variables from numeric ones, a common task in data management. Specifically, we want to transform the variable `hyp` in `births`, which represents an indicator of maternal hypertension (see `help(births)` for details). The variable is provided as a simple numeric code 0/1. Here we transform it into a proper categorical variable by calling `factor()` and assigning related labels for the categories:

```
head(births$hyp)
```

```
## [1] 0 0 0 0 1 0
```

```
births$hyp <- factor(births$hyp, labels=c("n", "y"))
head(births$hyp)
```

```
## [1] n n n n y n
## Levels: n y
```

In the call to `factor()` above, the levels are automatically assigned in alpha-numeric order (0 and then 1), with corresponding labels. The same process can be used to rename the categories of a variable, simply calling `factor()` with the existing variable but using different labels, or by re-assigning the levels using a replacement expression:

```
levels(births$hyp) <- c("no","yes")
head(births$hyp)
```

```
## [1] no  no  no  no  yes no
## Levels: no yes
```

Another common task is the re-ordering of the categories/levels of an existing variable. This can be done using the function `relevel()` (which however can only change the reference category, see `?relevel`), or more generally by another call to `factor()` stating directly the order of levels:

```
births$hyp <- factor(births$hyp, levels=c("yes","no"))
head(births$hyp)
```

```
## [1] no  no  no  no  yes no
## Levels: yes no
```

You can notice that the variable is unchanged, but the order of levels is swapped. This could have been accomplished using the original 0/1 numeric indicator by specifying both levels and labels in corresponding orders. let's erase the modified data frame, reload it, and create the factor again:

More often, categorical variables need to be created by truly continuous variables by defining intervals specified by selected cut-offs. In R, such a task is accomplished by the function `cut()`, which returns the corresponding factor. The main argument of `cut()` is `breaks`, which specifies the cut-off values that define the categories. Alternatively, `breaks` accepts a single value representing the number of categories, with cut-offs automatically selected to produce approximately balanced groups.

Let's see an application of `cut()`, and create a categorical variable `matagegr` representing groups of maternal age, using the information on the existing variable `matage` in `births` (see `help(births)` for details). The code:

```
births$matagegr <- cut(births$matage, breaks=c(0,25,30,35,40,100), right=F)
head(births$matage)
```

```
## [1] 34 30 35 31 33 33
```

```
head(births$matagegr)
```

```
## [1] [30,35) [30,35) [35,40) [30,35) [30,35) [30,35)
## Levels: [0,25) [25,30) [30,35) [35,40) [40,100)
```

The function produces three categories with internal cut-offs defined from 25 to 40 years of age in steps of 5 years. The argument `right` (with default to `TRUE`) states if intervals should be closed to the right (or left if `FALSE`), and therefore the third observation of `matage`, which is `35`, is assigned to the group `[35,40)`. Also, note that `breaks` must also include the external boundaries, and these should cover the entire range of the numeric variable. See `help(cut)` for further information.

By default, the levels of the factor are given labels identifying the boundaries of each interval. Alternatively, these can be directly specified with the argument `labels`. These lebels allows performing further transformation of categorical variables, for instance renaming and re-ordering categories, as shown above, but more importantly regrouping the levels. As an example, we can recode `matagegr` by re-assigning some of its levels in a replacement expression:

```
levels(births$matagegr)[2:3] <- "[25,35)"
head(births$matagegr)
```

```
## [1] [25,35) [25,35) [35,40) [25,35) [25,35) [25,35)
## Levels: [0,25) [25,35) [35,40) [40,100)
```

The expression above re-assigns the second and third levels (corresponding to `[25,30)` and `[30,35)`) to a single level `[25,35)`, basically merging the two categories together.