An introduction to R

Jorge Cimentada and Basilio Moreno

6th of July 2019

Learning more about R functions

So far we have only had a glimpse to the linear model formula in a previous example. Here we are going to go a bit deeper on the logic behind the **formula interface** used by some R functions.

- Run one example with the `lm` (Fitting linear models) function and the `mtcars` dataset.

Remember to ask for help if needed `?function`

```
lm(mpg ~ vs + cyl, data = mtcars)
by(mtcars, mtcars$cyl, summary)
mtcars$mpg_mean <- ifelse(mtcars$mpg >= mean(mtcars$mpg), 1, 0)
```

The formula interface

Let's take a look at the documentation for `t.test`:

```
?t.test
```

We see that there are two separate *methods* (more about this in a second) for interacting with `t.test`: the one we just used, passing arguments `x` and maybe `y`, and another one that uses a `formula`. Formulas play a huge role in R.

Here we check the difference in mpg recorded by type of transmission (**a**utomatic or **m**anual)

```
#am variable refers to Transmission (0 = automatic, 1 = manual)
my_test <- t.test(mpg ~ am, data=mtcars)
my_test
```

```
##
##  Welch Two Sample t-test
##
## data:  mpg by am
## t = -3.7671, df = 18.332, p-value = 0.001374
```

```
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -11.280194  -3.209684
## sample estimates:
## mean in group 0 mean in group 1
##       17.14737        24.39231
```

Note that we have not just printed the output of running the t-test. Instead, we have assigned a name to that output, because it is an object that contains a lot more information than what is printed in the screen. This is the most distinctive feature of R with respect to other statistical languages.

We can inspect the contents of the `my_test` object using the function `str`:

```
str(my_test)
```

```
## List of 9
## $ statistic  : Named num -3.77
# ..- attr(*, "names")= chr "t"
# $ parameter  : Named num 18.3
# ..- attr(*, "names")= chr "df"
# $ p.value    : num 0.00137
# $ conf.int   : num [1:2] -11.28 -3.21
# ..- attr(*, "conf.level")= num 0.95
# $ estimate   : Named num [1:2] 17.1 24.4
# ..- attr(*, "names")= chr [1:2] "mean in group 0" "mean in group 1"
# $ null.value : Named num 0
# ..- attr(*, "names")= chr "difference in means"
# $ alternative: chr "two.sided"
```

```
# $ method     : chr "Welch Two Sample t-test"
# $ data.name  : chr "mpg by am"
# - attr(*, "class")= chr "htest"
```

The formula interface

Note that `my_test` is a list contaning all the information related to this specific t-test. Here we keep all statistic parameters, which we can access and use independently. For instance:

```
my_test$statistic
my_test$conf.int
my_test$estimate
```

It is a good moment to go back to the documentation and compare the output of the test against the “Value” section of the help file.

Let's take a deeper look into the formula interface and the structure of objects using a linear model.

The formula interface: linear models

Consider the case in which we can to now run a regression on the number of affairs using information about. Do not much attention to the theortical soundness of the analysis:

```
sample_model <- lm(mpg ~ I(cyl - 3) * hp + factor(am), data=mtcars)
```

We can see here the elegance of the formula interface. The model is doing several things. First, we are recentering `cyl` (number of cylinders) so that 3 is the new 0 value. It is important that the expression is wrapped in the `I()` function to ensure that the `-` inside is taken as an arithmetical operator and not as a formula operator.

Then, multiply that new variable by the variable hp (horsepower) which is a factor, which uses yes as the

reference level in the dummy expansion. Not only that, the * operator creates the full interaction including the main effects. Finally, although `am` is an numerical variable, we pass it through `factor` to cast it into a categorical with *n* ??? 1 dummies. As we can see, the formula takes care of a lot of the transformations and lets us express the structure of the model very succintly. We could have passed the transformed data directly (look at the y and x arguments in the `lm` documentation), but this approach is considerably easier.

The formula interface: linear models

Lets take a look at the object to see the estimated coefficients:

```
sample_model
```

```
#     Call:
# lm(formula = mpg ~ I(cyl - 2) * hp + factor(am), data = mtcars)
#
# Coefficients:
#   (Intercept)     I(cyl - 2)             hp    factor(am)1  I(cyl - 2):hp
#      38.29101       -2.91100       -0.14151        3.76084        0.01854
```

The formula interface: linear models

Sometimes that is the only information that we need, but most of the time we want to make inference with those coefficients. We can see this information by getting a `summary` of the object:

```
summary_model <- summary(sample_model)
summary_model
```

```
#     Call:
# lm(formula = mpg ~ I(cyl - 2) * hp + factor(am), data = mtcars)
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -4.2202 -1.5052 -0.2882  1.0368  5.9707
#
# Coefficients:
#               Estimate Std. Error t value Pr(>|t|)
# (Intercept)   38.291011   4.335476   8.832  1.9e-09 ***
# I(cyl - 2)    -2.911001   0.943814  -3.084  0.00467 **
# hp            -0.141511   0.045432  -3.115  0.00433 **
# factor(am)1    3.760837   1.199249   3.136  0.00411 **
# I(cyl - 2):hp  0.018541   0.007691   2.411  0.02301 *
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 2.593 on 27 degrees of freedom
# Multiple R-squared:  0.8388,  Adjusted R-squared:  0.8149
# F-statistic: 35.13 on 4 and 27 DF,  p-value: 2.451e-10
```

The formula interface: linear models

Let's see how the two objects (`sample_model` and `summary_model`) differ by taking a look at what they contain:

```
names(sample_model)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "contrasts"     "xlevels"       "call"          "terms"
## [13] "model"
```

```
names(summary_model)
```

```
##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

The formula interface: linear models

The shortest introduction to objects and methods

This is one of the beauties of R as an statistical language. The object `summary_model` now holds all the information about the model. We could for instance retrieve the coefficients and the covariace matrix to get the normal-based confidence intervals (0.975 limit example):

```
coefficients(sample_model) + qt(0.975, df=sample_model$df.residual) *
sqrt(diag(vcov(sample_model)))
```

```
#   (Intercept)    I(cyl - 2)            hp   factor(am)1 I(cyl - 2):hp
# 47.18667281   -0.97445500   -0.04829208    6.22149132    0.03432208
```

and check that the result matches the outcome of the built-in function:

```
confint(sample_model)
```

```
#                          2.5 %      97.5 %
# (Intercept)   29.395349391 47.18667281
# I(cyl - 2)    -4.847546663 -0.97445500
# hp            -0.234729499 -0.04829208
# factor(am)1    1.300181717  6.22149132
```

```
# I(cyl - 2):hp  0.002759139  0.03432208
```

The two lines previous illustrate the way R works. `sample_model` is an object that contains a number of *attributes* like the coefficients or the residual degrees-of-freedom that were obtained when we fit the model. We access these attributes either through functions like `coefficients` or through the `$` operator, because `sample_model` is still a list.

```
names(sample_model)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "contrasts"     "xlevels"       "call"          "terms"
## [13] "model"
```

On the other hand, we can make operations over the elements in `sample_model`. Moreover, these function will know that they are being applied to the outcome of a l inear model, because that information is given by the class to which `sample_model` belongs.

```
class(sample_model)
```

```
## [1] "lm"
```

Finally, it is sensible to asume we will not be working exclusively with R. Most of the time (and because of multiple reasons) we need to work in a multi-platform environment.

How do we import data -like a survey result- into R? It depends on the format of the data we are interested in. However, most of the functions focused on that goal follow a similar working structure.

The most simple case is `read.table`, a built-in function in R. Take a moment to familiase yourself with its arguments with the following example.

```
read.table(file = "local/path/my_csv_file.csv", sep = ";", dec = ",",
           row.names = c("row1", "row2"), col.names = c("col1", "col2"))
```

This function will try to identify data structured in a

tabular way. The most likely format for this data to be written in is `.csv`, which stands for **c**omma **s**eparated **v**alues. `.csv` files are the best for compatibility since they can be read straight away in most software solutions. However, that comes at the cost of virtual size and the lack of extra features.

Data import and export: the basics

Additionaly, there is another function for writing data out of R. The most basic approach is `write.table` which uses the `.csv` format by default:

```
write.table(x = matrix(data = 0.61:20, nrow = 2, ncol = 2), file = "my_csv_file.csv",
            row.names = c("obs_1", "obs_2"), quote = FALSE, sep = ";", dec = ",", na = "")

?write.table
```

Here we are modifying some parameters for convenience. Instead of using a simple comma to separate values, the *sep*aration argument is set to ";" because this symbol makes it harder to find false separators in the values, while it makes it a bit easier to read the data when we open the raw file. Similarly: `.tsv` (tab **separated **v*alues).

Data import and export: reference guide

Despite the vast possibilities of data formats you may find in your work, we have built a starting guide to let you know what are some of the most common solutions in a handful of cases. From a realistic point of view, we encourage you to check with a web search on the specific format you are trying to work with. The reason is that there are countless packages available, and some of them are not finished software and hence their reliability and/or features can change from version to version.

Type of data	**Package**	**Function for import**	**Function for export**
Basic (unknown)	Built-in function {utils}	`read.table()`	`write.table()`
Basic (`.csv`)	Built-in function {utils}	`read.csv()`	`write.csv()`
Basic (`.tsv`)	Built-in function {utils}	`read.csv( ... , sep = "\t")`	`write.csv(... , sep = "\t")`
STATA (`.dta`)	Foreign	`read.dta()`	`write.dta()`
	Haven	`read_dta()`	`write_dta()`
SPSS (`.sav`)	Foreign	`read.spss()`	`write.foreign(... , package="SPSS")`
	Haven	`read_sav()`	`write_sav()`
SAS (`.sas`)	Foreign	`read.csv()`	`write.foreign(... , package="SAS")`
	Haven	`read_sas()`	`write_sas()`
MS Excel (`.xlsx` /	Readxl	`read_excel()`	*Not available*

.xls)			
	Writexl	*Not available*	`write_xlsx()`