Descriptives and simple statistics

October 30, 2017

Moving from the prompt to the script

So far, we have been doing everything through on the interpreter directly. We will use the interpreter a lot during data analysis to test things, but it is probably a good idea to keep our code somewhere. Here is when using a tool like RStudio starst to make sense: we want something that makes it easy to edit text files (like navigation tools or syntax highlighting) and also that connects to the R interpreter.

You will probably run the rest of the sessions by typing in the "Code" window and sending things to the console from there.

Basic data analysis

Let's start by reading in some data from the Internet.

```
affairs <- read.csv("http://koaning.io/theme/data/affairs.csv")
```

The dataset contains data about the number of affairs of 601 politicians and some sociodemographic information. A detailed description of the variables and some interesting results can be found in Fair, R. (1977) "A note on the computation of the tobit estimator", Econometrica, 45, 1723-1727.

Let's start by taking a look at the dataset. For instance, we can print the first 6 rows using the function `head`:

```
head(affairs)
```

```
##      sex age    ym child religious education occupation rate nbaffairs
## 1   male  37 10.00    no         3        18          7    4         0
## 2 female  27  4.00    no         4        14          6    4         0
## 3 female  32 15.00   yes         1        12          1    4         0
## 4   male  57 15.00   yes         5        18          6    5         0
## 5   male  22  0.75    no         2        17          6    3         0
## 6 female  32  1.50    no         2        17          5    5         0
```

We can also take a look at some descriptives with the function `summary` applied to individuals variables:[1]

```
summary(affairs$nbaffairs)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    0.00    0.00    1.46    0.00   12.00
```

and

```
summary(affairs$child)
```

```
##  no yes
## 171 430
```

Notice that `summary` does different things depending on the class of the input it receives. In the first case, `summary` sees a numeric variable and produces the mean and some cutpoints. In the second case, `summary` sees a factor variable (a categorical variable) and produces a frequency table. This is a pattern that we will encounter very frequently in R.

We can be more specific about getting a frequency table by using:

[1] The function could be applied to a dataset but I find that amount of information overwhelming.

```r
table(affairs$child)
```

```
##
##  no yes
## 171 430
```

To transform the previous table into a frequency we can take several routes, both illustrative of the way R works relative to software like SAS or Stata. The first one is to do it manually, by just dividing the frecuencies by the total size, calculated by summing over the column `children`. It is convenient here to remember that `child` is a factor that indicates whether the politician has children or not. Therefore, the number of observations is just the length of the vector.

```r
table(affairs$child)/length(affairs$child) ## We could also have used nrow(affairs)
```

```
##
##    no   yes
## 0.285 0.715
```

Note that we don't create new variables in between but instead we perform the operation on-the-fly with the output of the two functions. The second option is to compose two functions together:

```r
prop.table(table(affairs$child))
```

```
##
##    no   yes
## 0.285 0.715
```

The output of `table` is passed to `prop.table` which transforms a table into proportions.

Hypothesis testing

We can now start analyzing the data. For instance, we would like to check the difference in the mean of the number of affairs by whether the politician has children or not. The sample mean for each group can be calculated as:

```r
mean(affairs$nbaffairs[affairs$child == "no"])
mean(affairs$nbaffairs[affairs$child == "yes"])
```

A t-test can be performed in several ways. The most natural one for new people to R is passing variables. For instance, if we wanted to test one variable against the standard null:

```r
t.test(affairs$nbaffairs)
```

```
##
##  One Sample t-test
##
## data:  affairs$nbaffairs
## t = 10, df = 600, p-value <2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  1.19 1.72
## sample estimates:
## mean of x
##      1.46
```

We can also test equality of two means by passing *two* vectors to the function:

```
t.test(affairs$nbaffairs[affairs$child == "no"], affairs$nbaffairs[affairs$child == "yes"])
```

```
##
##  Welch Two Sample t-test
##
## data:  affairs$nbaffairs[affairs$child == "no"] and affairs$nbaffairs[affairs$child == "yes"]
## t = -3, df = 400, p-value = 0.005
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.286 -0.234
## sample estimates:
## mean of x mean of y
##     0.912     1.672
```

The thing to notice here is that the second vector is a second *optional argument* to the function and, by passing it, the function performs a different routine. 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 usesa `formula`. Formulas play a huge role in R:

```
my_test <- t.test(nbaffairs ~ child, data=affairs)
my_test
```

```
##
##  Welch Two Sample t-test
##
## data:  nbaffairs by child
## t = -3, df = 400, p-value = 0.005
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.286 -0.234
## sample estimates:
##  mean in group no mean in group yes
##             0.912             1.672
```

The LHS is the variable we want to test but split by the groups indicated in the RHS. The argument `data` indicates where those two variables live: they are columns of the dataset `affairs`. The formula interface probably makes a lot more sense if we consider how we would run the same test using a linear model, which we will see in a moment. The outcome variable is the LHS of the equation in which we separate the equal sign with a `~`. The RHS is a dummy variable (a factor) that splits the sample in two groups. There are other ways to pass data to the t test. Take a look at the documentation for more information.

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 -2.84
##   ..- attr(*, "names")= chr "t"
##  $ parameter  : Named num 397
##   ..- attr(*, "names")= chr "df"
```

```
##  $ p.value    : num 0.00473
##  $ conf.int   : atomic [1:2] -1.286 -0.234
##   ..- attr(*, "conf.level")= num 0.95
##  $ estimate   : Named num [1:2] 0.912 1.672
##   ..- attr(*, "names")= chr [1:2] "mean in group no" "mean in group yes"
##  $ 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 "nbaffairs by child"
##  - attr(*, "class")= chr "htest"
```

Note that `my_test` is a list that contains all the information pertaining to the t-test we ran. It contains the statistic, the degrees of freedom, the confidence interval, ... and more importantly, we can access all of those elements and use them elsewhere. For instance, we can get the test statistic from the element `statistic`, or the confidence interval or the estimate by accessing the elements in the list:

```
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

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(nbaffairs ~ I(age - 18)*child + factor(religious), data=affairs)
```

We can see here the elegance of the formula interface. The model is doing several things. First, we are recentering age so that 18 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 `child` 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 `religious` 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.

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

```
sample_model
```

```
##
## Call:
## lm(formula = nbaffairs ~ I(age - 18) * child + factor(religious),
##     data = affairs)
##
## Coefficients:
##          (Intercept)           I(age - 18)              childyes
##                1.053                 0.116                 1.597
##   factor(religious)2    factor(religious)3    factor(religious)4
##               -0.880                -0.653                -1.923
```

```
##   factor(religious)5  I(age - 18):childyes
##               -1.899                -0.100
```

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 = nbaffairs ~ I(age - 18) * child + factor(religious),
##     data = affairs)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -3.669 -1.915 -1.033 -0.115 11.134
##
## Coefficients:
##                       Estimate Std. Error t value Pr(>|t|)
## (Intercept)             1.0525     0.5899    1.78  0.07489
## I(age - 18)             0.1164     0.0370    3.14  0.00175
## childyes                1.5966     0.5169    3.09  0.00210
## factor(religious)2     -0.8798     0.5284   -1.67  0.09642
## factor(religious)3     -0.6526     0.5442   -1.20  0.23097
## factor(religious)4     -1.9228     0.5218   -3.69  0.00025
## factor(religious)5     -1.8992     0.6110   -3.11  0.00197
## I(age - 18):childyes   -0.1002     0.0409   -2.45  0.01457
##
## Residual standard error: 3.22 on 593 degrees of freedom
## Multiple R-squared:  0.0612, Adjusted R-squared:  0.0501
## F-statistic: 5.52 on 7 and 593 DF,  p-value: 3.58e-06
```

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 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:

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

```
##          (Intercept)          I(age - 18)             childyes
##               2.2110               0.1891               2.6117
##   factor(religious)2   factor(religious)3   factor(religious)4
##               0.1579               0.4163              -0.8980
##   factor(religious)5 I(age - 18):childyes
##              -0.6993              -0.0199
```

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

```
confint(sample_model)
```

```
##                         2.5 %  97.5 %
## (Intercept)          -0.1060  2.2110
## I(age - 18)           0.0437  0.1891
## childyes              0.5815  2.6117
## factor(religious)2   -1.9174  0.1579
## factor(religious)3   -1.7214  0.4163
## factor(religious)4   -2.9476 -0.8980
## factor(religious)5   -3.0992 -0.6993
## I(age - 18):childyes -0.1806 -0.0199
```

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 linear model, because that information is given by the class to which `sample_model` belongs.

```
class(sample_model)
```

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

In this case, `sample_model` does not contain the confidence interval (why should it?), but `confint` knows where to look for the information it needs in the object. `confint` is therefore a *method*.