Statistics

January 15, 2019

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.
##   0.000   0.000   0.000   1.456   0.000  12.000
```

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
## 0.2845258 0.7154742
```

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
## 0.2845258 0.7154742
```

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 = 10.82, df = 600, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  1.191643 1.720171
## sample estimates:
## mean of x
##  1.455907
```

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 = -2.8412, df = 396.56, p-value = 0.004726
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.2855560 -0.2340687
## sample estimates:
## mean of x mean of y
## 0.9122807 1.6720930
```

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 = -2.8412, df = 396.56, p-value = 0.004726
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.2855560 -0.2340687
## sample estimates:
##  mean in group no mean in group yes
##         0.9122807         1.6720930
```

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   : num [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 pay much attention to the theoretical 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 (use `:` instead of `*` to include interactions but not 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
##               1.0525                0.1164                1.5966
##   factor(religious)2    factor(religious)3    factor(religious)4
```

```
##              -0.8798               -0.6526               -1.9228
##   factor(religious)5  I(age - 18):childyes
##              -1.8992               -0.1002
```

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
## -3.6692 -1.9148 -1.0335 -0.1145 11.1345
##
## Coefficients:
##                       Estimate Std. Error t value Pr(>|t|)
## (Intercept)            1.05252    0.58988   1.784  0.07489 .
## I(age - 18)            0.11640    0.03701   3.145  0.00175 **
## childyes               1.59658    0.51687   3.089  0.00210 **
## factor(religious)2    -0.87977    0.52836  -1.665  0.09642 .
## factor(religious)3    -0.65256    0.54422  -1.199  0.23097
## factor(religious)4    -1.92280    0.52178  -3.685  0.00025 ***
## factor(religious)5    -1.89925    0.61097  -3.109  0.00197 **
## I(age - 18):childyes -0.10023    0.04091  -2.450  0.01457 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.215 on 593 degrees of freedom
## Multiple R-squared:  0.06121,    Adjusted R-squared:  0.05013
## F-statistic: 5.524 on 7 and 593 DF,  p-value: 3.585e-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"
```

Prediction

Let's take a more careful look at the model we fit before:

```
affairs <- read.csv("http://koaning.io/theme/data/affairs.csv")
sample_model <-  lm(nbaffairs ~ I(age - 18)*child + factor(religious), data=affairs)
```

We took a look at some values of interest, like the estimated coefficients or the confidence intervals around them. It may also be interesting to take a look at predictions on the original dataset that we used (remember that `sample_model` carries the data used to fit the model).

```
yhat <- predict(sample_model)
head(yhat)
```

```
##         1         2         3         4         5         6
## 2.6115147 0.1772928 2.8754507 1.3803998 0.6383385 1.8023176
```

The `predict` method takes a number of useful arguments, like `newdata`, which applies the estimated coefficients to a new dataset.

```
my_predictions <- predict(sample_model,
                          newdata=data.frame("age"=54, "child"="yes", religious=1))
my_predictions
```

```
##        1
## 3.231144
```

Usually, we want to see predictions with their uncertainty. Let's take a look at the documentation to see how to get confidence intervals:

```
?predict
```

Not very useful, right? The reason is that `predict` is a *generic function* that operates on different kinds of objects/models. Think about predictions for a linear model or for a logistic regression. They are still predictions but they are calculated differently and they should be offering different options. But they user should not need to remember the class of the model that was fit: and the end of the day, we have been insisting on the fact that objects in `R` carry a lot of information around. If we look at the bottom of the help file, we will see the method for `lm` models, which is what we want:

```
?predict.lm
```

After this small detour, we finally see how to get the confidence intervals:

```
my_predictions <- predict(sample_model,
                          newdata=data.frame("age"=54, "child"="yes", religious=1),
                          interval="confidence")
my_predictions
```

```
##        fit      lwr      upr
## 1 3.231144 2.057784 4.404504
```

A bit more on modeling

We can think about running some other kinds of models on our dataset. For instance, we could think about running a logistic regression.

```
logit_model <-  glm(I(nbaffairs > 0) ~ I(age - 18)*child + factor(religious),
                    data=affairs,
                    family=binomial(link="logit")) # link="logit" is the default
summary(logit_model)
```

```
##
## Call:
```

```
## glm(formula = I(nbaffairs > 0) ~ I(age - 18) * child + factor(religious),
##     family = binomial(link = "logit"), data = affairs)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.1598  -0.8241  -0.6534  -0.3417   2.3606
##
## Coefficients:
##                       Estimate Std. Error z value Pr(>|z|)
## (Intercept)          -1.39595    0.43642  -3.199 0.001381 **
## I(age - 18)           0.05451    0.02816   1.936 0.052876 .
## childyes              1.33186    0.41607   3.201 0.001369 **
## factor(religious)2   -0.75853    0.35243  -2.152 0.031375 *
## factor(religious)3   -0.42668    0.35605  -1.198 0.230773
## factor(religious)4   -1.38809    0.36064  -3.849 0.000119 ***
## factor(religious)5   -1.29928    0.43945  -2.957 0.003110 **
## I(age - 18):childyes -0.05333    0.03065  -1.740 0.081869 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 675.38  on 600  degrees of freedom
## Residual deviance: 638.87  on 593  degrees of freedom
## AIC: 654.87
##
## Number of Fisher Scoring iterations: 4
```

Nothing in the previous call should be odd, we just applied the same logic as before but to a new particular type of model.

One of the things that we could do now is check to what extent the model is performing well. We could take a significance testing approach, but we could also evaluate performance in terms of prediction. We are dealing with a categorical output, so we could for instance check the confusion matrix that is implicit from predicting probabilities:

```
phat <- predict(logit_model, newdata=affairs, type="response")
table(affairs$nbaffairs > 0, phat > 0.5, dnn=list("Observed", "Predicted"))
```

```
##         Predicted
## Observed FALSE TRUE
##    FALSE   451    0
##    TRUE    149    1
```

The model performs poorly, but that's probably because the model predicts low probabilities to a positive event (an affair). We could then play with the probability threshold to have a more realistic confusion matrix:

```
table(affairs$nbaffairs > 0, phat > quantile(phat, .5), dnn=list("Observed", "Predicted"))
```

```
##         Predicted
## Observed FALSE TRUE
##    FALSE   256  195
##    TRUE     49  101
```

Still not a good performance, but still much better than the original matrix we got.

We could also explore the predictors and see their marginal effects. For instance, by checking how the probability of a positive even changes as we move some of the variables on the RHS. One way of accomplishing

this is by, for instance, applying our model to a grid of variables:

```
fake_data <- expand.grid(age = c(18, 36, 54, 72),
                         child = c("no", "yes"),
                         religious = 1)
fake_data$prediction <- predict(logit_model, newdata=fake_data, type="response")
fake_data
```

```
##   age child religious prediction
## 1  18    no         1  0.1984596
## 2  36    no         1  0.3977662
## 3  54    no         1  0.6379292
## 4  72    no         1  0.8245605
## 5  18   yes         1  0.4839842
## 6  36   yes         1  0.4892990
## 7  54   yes         1  0.4946163
## 8  72   yes         1  0.4999348
```

We did two things here. First, we created a fake dataset by expanding on all the combinations of the values that were passed to `expand.grid`. Then, we applied our predicted model to this new dataset and got the predicted probabilities for each case. Notice I put those predictions back on the fake dataset to be able to see to what combination each prediction corresponds.

We can now see how the change in the probability for different combinations of the age and child variable. But inspecting the model this way may be hard. It is probably better to accomplish this with plots.