MODULE 4: MODELING DATA

7316 - INTRODUCTION TO DATA ANALYSIS WITH R

Mickaël Buffart (mickael.buffart@hhs.se)

TABLE OF CONTENTS

In Module 3, we have seen how to describe data with static visuals and descriptive statistics, including correlation tables. In this module, we will cover how to model data in R with common models and display the results.

1. R Formulæ

In R, to estimate a model, you need formulæ. A formula is an R object that states the equation representing the relationship you are trying to model. As a simple example, let us assume you would like to estimate an OLS model described by the following equation:

$$Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + \epsilon$$

In R, you would write **the formul**a as: `Y ~ X1 + X2`. You can write formulæ for all kinds of mathematical models. Because the formula is an R object, just as any object, you can save it in your environment, change its type on the fly, manipulate it, etc. For example, assign the formula with `myf <- Y ~ X1 + X2`.

1.1 Formula syntax:

- The left-hand side and the right-hand side are separated with `~`. Example: `Y ~ X`
- In a standard linear model, separate the independent variables with `+`. Example: `Y ~ X1 + X2` .
- You can summarize all the variables in a dataframe with a `.`. For example, `Y ~ .` is the formula to regress `Y` on any other variables in your dataframe of interest.
- For the rest, I provide below some examples of formulæ and the equivalent equation:

Example of formulæ

Formula	Equation	Note
`Y ~ X1 + X2`	$Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + \epsilon$	Linear model
`Y ~ .`	$Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + \epsilon$	If our dataframe contains only `Y`, `X1`, and `X2`, `Y ~ .` is the same as `Y ~ X1 + X2`.
`Y ~ X1 + X2 - 1`	$Y = \beta_1 X_1 + \beta_2 X_2 + \epsilon$	`-` is used to remove something from the equation. `1` stands for the intercept. It is also possible to remove a variable, as below. You could also force including the intercept in the equation by adding `+ 0`.
`Y ~ . - X2`	$Y = \alpha + \beta_1 X_1 + \epsilon$	The formula means: to include all the variables in the dataframe, and remove `X2`.
`Y ~ X + I(X^2)`	$Y = \alpha + \beta_1 X + \beta_2 X^2 + \epsilon$	Quadratic effect In the formula, the function I( ) stands for as-is. If you forget it, R will believe X is included twice in the model and remove the second occurrence.
`Y ~ X1*X2`	$Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2 + \epsilon$	Interacting effect and main effects
`Y ~ X1:X2`	$Y = \alpha + \beta_1 X_1 X_2 + \epsilon$	Interacting effect without the main effects

Because you can manipulate formulæ as a normal R object, you can also update them. Let's take an example. Imagine that you are trying to run 4 OLS regression and produce the following table:

	Model 1	Model 2	Model 3	Model 4
C1	2.026***	1.956***	2.089***	2.028***
	(0.233)	(0.103)	(0.235)	(0.099)
C2	2.990***	3.190***	2.988***	3.188***
	(0.193)	(0.086)	(0.192)	(0.081)
C3	-0.212	-0.255	-0.141	-0.173
	(0.460)	(0.203)	(0.460)	(0.194)
X1		6.775***		6.797***
		(0.249)		(0.236)
X2			7.254*	8.328***
			(4.353)	(1.836)
Constant	246.202***	120.677***	132.633*	-10.105
	(19.348)	(9.714)	(70.819)	(30.276)
Observations	184	184	184	184
R^2	0.662	0.934	0.667	0.941
Adjusted R^2	0.657	0.933	0.660	0.940

Example of OLS regression

You could either write each model and equation one by one, with the risk, if you have many models or variables, or making typos or forgetting one term, or you can update your formulæ based on the previous ones, as follow:

```
# the four formulae of the models in the table above
model_1 <- Y ~ C1 + C2 + C3
model_2 <- update(model_1, . ~ . + X1)
model_3 <- update(model_1, . ~ . + X2)
model_4 <- update(model_2, . ~ . + X2)
```

2. OLS regression in R

Being able to reproduce equations is not enough. To obtain the output above, you also need to estimate models. For example, the primary method of performing an OLS regression in R is to use the `lm()` function. To fit an OLS model, you can just call the `lm()` function. Once you have fitted the model, you can visualize it with the `summary()` function, store it in an object, or use it for prediction, extracting residuals, etc.

To fit the model 1 above, using OLS regressions, write:

```
# Loading data.You can find ols_dataset on Canvas
df <- rio::import("./data/ols_dataset.xlsx")

# Fitting
fit_1 <- lm(Y ~ C1 + C2 + C3, data = df)

# You could as well use the formula that we assigned to model_1:
fit_1 <- lm(model_1, data = df)
```

- Note that we assigned the fitted model (generated with `lm()`) into `fit_1`. You can then save it in an `.Rds` file, display the results, extract residuals, etc.

- To display the fitted model, use:

```
summary(fit_1)
```

- The ouptut will be:

```

Call:
lm(formula = model_1, data = df)

Residuals:
    Min      1Q  Median      3Q     Max 
-27.737  -6.042   0.258   5.493  27.450 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 246.2023    19.3482  12.725  < 2e-16 ***
C1            2.0265     0.2330   8.696 2.11e-15 ***
C2            2.9903     0.1928  15.512  < 2e-16 ***
C3           -0.2121     0.4598  -0.461    0.645    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9.274 on 180 degrees of freedom
Multiple R-squared:  0.6623,	Adjusted R-squared:  0.6567 
F-statistic: 117.7 on 3 and 180 DF,  p-value: < 2.2e-16
```

- Note that some measures of diagnostics, such as the R2, the F-statistic, or the distribution of residuals, are also available in the output of `summary()`.
- You could also extract residuals, fitted values, etc.. Bellow, I assign them to new variables into the dataframe.

```
fit_1$coefficients[1]
```

```
(Intercept) 
   246.2023 
```

```
# Extract the fitted values
df$fitted <- fit_1$fitted.values

# Extract the residuals of the models
df$resid <- fit_1$residuals
```

2.1 Diagnostic checking for OLS regression

Saving the residuals, the fitted values, or other parameters, may be useful for diagnostic checking of your models.

2.1.1 Plotting diagnostics

- Using what we have see last time, we can plot the actual and the fitted values:

```
library(ggplot2)
```

```
Warning: package 'ggplot2' was built under R version 4.3.1
```

```
ggplot(df) + aes(y = Y, x = C1) +
  geom_point() +
  geom_segment(aes(xend = C1, yend = fitted), size = 0.2) +
  ggthemes::theme_clean()
```

```
Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
ℹ Please use `linewidth` instead.
```

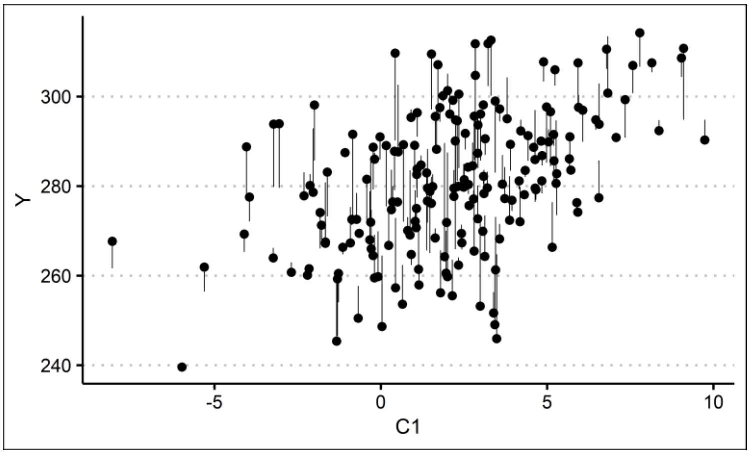

Figure 1: Ploting diagnostic

- The `lm()` function contains also a method for creating diagnostic visuals in base R, that can be displayed as ggplot using `ggfortify`. `ggfortify` is a package to convert automatic plots into plots that are compatible with `ggplot2` functions.
  - The visual diagnostic are useful to assess linearity (*residuals vs fitted*), normality of the residuals (*Normal Q-Q*), homogeneity of variance (*scale-location*), or outliers (*residuals vs leverage*).

```r
library(ggfortify)
```

```
Warning: package 'ggfortify' was built under R version 4.3.1
```

```r
autoplot(fit_1) + ggthemes::theme_clean()
```

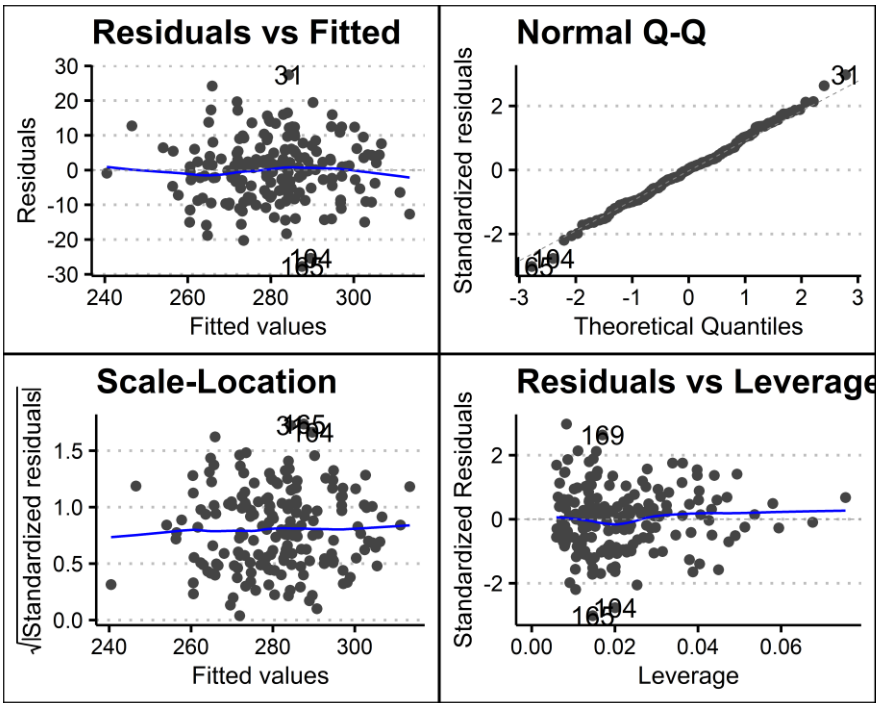

Figure 2: Autoplot Diagnostics

2.1.2 Some other measures of diagnostics

Other diagnostics measures can quickly be produced in R, either with base R, or with well-known packages: cook's distance (`cooks.distance()`), variance inflation factors (`car::vif()`), normality of the residuals, and others:

- Cook's distance:

```
# Extract cooks distance to detect outliers
df$cooks <- cooks.distance(fit_1)
```

- VIFs:

```
# Display variance inflation factors (VIF) to assess multicolinearity issues
car::vif(fit_1)

      C1       C2       C3 
1.021598 1.016482 1.017324 
```

- Skewness

```
# Skewness
moments::skewness(df$resid)

[1] 0.02866212
```

- Kurtosis

```
# Kurtosis
moments::kurtosis(df$resid)

[1] 3.25252
```

- Jarque-Bera Normality Test

```
# Jarque-Bera Normality Test
moments::jarque.test(df$resid)


	Jarque-Bera Normality Test

data:  df$resid
JB = 0.51407, p-value = 0.7733
alternative hypothesis: greater
```

2.1.3 Heteroskedasticity-robust errors

In practice, errors should *almost always* be specified in a manner that is heteroskedasticity and autocorrelation consistent. You can run the Breush-Pagan test for heteroskedasticity with:

```
# Breush-Pagan test
lmtest::bptest(fit_1)


	studentized Breusch-Pagan test

data:  fit_1
BP = 1.7086, df = 3, p-value = 0.635
```

In the example above, we do not have heteroskedasticity issues (*p-value* > 0.05), but we still re-estimate our standard errors with robust standard errors, for the example. The `sandwich` allows for the specification of heteroskedasticity-robust, cluster-robust, and heteroskedasticity and autocorrelation-robust error structures, as follows:

```
# STEP 1: estimate your model
m_example <- lm(Y ~ C1 + C2 + C3, data = df)

# STEP 2: choose an alternative variance structure
lmtest::coeftest(m_example,
                 vcov = sandwich::vcovHC(m_example,
                                         type = "HC1"))


t test of coefficients:

             Estimate Std. Error t value Pr(>|t|)
(Intercept) 246.20225   17.32266 14.2127   <2e-16 ***
C1            2.02646    0.20163 10.0506   <2e-16 ***
C2            2.99031    0.20134 14.8520   <2e-16 ***
C3           -0.21213    0.40696 -0.5213   0.6028
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

- `"HC1"` is one type of robust standard error you may want to use for the model. It is equivalent, in *Stata*, to `vce(robust)` option. Other estimators may be chosen, if more suitable in your case (default is `"HC3"`; see Long & Ervin, 2000[1], for more information).

3. Formatting regression output: stargazer

We have seen, above, that you can display the estimated models with `summary()`. `summary()` is great for quickly displaying results in R. Still, it is not suitable to generate results that you want to copy-paste into reports and presentations: the output formatting is not compelling.

The Stargazer package is a really good option for creating compelling regression and summary output tables. `stargazer` provides ready-made tables that you can export directly to your documents. Below I provide an example:

```
# First, let us estimate all our models
fit_2 <- lm(model_2, data = df)
fit_3 <- lm(model_3, data = df)
fit_4 <- lm(model_4, data = df)

# save the table in an html document, that you can open with Word
stargazer::stargazer(fit_1, fit_2, fit_3, fit_4,
                     type = "html",
                     out = "table_1.html")
```

- `fit_1`, `fit_2`, `fit_3`, and `fit_4` are the models (model objects) you want to display side by side in the table. You can include as many models as you like, before stating the parameters of the `stargazer` function.

[1] Long J. S., Ervin L. H. (2000). "Using Heteroscedasticity Consistent Standard Errors in the Linear Regression Model." *The American Statistician*, 54, 217–224.

- `stargazer` is a great tool for generating nice regression tables. Unfortunately, if you generate Word documents, the output of Stargazer cannot be exported directly. The easiest way to transfer a Stargazer table into Word is:

1. Save the `stargazer` output in an HTML file (with `type = "html"` and `out = "filename.html"`; see previous model).
2. Open the HTML file that you generate into Word (you can do this through the context menu)
3. You can then copy-paste the table into a Word document.

- Looking at the options of Stargazer, you will find many to adjust and customize your output tables, including design for specific journals and publications. Example:

```
stargazer::stargazer(
  # The models we want to display
  fit_1, fit_2, fit_3, fit_4,

  # Tables and labels
  model.numbers = FALSE,
  align = TRUE,

  dep.var.caption  = "Dependent variable: Y",
  dep.var.labels = " ",

  column.labels = c("Model 1", "Model 2", "Model 3", "Model 4"),

  covariate.labels = c("Control 1",
                      "Control 2",
                      "Control 3",
                      "X 1",
                      "X 2"),

  # We want 3 digits
  digits = 3,

  # We want the style of the AER
  #  -> Many other styles are available
  style = "aer",

  # We can also add a custom line of information, including labels or values
  add.lines = list(c('My line', "a value",  "again",  "3", 4)),

  # html file, save in top_table
  type = 'html',
  out = "table_designed.html")
```

- As for the descriptive statistics table we explored in module 3, you can also choose the labels for the columns (usually, the models' names) and the rows (usually, the variables' names). Be aware, nonetheless, that if you mix up the labels, they will appear in the order you state them (not necessarily corresponding to the correct variable you are displaying).

- Often, journals have specific guidelines on how to display tables (including, how many stars to display for p-value < .05, how many decimals, etc.).
  You can choose the style with `style =`. In the example above, I chose *American Economic Review*, but other options include the *American Journal of Sociology*, *Administrative Science Quarterly*, and many others.

4. General linear models with R

Many other models, such as logistic regression, Poisson, etc., are available in the `glm` function (for *general linear models*). Everything works the same way.

4.0.1 Logit and Probit models

- Examples:

```
# Logit model
model_logistic <- glm(Y_dum ~ X1 + X2 + C1 + C2 + C3,
                      data = df,
                      family = binomial(link = "logit"))

# Probit model
model_probit <- glm(Y_dum ~ X1 + X2 + C1 + C2 + C3,
                    data = df,
                    family = binomial(link = "probit"))
```

```
Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
summary(model_logistic)
```

```

Call:
glm(formula = Y_dum ~ X1 + X2 + C1 + C2 + C3, family = binomial(link = "logit"),
    data = df)

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -167.7556    46.5403  -3.605 0.000313 ***
X1             2.8665     0.5696   5.033 4.84e-07 ***
X2             7.0615     2.4466   2.886 0.003898 **
C1             0.9670     0.2160   4.476 7.61e-06 ***
C2             1.3895     0.2737   5.078 3.82e-07 ***
C3            -0.3062     0.2529  -1.211 0.226085
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 254.882  on 183  degrees of freedom
Residual deviance:  60.588  on 178  degrees of freedom
AIC: 72.588

Number of Fisher Scoring iterations: 8
```

- The functions work the same way as we have seen for `lm()`. The only difference is that you use the `family` parameter to define which family of estimate you would like to use. In the examples, we use binomial logit, or binomial probit estimates.

- As for the OLS models, you can use `stargazer` to display the results.
- Then, as for the OLS models, you can estimate robust standard errors using the same tools:

```
library(sandwich)

Warning: package 'sandwich' was built under R version 4.3.1

# Example of robust standard errors
lmtest::coeftest(model_logistic, vcov. = vcovHC, type = "HC1")


z test of coefficients:

              Estimate Std. Error z value  Pr(>|z|)
(Intercept) -167.75557   41.18485 -4.0732 4.636e-05 ***
X1             2.86649    0.45314  6.3258 2.519e-10 ***
X2             7.06154    2.34000  3.0178  0.002547 **
C1             0.96695    0.20371  4.7468 2.067e-06 ***
C2             1.38952    0.27169  5.1143 3.149e-07 ***
C3            -0.30619    0.29985 -1.0212  0.307175
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

4.1 Predictions

- Logit and probit models are not displaying linear relationships. You may want to illustrate your results with predictions for specific or average cases.
- You can compute predictions with predict(), either using the original dataframe you provided to estimate the model, or any other corresponding dataset. Example:

```
# Predict the output of the logistic model for 2 custom observations
pred <- predict(model_logistic,
                newdata = data.frame(
                  X1 = c(18, 19),
                  X2 = c(mean(df$X2), mean(df$X2)),
                  C1 = c(mean(df$C1), mean(df$C1)),
                  C2 = c(mean(df$C2), mean(df$C2)),
                  C3 = c(mean(df$C3), mean(df$C3))
                 ),
                type = "response")

# Display the prediction
pred

        1         2
0.2991520 0.8823788

# Compute the difference in probabilities
diff(pred)

        2
0.5832267
```

- In the example above, `predict()` is used to generate prediction of the model_logistic (first argument of the function)

- With the `newdata` argument, we provide a dataframe with the case we would like to estimate. In the proposed example, the two observations are identical on all points except `X1`. `X1` varies from 18 to 19, while all other parameters are set to the average value of our dataset. Note that the average individual may not exist or be unrealistic. You can choose any other–more suitable value.
- `type = "response"` indicates that we want the predictions to be on the scale of the response variable (in the case of logit models, these are log odds).

4.1.1 Accuracy analysis

- Predicting is nice, but you may want to know how accurate where your predictions. You can compute accuracy measures for sensitivity and specificity:

```
# Predict response values
df$pred_logit <- predict(model_logistic, type = "response")

# Convert probabilities into a dummy variable, as is Y_dum
df$pred_logit <- df$pred_logit >= 0.5

# Note, the confucionMatrix function requires factors as input
caret::confusionMatrix(as.factor(df$Y_dum),
                       as.factor(df$pred_logit))

Confusion Matrix and Statistics

          Reference
Prediction FALSE TRUE
     FALSE    81    8
     TRUE      8   87

               Accuracy : 0.913
                 95% CI : (0.8626, 0.9495)
    No Information Rate : 0.5163
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.8259

 Mcnemar's Test P-Value : 1

            Sensitivity : 0.9101
            Specificity : 0.9158
         Pos Pred Value : 0.9101
         Neg Pred Value : 0.9158
             Prevalence : 0.4837
         Detection Rate : 0.4402
   Detection Prevalence : 0.4837
      Balanced Accuracy : 0.9130

       'Positive' Class : FALSE
```

- In the example above, we estimate sensitivity and specificity in 4 steps:

1. We estimate the model (this is the object called `model_logistic`)
2. We compute predictions (predict response valued): in the example above, we left `newdata` argument empty in the predict function. This means the prediction will be made on the

same data we used to estimate the model. In general, you should create a train and a test sample and analyze the accuracy of your model on the test dataset only. If you do not have a test and a train dataset, you can create those by randomly splitting your original dataset into two groups (during the second module, we saw how to extract a random sample from a dataset)

3. The prediction of the logistic models is probabilities. But our dependent variable is `TRUE` or `FALSE`. In the example, if the predicted probability is above 50% to be `TRUE`, we state it to `TRUE`; otherwise, to `FALSE`. We could choose a different cutoff point.
4. Using the `caret::confusionMatrix()`, we estimate the model's accuracy. Note that the confusion matrix only accepts factors (actual and predicted) with the same levels as inputs. This is why we convert the logical vectors into factors with `as.factor()`.

- In the example above, the predictions are very good: we have 91% true positive (sensitivity) and 92% of true negative (specificity).

4.2 Other `glm` models

- We have seen the above examples of the glm function for logit and probit regression. Other families are possible, including Poisson, gamma, quasi, quasibinomial, and quasipoisson. Other links are possible for the binomial: Cauchy CDF, log, and c-loglog. They are always estimated following the same format. Example:

```
# Example of count data
count_data <- Ecdat::Doctor

# Poisson model
model_poisson <- glm(doctor ~ children + access + health,
                     data = count_data,
                     family = poisson())

summary(model_poisson)


Call:
glm(formula = doctor ~ children + access + health, family = poisson(),
    data = count_data)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  0.37509    0.11016   3.405 0.000662 ***
children    -0.17592    0.03164  -5.560 2.70e-08 ***
access       0.93694    0.19278   4.860 1.17e-06 ***
health       0.28984    0.01825  15.880  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 1766.2  on 484  degrees of freedom
Residual deviance: 1508.8  on 481  degrees of freedom
AIC: 2179.5

Number of Fisher Scoring iterations: 6
```

4.3 Using models from Stata

Sometimes, you may want to use a model from Stata within R, because the specific model you would like to use is better implemented in Stata (this is the case, for example, of some mixed effect models). To call a Stata function from R:

1. Write your Stata script in a `.do` file. An example of a `.do` file could be (Stata code below):

```
reg dv iv iv_2
```

2. call the Stata file from R:

```
# 1. You need to state where Stata is installed on your machine
options("RStata.StataPath" = "\"C:\\Program Files\\Stata17\\StataSE-64\"")
options("RStata.StataVersion" = 17)

# call Stata
RStata::stata(src = "mydofile.do", data.in = df)
```

- where `data.in` points out to your data.frame in R, that you want to load in Stata.
- **Warning:** make sure your variable names in your data.frame are compatible with Stata. See the Module 1 section on styles for more details.