7316 - INTRODUCTION TO R

Module 4: Producing visuals

Teacher: Mickaël Buffart (mickael.buffart@hhs.se)

TABLE OF CONTENTS

This document was generated with R markdown.

Now, you know how to import data, create and clean a dataset, either from files, or from the web. In this module, we will learn how to create visuals from the dataset, including graphs and outputs of regression models.

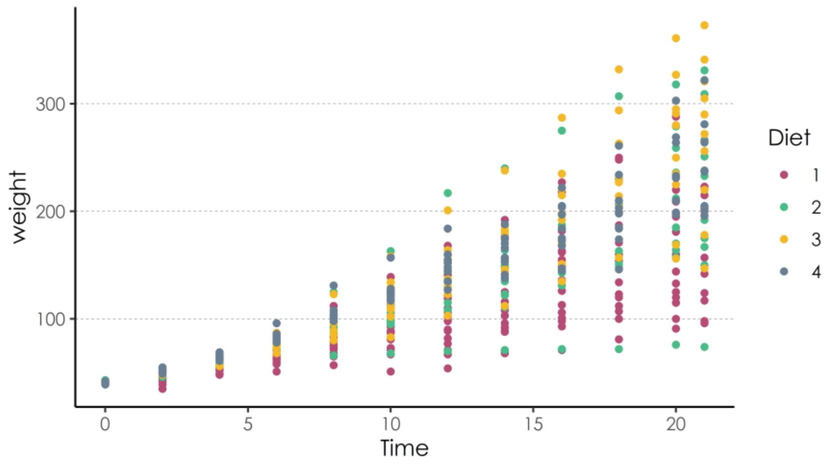
1 Graphs in R

1.1 Data visualization overview

One of the strong points of R is creating very high-quality data visualization. R is very good at both "static" data visualization and interactive data visualization designed for web use. Today, we will cover static data visualization.

1.2 ggplot2 for data visualization

The main package for publication-quality static data visualization in R is `ggplot2`, which is part of the tidyverse collection of packages. With `ggplot2` you control all the elements of your graphs, from the data you visualize and the way you represent it, to the style of the graph and the quality of the output (high-resolution…).



Example of ggplot2 graph, with SSE stylesheet

The workhorse function of ggplot2 is `ggplot()`, response for creating a very wide variety of graphs. The `gg` stands for "*grammar of graphics*". In each `ggplot()` call, the appearance of the graph is determined by specifying:

- the `data`(frame) to be used,
- the `aes`(thetics) of the graph: like size, color, x and y variables,
- the `geom`(etry) of the graph: the chosen representation (*e.g.* `point`, `histogram`, `bar`, `qq`, `curve`, `density`, `abline`, `boxplot`, `map`, etc., etc.)
- You may then add other funtions to your graph, to define, `labs`, `ggtitle`, `ggtheme`, etc.

```
library(ggplot2)

mygraph <- ggplot(mydata) + aes(...) + geom(...) + ...
```

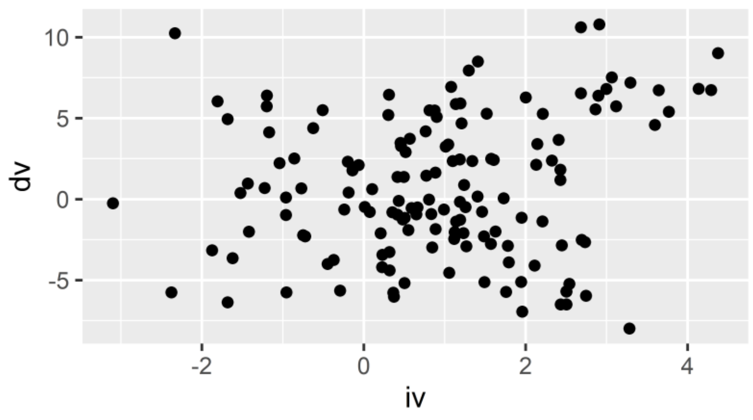
1.3 Scatterplots

First, let's look at a simple scatterplot, which is defined by using the geometry `geom_point()`.

Note: `ggplot2` requires many functions to run. I recommend you in this case to load the full library before using it, or you will spend your time writing `ggplot::` everywhere.

```
df <- rio::import("data/graph_reg.Rds")

# Loading the grammar of graphs
library(ggplot2)
# Defining data
ggplot(df) +
  # defining aesthetics
  aes(x = iv, y = dv) +
  # defining geometry
  geom_point()
```

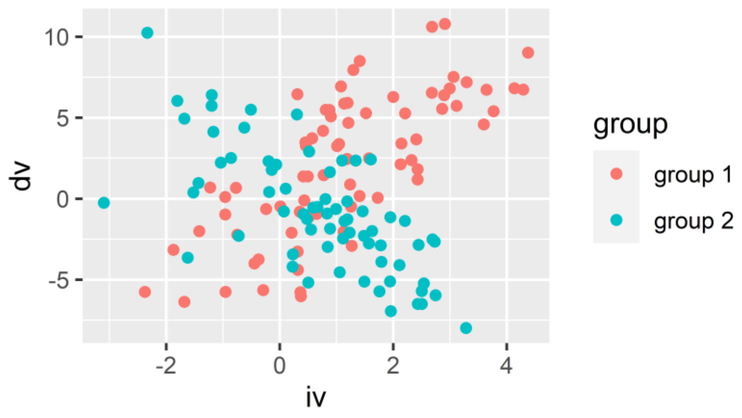


- Each of the graph elements is an R function.
- Functions in `ggplot2` are added up with a `+` symbol.
- Variables names are not quoted (`""`) in `ggplot2`: they are written directly with their full names (here: `iv`, `dv`)

1.3.1 *Colors and shapes*

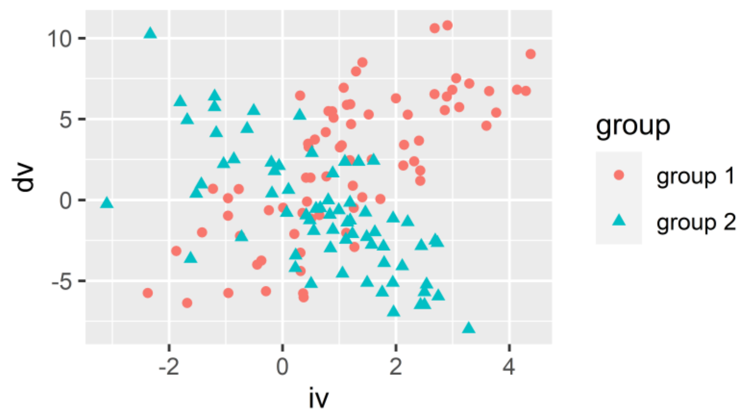
- Graphs can be extensively customized using additional arguments inside of elements, or with additional functions. For example, you may want to **color the element by group**:

```
ggplot(df) + aes(x = iv,
                 y = dv,
                 color = group) +
  geom_point()
```



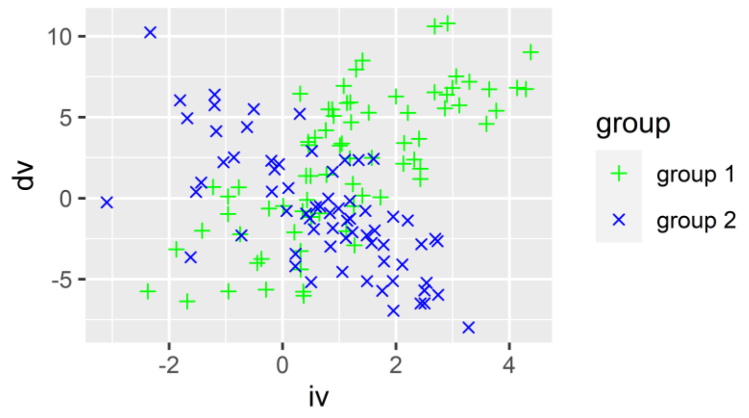
- Or you want to combine it with different shapes, in case you print in black and white:

```
ggplot(df) + aes(x = iv,
                 y = dv,
                 color = group,
                 shape = group) +
  geom_point()
```



Now, if you are unhappy with the choice of color or shapes, you can customize them:

```
ggplot(df) + aes(x = iv,
                 y = dv,
                 color = group,
                 shape = group) +
  geom_point() +
  scale_color_manual(values = c("green", "blue")) +
  scale_shape_manual(values = c(3, 4))
```

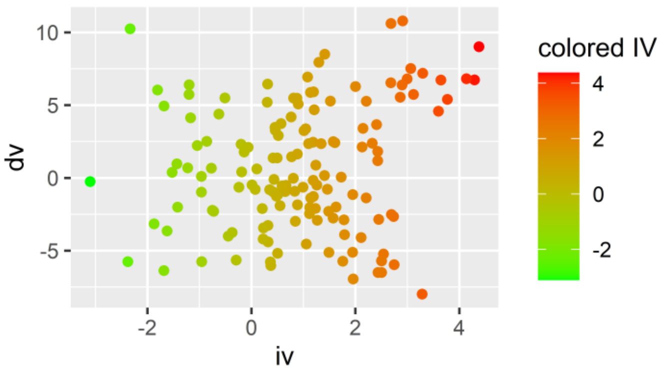


- The color can be written as color names in english, `blue`, `green`, `red`, `black`, etc. or as hexadecimal values, *e.g.* `#FF5733`. You can find a color picker here: https://htmlcolorcodes.com/
- The number of shapes are limited to 25 (numbered from 1 to 25). You can test them as you like, or you can find the list here: http://www.sthda.com/english/wiki/ggplot2-point-shapes
- In both cases, you have to make sure that your color or shape list contains as many values as the number of groups you want to plot (or, it can contain more, but not less).

1.3.2 *Adding gradient colors*

If you have a continuous scale to color, you may want to add gradient.

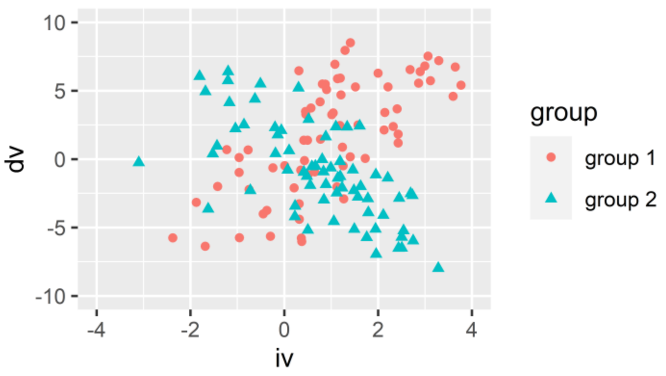
```
ggplot(df) + aes(x = iv,
                 y = dv,
                 color = iv) + geom_point() +
  scale_color_gradient(low = "green", high = "red",
                       name = "colored IV")
```



1.4 Axis and titles

Sometimes, you would like to use a customize scale for the axis:

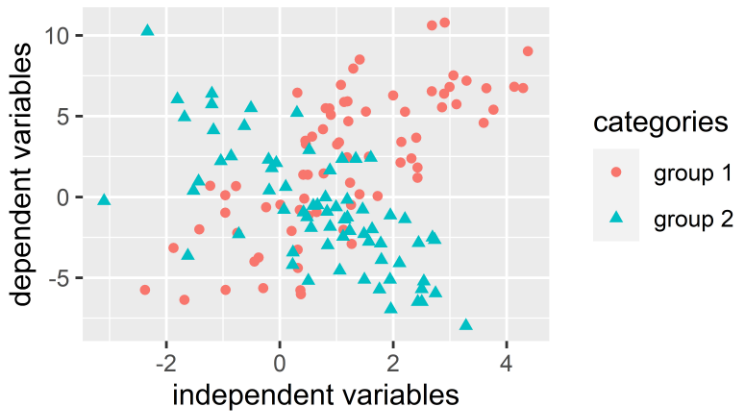
```
ggplot(df) + aes(x = iv,
                 y = dv,
                 color = group,
                 shape = group) +
  geom_point() +
  xlim(-4, 4) + ylim(-10,10)
## Warning: Removed 6 rows containing missing values (geom_point).
```



- **Note** the warning message you get: this is because some data points are out of bounds, hence, not plotted on the graph.

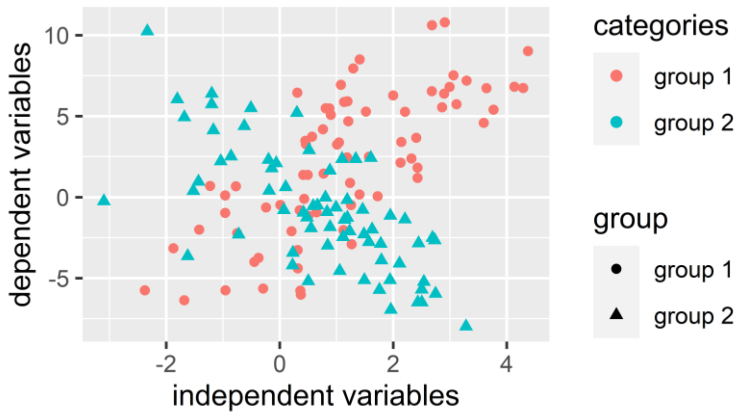
You can also change the name of axis, for example, using full labels:

```
ggplot(df) + aes(x = iv,
                 y = dv,
                 color = group,
                 shape = group) +
  geom_point() +
  labs(x = "independent variables",
       y = "dependent variables",
       color = "categories",
       shape = "categories")
```



- **Note** that you have to indicate both `color` and `shape` in your graph to change the title of your legend, because you graph include colors and shapes. If you change only one, the legen will be repeated twice: once with the new label you wrote, one with the variable name. Example:

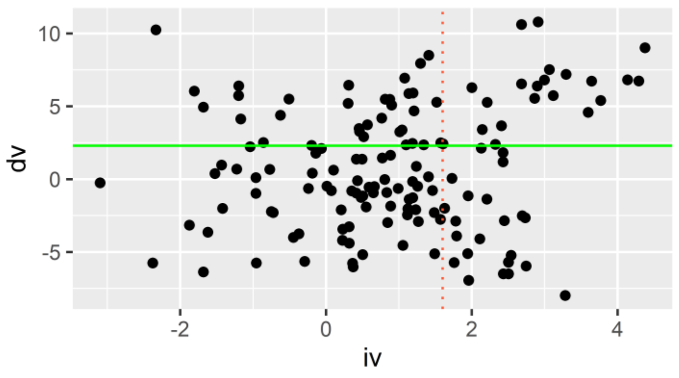
```
ggplot(df) + aes(x = iv,
                 y = dv,
                 color = group,
                 shape = group) +
  geom_point() +
  labs(x = "independent variables",
       y = "dependent variables",
       color = "categories")
```



1.4.1 Lines

To plot a straight line somewhere on your graph, you can use `geom_vline()` (vertical) or `hline` (horizontal):

```
ggplot(df) + aes(x = iv,
                 y = dv) +
  geom_point() +
  geom_vline(xintercept = 1.6,
             color = "#FF5733",
             linetype = "dotted") +
  geom_hline(yintercept = 2.3,
             color = "green",
             linetype = "solid")
```

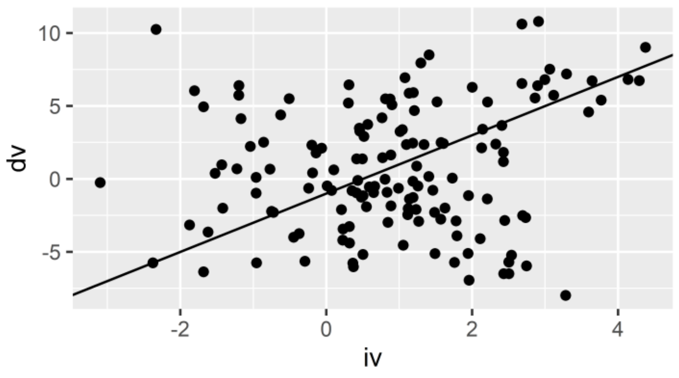


- Here again, you can customize the position, color, or linetype. The seven styles of linetypes are available here.

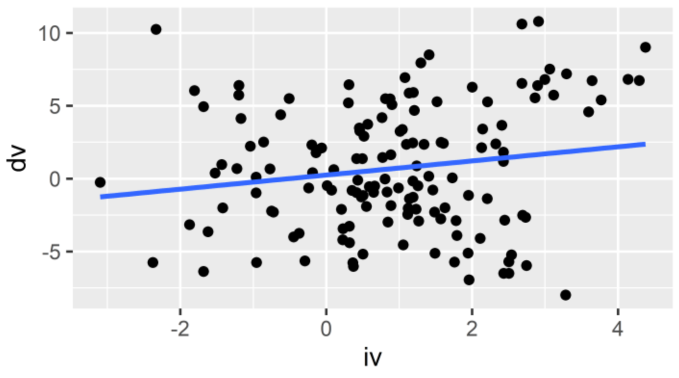
1.4.2 ablines and regression lines

You can also plot a line accross the graph, indicated slope and intercept, using `abline()`, or you can plot a regression lines fitted to your data, using `geom_smooth()`:

```
# With abline (not fitted line)
ggplot(df) +
  aes(x = iv, y = dv) +
  geom_point() +
  geom_abline(intercept = -1,
              slope = 2)
```



```
# With geom_smooth (fitted)
ggplot(df) + aes(x = iv,
                 y = dv) +
  geom_point() +
  geom_smooth(method = "lm",
              formula = "y ~ x", se = FALSE)
```



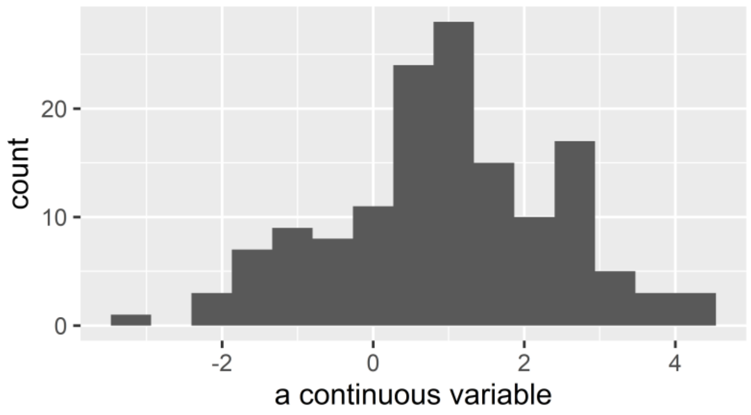
The parameter of `geom_smooth`:

- `method`: the model to use to fit the data (here, OLS model)
- `formula`: the formula used to fit the model (here, y on x, but more complex formula could be written, as we will see)
- `se`: a logical whether standard errors should be plotted or not.

1.5 Bars and histograms

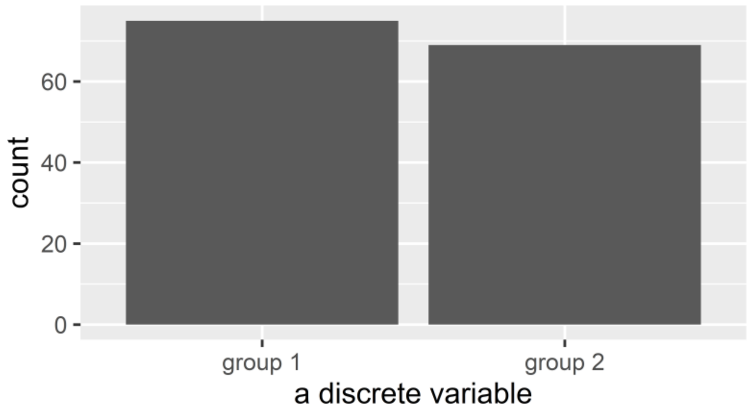
For continuous variables, you may want to plot an histogram with `geom_histogram()`.

```r
ggplot(df) +
  aes(x = iv) +
  geom_histogram(bins = 15) +
  labs(x = "a continuous variable")
```

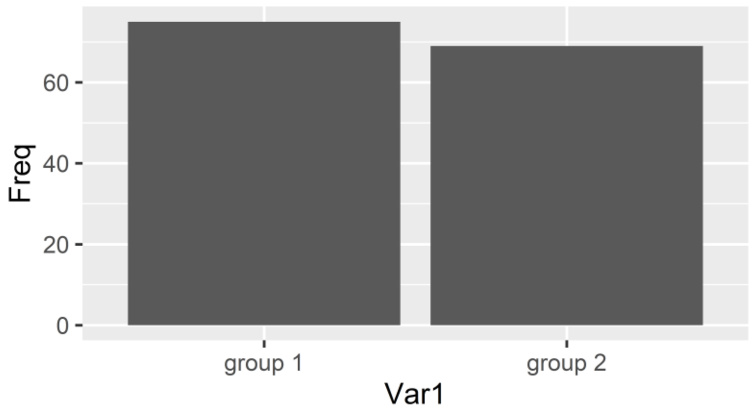


`bar_plot()` is the same, but for discrete variables. By default, a bar plot uses frequencies for its values, but you can use values from a column by specifying `stat = "identity"` inside `geom_bar()`.

```r
# Bar plot with frequency
ggplot(df) +
  aes(x = group) +
  geom_bar() +
  labs(x = "a discrete variable")
```



```r
tmp <- as.data.frame(table(df$group))
ggplot(tmp) +
  aes(x = Var1, y = Freq) +
  geom_bar(stat = "identity")
```

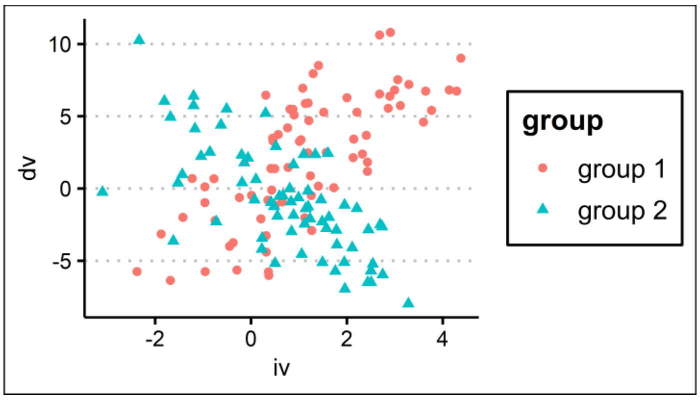


1.6 Adding themes

Another option to affect the appearance of the graph is to use **themes**, which affect a number of general aspects concerning how graphs are displayed.

- Some default themes come installed with ggplot2/tidyverse, but some of the best in my opinion come from the package ggthemes. You can see the gallery of themes here.
- To apply a theme, just add `+ themename()` to your ggplot graphic, or `+ ggthemes::themename()`.

```
ggplot(df) + aes(x = iv,
                 y = dv,
                 color = group,
                 shape = group) +
  geom_point() + ggthemes::theme_clean()
```



1.7 High quality graph

Once your plot is ready, if you want to save it in high resolution, you can use:

```
png(filename = "figure_1.png",
    unit  = "cm", width = 12, height = 12,
    res = 800)

ggplot(df) + aes(x = iv,
                 y = dv,
                 color = group,
                 shape = group) +
  geom_point() + ggthemes::theme_clean()

dev.off()
```

- where `width` and `height` are the size of your plot, saved in `figure_1.png`, and `res` is the resolution (*i.e.* the quality) of the image. By default, the resolution of plots is 150 dpi. A higher value is preferable.

1.8 More with ggplot2

This has been just a small overview of things you can do with ggplot2. To learn more about it, you can read the book in the syllabus about `ggplot2`: https://ggplot2-book.org/. You may also want to read the STHDA Guide to ggplot2, a good general guide to ggplot2 that is still pretty thorough. Finally, the RStudio cheat sheet may help you move further.

2 Regression Basics

The basic method of performing an OLS regression in R is to the 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, or store it in an object, or use it for prediction, extracting residuals, etc.
To run an OLS regression, write:

```
model_0 <- lm(dv ~ iv, data = df)
```

- `dv ~ iv` is **the formula**. It tells to regress y on x.
- You can then visualize the output of your model:

```
summary(model_0)
##
## Call:
## lm(formula = dv ~ iv, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -9.8213 -3.1418 -0.1075  3.3896 11.1311
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   0.2478     0.4240   0.584   0.5599
## iv            0.4845     0.2434   1.990   0.0485 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.287 on 142 degrees of freedom
## Multiple R-squared:  0.02714,    Adjusted R-squared:  0.02029
## F-statistic: 3.961 on 1 and 142 DF,  p-value: 0.04849
```

- You could also extract residuals into another object:

```
resid <- model_0$residuals
```

2.1 R formulae

`dv ~ iv`, in the model above, is **the formula**. You can write formula for all kinds of mathematical models. Bellow, I give some example of model complex formulae.

```
# multiple terms
model_example <- lm(dv ~ iv + iv_2, data = df)
```

```
# quadratic terms
model_example <- lm(dv ~ iv + I(iv^2), data = df)
summary(model_example)
```

- In the example above, `I()` means *as is*. It avoids R thinking that we included twice the same variable in the model.

```
# Interaction term without the main term
model_1 <- lm(dv ~ iv:group, data = df)
summary(model_1)
##
## Call:
## lm(formula = dv ~ iv:group, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -6.8952 -2.1301  0.0882  1.9296  5.9065
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)         0.1379     0.2816   0.490    0.625
## iv:groupgroup 1     1.9679     0.1956  10.058  < 2e-16 ***
## iv:groupgroup 2    -1.8010     0.2345  -7.682  2.4e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.847 on 141 degrees of freedom
## Multiple R-squared:  0.5741, Adjusted R-squared:  0.5681
## F-statistic: 95.04 on 2 and 141 DF,  p-value: < 2.2e-16

# Interaction term with the main term
model_2 <- lm(dv ~ iv * group, data = df)
summary(model_2)
##
## Call:
## lm(formula = dv ~ iv * group, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -7.1682 -2.0663  0.0633  2.0730  6.0343
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)        -0.2262     0.4127  -0.548    0.584
## iv                  2.0845     0.2180   9.563   <2e-16 ***
## groupgroup 2        0.6797     0.5638   1.205    0.230
## iv:groupgroup 2    -3.9775     0.3288 -12.096   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.842 on 140 degrees of freedom
## Multiple R-squared:  0.5785, Adjusted R-squared:  0.5695
## F-statistic: 64.05 on 3 and 140 DF,  p-value: < 2.2e-16
```

- You may also regress your DV on all the variable in your dataframe:

```
# Regress on all variables in df
model_3 <- lm(dv ~ ., data = df)
```

2.2 Formatting regression output: tidyr

With the `tidy()` function from the `broom` package, you can easily create standard regression output tables.

```
library(broom)
tidy(model_2)
## # A tibble: 4 x 5
##   term             estimate std.error statistic  p.value
##   <chr>               <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)        -0.226     0.413    -0.548 5.84e- 1
## 2 iv                  2.08      0.218     9.56  5.52e-17
## 3 groupgroup 2        0.680     0.564     1.21  2.30e- 1
## 4 iv:groupgroup 2    -3.98      0.329   -12.1   1.66e-23
```

2.3 Formatting regression output: stargazer

Another really good option for creating compelling regression and summary output tables is the `stargazer` package.

```
stargazer::stargazer(model_0, model_2, model_3,
                     type = 'html', out = "test.html")
```

- Looking at the options of stargazer, you will find many options to adjust and customize your output tables.

2.4 Useful output from regression

A couple of useful data elements that are created with a regression output object are fitted values and residuals. You can easily access them as follows:

- **Residuals:** Use the residuals() function.

```
resid_model_2 <- model_2$residuals
```

- **Predicted values:** Use the fitted() function.

```
resid_model_2 <- model_2$fitted.values
```

2.5 More complex models

Many other models, such as a logistic regression, poisson, etc., are available in the glm function. Everything works the same way. For example:

```
# Logit model
model_logistic <- glm(as.factor(group) ~ iv + iv_2,
                      data = df,
                      family = binomial(link = "logit"))

# Logit model
model_probit <- glm(as.factor(group) ~ iv + iv_2,
                    data = df,
                    family = binomial(link = "probit"))

# Poisson model
model_poisson <- glm(discrete_var ~ iv + iv_2,
                     data = df,
                     family = poisson())
summary(model_poisson)
```

2.6 Specifying the variance structure

In practice, errors should *almost always* be specified in a manner that is heteroskedasticity and autocorrelation consistent.

The sandwich allows for specification of heteroskedasticity-robust, cluster-robust, and heteroskedasticity and autocorrelation-robust error structures.

2.6.1 *Heteroskedasticity-robust errors*

```
# STEP 1: estimate your model
model_robust <- lm(dv ~ iv + iv_2 + group, data = df)

lmtest::coeftest(model_robust,
         vcov = sandwich::vcovHC(model_robust,
         type = "HC1"))
##
## t test of coefficients:
##
##                 Estimate Std. Error t value  Pr(>|t|)
## (Intercept)    1.8171071  0.6413224  2.8334  0.005287 **
## iv             0.3301019  0.2946912  1.1202  0.264563
## iv_2          -0.0055486  0.0352013 -0.1576  0.874979
## groupgroup 2 -2.9095565  0.6831102 -4.2593 3.741e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

- `"HC1"` is the type of robust standard error you want to use for the model. You may choose another one.

2.7 Using models from Stata

Sometimes, you may want to use 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. Example of a `.do` file could be (Stata code bellow):

```
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 dataframe 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.

3 TO DO BEFORE THE NEXT CLASS

1. Go through this material again and try other cases: run all the codes for yourself, change the values, change the parameters, see what happens.
2. If you haven't finished during the class, finish the practice assignment of Module 4.
3. If you want: create a github account for yourself, and install Github desktop on your laptop. We will learn how to use them next class.

Next time, we will move on, and consider the knowledge provided here is mastered.