R-Introduction - Part 2

A Workshop for Modul University Vienna Faculty

Gunther Maier

Table of contents

1 Introduction

This is the second part of a very brief introduction to R. I compiled this information for a presentation to faculty at Modul University Vienna.

In this part I assume that you are familiar with the topics we covered in the first workshop. This includes topics like - the difference between R and RStudio - The main parts of RStudio - What are packages and where can I get them? - How to install a package - How to get help

I will repeat the parts we did about loading data.

2 Loading data

2.1 Using data available in R and R packages

R comes with a set of data-sets that are handy for testing and demonstration purposes. The data() command without parameters generates a list of all the datasets (the output goes to a separate window and therefore does not show in Quarto).

```
data()
```

If you want to see **all** datasets available in **all** installed packages, use the following command. It is mentioned at the bottom of the output of "data()". Again, the output does not show in Quarto,

```
data(package = .packages(all.available = TRUE))
```

One of the datasets available by default (in the standard package "datasets") is "mtcars". To use this dataset in our analysis, we type

```
data("mtcars")
```

This loads the dataset "mtcars" into the environment. Check the Environment pane in the Environment window. You should see an entry named "mtcars".

The data is now available as a dataframe in your R session. To investigate the data, we can list the first six lines with

```
head(mtcars)
```

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

In one webpage we saw that there is also a dataset "mpg" with information about fuel consumption of various cars. We try to load this datase with

```
data("mpg")
```

```
Warning in data("mpg"): data set 'mpg' not found
```

This command generates an error message. The reason is that the package "ggplot2", which contains this dataset is not available in our R session yet. If we do not want to load this package, we can set the "package" parameter in the data() command:

```
data(mpg, package="ggplot2")
```

Alternatively, we can first load the package and then the dataset.

```
library(ggplot2)
data(mpg)
```

Again, we list the head of the dataframe.

```
head(mpg)
```

```
# A tibble: 6 x 11
  manufacturer model displ  year   cyl trans      drv     cty   hwy fl    class
  <chr>        <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>
1 audi         a4      1.8  1999     4 auto(l5)   f        18    29 p     compa~
2 audi         a4      1.8  1999     4 manual(m5) f        21    29 p     compa~
3 audi         a4      2    2008     4 manual(m6) f        20    31 p     compa~
4 audi         a4      2    2008     4 auto(av)   f        21    30 p     compa~
5 audi         a4      2.8  1999     6 auto(l5)   f        16    26 p     compa~
6 audi         a4      2.8  1999     6 manual(m5) f        18    26 p     compa~
```

2.2 Importing an Excel-file

We first have to load the package `readxl`. The package contains a function `read_excel`. Take a look at the description of this function through `?read_excel`.

```
library(readxl)
```

To demonstrate and test, we need data in Excel format. Let us use the Austrian results of the EU-elections 2014. Go to the webpage https://www.data.gv.at/, and select "Daten" - "Datensatz finden" (you may want to witch to English and then select "data" - "find dataset"). Enter "EU Wahl 2014" to the search field and hit Enter. Scroll down to "Ergebnisse der EU-Wahl 2014 (BMI)" and click the green button with the white "X". This will download the Excel-file to your download directory. Move or copy the file to your project directory.

Now, run

```
EU_elections <- read_excel(
  "eu-wahl2014-endgueltiges_ergebnis_mit_briefwahl.xlsx")
```

```
New names:
* `` -> `...6`
* `` -> `...7`
* `%` -> `%...9`
* `%` -> `%...11`
* `%` -> `%...13`
* `%` -> `%...15`
* `%` -> `%...17`
* `%` -> `%...19`
* `%` -> `%...21`
* `%` -> `%...23`
* `%` -> `%...25`
```

R alerts us that some of the variables were given new names. To view the first six lines and five columns of the dataset, enter

```
head(EU_elections, c(6, 5))
```

```
# A tibble: 6 x 5
  GKZ    Gebietsname       `Wahlbe-\r\nrechtigte` Wahl-\r\nbeteil-\r\n~1 Stimmen
  <chr>  <chr>                              <dbl>                  <dbl> <chr>
1 <NA>   <NA>                                  NA                     NA "abge-~
2 G00000 Österrreich                      6410602                  0.454 "29094~
3 G10000 Burgenland                        233920                  0.537 "12553~
4 G1A000 Burgenland Nord                   122224                  0.533 "65136"
5 G1A099 Wahlkarten - Bur~                      0                     NA "4902"
6 G1B000 Burgenland Süd                    111696                  0.541 "60397"
# i abbreviated name: 1: `Wahl-\r\nbeteil-\r\nigung\r\nin %`
```

The parameter `c(6, 5)` sets the number of rows to display to 6 and the number of columns to 5.

Alternatively, you may want to load the whole data frame into the viewer with

```
View(EU_elections)
```

When we look at the data, we find that there are some problems. Underneath the variable names, we see a row with mainly "NA" values. In columns 5-7 we see that there seems to be a second row for the variable header. Because of the strings in these fields, all the values in columns 5-7 are stored as characters (indicated by the "").

To avoid these problems, we can tell the function `read_excel()` to skip the first two lines.

```
EU_elections <- read_excel(
  "eu-wahl2014-endgueltiges_ergebnis_mit_briefwahl.xlsx",
```

```
    skip=2)
  head(EU_elections, c(6, 7))
```

```
# A tibble: 6 x 7
  G00000 Österrreich `6410602` `0.45385706365798406` `2909497` `85936` `2823561`
  <chr>  <chr>           <dbl>                 <dbl>     <dbl>   <dbl>     <dbl>
1 G10000 Burgenland     233920                 0.537    125533    5189    120344
2 G1A000 Burgenland~    122224                 0.533     65136    2655     62481
3 G1A099 Wahlkarten~         0                NA          4902     113      4789
4 G1B000 Burgenland~    111696                 0.541     60397    2534     57863
5 G1B099 Wahlkarten~         0                NA          5259     113      5146
6 G10099 Wahlkarten~         0                NA         10161     226      9935
```

Now, we have the problem that the first line of data is used as variable names. So, we also tell the function to **not** read column names

```
  EU_elections <- read_excel(
    "eu-wahl2014-endgueltiges_ergebnis_mit_briefwahl.xlsx",
    skip=2,
    col_names = FALSE)
```

```
New names:
* `` -> `...1`
* `` -> `...2`
* `` -> `...3`
* `` -> `...4`
* `` -> `...5`
* `` -> `...6`
* `` -> `...7`
* `` -> `...8`
* `` -> `...9`
* `` -> `...10`
* `` -> `...11`
* `` -> `...12`
* `` -> `...13`
* `` -> `...14`
* `` -> `...15`
* `` -> `...16`
* `` -> `...17`
* `` -> `...18`
* `` -> `...19`
* `` -> `...20`
* `` -> `...21`
* `` -> `...22`
```

```
* `` -> `...23`
* `` -> `...24`
* `` -> `...25`
```

```
head(EU_elections, c(6, 7))
```

```
# A tibble: 6 x 7
  ...1   ...2                            ...3   ...4    ...5  ...6    ...7
  <chr>  <chr>                          <dbl>  <dbl>   <dbl> <dbl>   <dbl>
1 G00000 Österrreich                  6410602  0.454 2909497 85936 2823561
2 G10000 Burgenland                    233920  0.537  125533  5189  120344
3 G1A000 Burgenland Nord               122224  0.533   65136  2655   62481
4 G1A099 Wahlkarten - Burgenland Nord       0 NA        4902   113    4789
5 G1B000 Burgenland Süd                111696  0.541   60397  2534   57863
6 G1B099 Wahlkarten - Burgenland Süd        0 NA        5259   113    5146
```

Instead of `col_names = FALSE`, we can also provide a vector of variable names to the parameter `col_names`. This will give us a nice data frame to use.

```
EU_elections <- read_excel(
  "eu-wahl2014-endgueltiges_ergebnis_mit_briefwahl.xlsx",
  skip=2,
  col_names = c("GKZ", "Name", "Voters", "Turnout", "Votes", "invalid",
                "valid", "ÖVP", "ÖVP_percent", "SPÖ", "SPÖ_percent",
                "FPÖ", "FPÖ_percent", "GRÜNE", "GRÜNE_percent", "BZÖ",
                "BZÖ_percent", "NEOS", "NEOS_percent", "RECOS",
                "RECOS_percent", "ANDERS", "ANDERS_percent", "EUSTOP",
                "EUSTOP_percent"))
head(EU_elections, c(6, 7))
```

```
# A tibble: 6 x 7
  GKZ    Name                          Voters Turnout   Votes invalid   valid
  <chr>  <chr>                          <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
1 G00000 Österrreich                  6410602   0.454 2909497   85936 2823561
2 G10000 Burgenland                    233920   0.537  125533    5189  120344
3 G1A000 Burgenland Nord               122224   0.533   65136    2655   62481
4 G1A099 Wahlkarten - Burgenland Nord       0  NA        4902     113    4789
5 G1B000 Burgenland Süd                111696   0.541   60397    2534   57863
6 G1B099 Wahlkarten - Burgenland Süd        0  NA        5259     113    5146
```

2.3 Importing a CSV-file directly from the Internet

We pack two new topics under this heading: (1) reading CSV-files and (2) reading data directly from the Internet.

CSV is a common data format and stands for "Comma Separated Variables". This name, however, is misleading because in German speaking countries it is usually the semicolon (;), not the comma that separates variables.

R provides two functions for reading a CSV file: `read.csv()` and `read.csv2()`. The first version uses English standards (comma as variable separator and period as decimal separator), the second version uses German standards (semicolon as variable separator and comma as decimal separator). In both cases, you can define the respective separators with the parameters `sep=` and `dec=`.

Some data import functions in R allow you to specify a url instead of a local filepath for the file. The two CSV-functions are among them, `read_excel()`is not. When you go back to the https://www.data.gv.at/ webpage, to the entry "Ergebnisse der EU-Wahl 2014 (BMI)", you will see an orange icon marked "CSV". Right-click this icon and select Copy Link to copy the url into your computer's clipboard. Paste this url into the following call of `read.csv2()`.

```
csvurl <- paste("https://www.data.gv.at/katalog/dataset/",
                "2b10a91b-51d5-4e34-b992-8fd3a3121f0d/resource/",
                "a5ccea30-a505-4376-bfa3-fa2585c69192/download/",
                "eu-wahl2014-endgueltiges_ergebnis_mit_briefwahl.csv",
            sep="")
EU_elections_csv <- read.csv2(csvurl)
```

This does not work. The reason is again the issue with the column names, which are not unique. We can again skip the first two lines with `skip = 2`. Unfortunately, this function does not allow us to provide variable names as before. We can only set `header = FALSE` to prevent the function from interpreting the first line as variable names.

```
csvurl <- paste("https://www.data.gv.at/katalog/dataset/",
                "2b10a91b-51d5-4e34-b992-8fd3a3121f0d/resource/",
                "a5ccea30-a505-4376-bfa3-fa2585c69192/download/",
                "eu-wahl2014-endgueltiges_ergebnis_mit_briefwahl.csv",
            sep="")
EU_elections_csv <- read.csv2(csvurl,
    skip=2,
    header = FALSE)
head(EU_elections_csv, c(6, 7))
```

```
     V1                              V2     V3     V4     V5   V6     V7
1 G10000                     Burgenland 233920 53,66% 125533 5189 120344
2 G1A000                 Burgenland Nord 122224 53,29%  65136 2655  62481
3 G1A099   Wahlkarten - Burgenland Nord      0          4902  113   4789
4 G1B000               Burgenland S\xfcd 111696 54,07%  60397 2534  57863
5 G1B099 Wahlkarten - Burgenland S\xfcd      0          5259  113   5146
6 G10099         Wahlkarten - Burgenland      0         10161  226   9935
```

Note that the "ü" in "Burgenland Süd" is not read correctly. This issue is related to the import of the file from the Internet. German Umlaut characters are treated differently in various character systems, socalled "encodings". To fix this, we need to use the parameter `encoding` and specify the correct encoding. An alternative fix would be to download the file to the local machine and then read it into R in a second step.

We use the following code to read the file with the correct encoding:

```
csvurl <- paste("https://www.data.gv.at/katalog/dataset/",
                "2b10a91b-51d5-4e34-b992-8fd3a3121f0d/resource/",
                "a5ccea30-a505-4376-bfa3-fa2585c69192/download/",
                "eu-wahl2014-endgueltiges_ergebnis_mit_briefwahl.csv",
            sep="")
EU_elections_csv <- read.csv2(csvurl,
    skip=2,
    header = FALSE,
    encoding = "latin1")
head(EU_elections_csv, c(6, 7))
```

```
     V1                            V2     V3     V4     V5   V6     V7
1 G10000                   Burgenland 233920 53,66% 125533 5189 120344
2 G1A000               Burgenland Nord 122224 53,29%  65136 2655  62481
3 G1A099 Wahlkarten - Burgenland Nord      0           4902  113   4789
4 G1B000                Burgenland Süd 111696 54,07%  60397 2534  57863
5 G1B099  Wahlkarten - Burgenland Süd      0           5259  113   5146
6 G10099       Wahlkarten - Burgenland      0          10161  226   9935
```

Now the Umlaut characters are displayed correctly.

With `read.csv2()`, we cannot specify the variable names and have to set the correct names in a separate step. With the function `colnames()` we can get and set the column names of a data frame.

```
colnames(EU_elections_csv) <- c("GKZ", "Name", "Voters", "Turnout",
      "Votes", "invalid", "valid", "ÖVP", "ÖVP_percent", "SPÖ",
      "SPÖ_percent", "FPÖ", "FPÖ_percent", "GRÜNE", "GRÜNE_percent",
      "BZÖ", "BZÖ_percent", "NEOS", "NEOS_percent", "RECOS",
      "RECOS_percent", "ANDERS", "ANDERS_percent", "EUSTOP",
      "EUSTOP_percent")
head(EU_elections_csv, c(6, 7))
```

```
     GKZ                          Name Voters Turnout  Votes invalid  valid
1 G10000                   Burgenland 233920  53,66% 125533    5189 120344
2 G1A000               Burgenland Nord 122224  53,29%  65136    2655  62481
3 G1A099 Wahlkarten - Burgenland Nord      0           4902     113   4789
```

4	G1B000	Burgenland Süd	111696	54,07%	60397	2534	57863
5	G1B099	Wahlkarten - Burgenland Süd	0		5259	113	5146
6	G10099	Wahlkarten - Burgenland	0		10161	226	9935

3 Viewing data

In any statistical software, it is always a good idea, to check and validate your data before you use it. That the routine you used to read the data does not generate an error message, does not imply that your data has been read *correctly*. Some data may be distorted, read in the wrong format, some numbers may end up in the wrong variables, etc.

I do not want to go into depth about checking and verifying data. At this stage, I just want to collect a few tools that you can use to look at your data.

3.1 `head()`, `tail()`, and `data.table()`

We have already seen the command `head()` at work above. By default, it shows the first 6 rows for all the variables in the data frame. When the variables do not fit horizontally, the function prints them in blocks underneath one another. This output may be somewhat confusing and difficult to read. As you saw above, you can explicitly specify the numbers of rows and columns to display.

The command `tail()` does the same as `head()`, but with the last six lines of the dataframe. The function `data.table()`, which is available in the package "data.table", combines the functionality of `head()` and `tail()` and shows the first and the last six lines of the dataset.

3.2 `View()`

The command `View()` (note the capital "V") displays the data frame in a spreadsheet like form. The output is placed in a new tab in the Source window of RStudio. This is probably the best option to inspect your data because you can scroll through all the rows and columns. The function may fail, however, when you have to deal with a very large data frame.

When you move the cursor over the name of a column, you will see more information about this variable in a tooltip. Run `View(EU_elections)`, place the cursor over "Turnout", and after a second you will see that this is column 4 of the data frame, that the format is "numeric" and that the numbers are in the range "0.1 - 0.8".

You can call this function also from the Environment pane of the Environment window. To the right of every listed data frame there is a light-blue icon with a data grid. Clicking that will load this data frame into `View()`.

3.3 The glimpse command

Somewhere between `head()` and `tail()` on the one hand and `View()` on the other is the `glimpse()` command from the package "dplyr". This package is loaded automatically when you load the package "tidyverse" (see below).

```
library(dplyr)
```

```
Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

```r
glimpse(EU_elections)
```

```
Rows: 2,707
Columns: 25
$ GKZ            <chr> "G00000", "G10000", "G1A000", "G1A099", "G1B000", "G1B0~
$ Name           <chr> "Österrreich", "Burgenland", "Burgenland Nord", "Wahlka~
$ Voters         <dbl> 6410602, 233920, 122224, 0, 111696, 0, 0, 10479, 10479,~
$ Turnout        <dbl> 0.4538571, 0.5366493, 0.5329232, NA, 0.5407266, NA, NA,~
$ Votes          <dbl> 2909497, 125533, 65136, 4902, 60397, 5259, 10161, 5681,~
$ invalid        <dbl> 85936, 5189, 2655, 113, 2534, 113, 226, 167, 157, 10, 4~
$ valid          <dbl> 2823561, 120344, 62481, 4789, 57863, 5146, 9935, 5514, ~
$ ÖVP            <dbl> 761896, 37331, 18266, 1342, 19065, 1674, 3016, 2111, 18~
$ ÖVP_percent    <dbl> 0.2698351, 0.3102024, 0.2923449, 0.2802255, 0.3294852, ~
$ SPÖ            <dbl> 680180, 40361, 20740, 1648, 19621, 1621, 3269, 1126, 98~
$ SPÖ_percent    <dbl> 0.2408944, 0.3353802, 0.3319409, 0.3441219, 0.3390941, ~
$ FPÖ            <dbl> 556835, 21446, 11485, 632, 9961, 617, 1249, 806, 746, 6~
$ FPÖ_percent    <dbl> 0.1972102, 0.1782058, 0.1838159, 0.1319691, 0.1721480, ~
$ GRÜNE          <dbl> 410089, 9686, 5581, 570, 4105, 603, 1173, 777, 691, 86,~
$ GRÜNE_percent  <dbl> 0.14523823, 0.08048594, 0.08932315, 0.11902276, 0.07094~
$ BZÖ            <dbl> 13208, 280, 154, 12, 126, 10, 22, 15, 12, 3, 4, 4, 0, 4~
$ BZÖ_percent    <dbl> 0.004677781, 0.002326664, 0.002464749, 0.002505742, 0.0~
$ NEOS           <dbl> 229781, 5955, 3341, 366, 2614, 385, 751, 422, 375, 47, ~
$ NEOS_percent   <dbl> 0.08137986, 0.04948315, 0.05347226, 0.07642514, 0.04517~
$ RECOS          <dbl> 33224, 916, 493, 37, 423, 31, 68, 60, 57, 3, 1, 1, 0, 1~
$ RECOS_percent  <dbl> 0.011766702, 0.007611514, 0.007890399, 0.007726039, 0.0~
$ ANDERS         <dbl> 60451, 1393, 781, 63, 612, 81, 144, 73, 68, 5, 7, 7, 0,~
$ ANDERS_percent <dbl> 0.021409490, 0.011575151, 0.012499800, 0.013155147, 0.0~
$ EUSTOP         <dbl> 77897, 2976, 1640, 119, 1336, 124, 243, 124, 112, 12, 2~
$ EUSTOP_percent <dbl> 0.02758821, 0.02472911, 0.02624798, 0.02484861, 0.02308~
```

This function displays the size of the data frame (number of rows and number of columns) and then for every variable its name, its data type, and the first few values (whatever fits into the available horizontal space).

3.4 Nicer looking tables

There are a number of packages that provide functions for nicely formatted tables. One example is the function `gt()` from the "gt" package. Try it out with the following code.

```
library(gt)
gt(EU_elections[1:8, 1:7])
```

GKZ	Name	Voters	Turnout	Votes	invalid	valid
G00000	Österrreich	6410602	0.4538571	2909497	85936	2823561
G10000	Burgenland	233920	0.5366493	125533	5189	120344
G1A000	Burgenland Nord	122224	0.5329232	65136	2655	62481
G1A099	Wahlkarten - Burgenland Nord	0	NA	4902	113	4789
G1B000	Burgenland Süd	111696	0.5407266	60397	2534	57863
G1B099	Wahlkarten - Burgenland Süd	0	NA	5259	113	5146
G10099	Wahlkarten - Burgenland	0	NA	10161	226	9935
G10100	Eisenstadt(Stadt)	10479	0.5421319	5681	167	5514

I added `[1:8, 1:7]` to the name of the data frame to restrict the table to the first eight rows and the first seven columns. This is solely for display purposes for the PDF-output. When you leave this out and use `gt(EU_elections)`, the whole data frame will be shown in the table.

When you call this function from the Console, the output will show up in the Viewer pane of the Files window. Click the Zoom button to get a larger view in a separate window. With the Export button you can also save the table as a graphic, as a webpage, or copy it to the clipboard to paste it into another program.

4 Managing data

When we have read our data into R, we usually have to do some data manipulations before we can run some analysis. For these tasks, we will use the package "tidyverse" that is quite popular for these tasks. One advantage of the data-management functions in this package is that these functions can be piped with the result of one function being sent into the next one. We will see this in action below.

For this section, I follow the video "Data wrangling with R in 27 minutes" by Andrew Gard, a professor of mathematics and computer science at Lake Forest College, in Lake Forest IL, USA (https://youtu.be/oXImkptBpqc?si=7tk9yqxUoI9I-ooX).

Before we continue, let us first load the package "tidyverse" (this produces some output which I suppress here):

```
library(tidyverse)
```

When you load this package, you are actually loading a set of sub-packages. The functions that we will use in this section are all in the package "dplyr". Instead of "tidyverse" you could actually just load this package and still get the functions that I will discuss here.

Another important sub-package of "tidyverse" is "ggplot2", an innovative package for plotting variables.

4.1 Selecting rows

The dataframe "EU_elections" that we loaded before, contains information about all communes in Austria. In a first step, suppose, we are only interested in the results for Lower Austria and want to extract those from the data frame.

When we inspect the data we see that for all communes in Lower Austria the first variable (GKZ) starts with "G3". So, we want to select those rows where "GKZ" starts with "G3". We do this with the function `filter()`. We can use this function in the standard form by giving it the input data frame and a condition for the filtering. For the condition, we use the function `str_starts()` and pass it the name of the variable to investigate and the string that we want the variable to start with. The following function returns "TRUE" when the variable "GKZ" starts with "G3" and FALSE otherwise.

```
str_starts(GKZ, "G3")
```

So, our filter is as follows. We put the result into the data frame "tst".

```
tst <- filter(EU_elections, str_starts(GKZ, "G3"))
```

Load the result into the viewer with `View(tst)`. You should only see data for Lower Austria.

Using piping, we can achieve exactly the same result with

```
tst <- EU_elections %>%
  filter(str_starts(GKZ, "G3"))
```

Here, we first determine the data frame and then pipe that to `filter()` with the piping command `%>%`. In RStudio, the keyboard shortcut for the piping command is "Ctrl-Shift-M". Note that we omit the name of the data frame in the `filter()` command.

In this simple example, the benefits of piping are not so obvious. They may become clearer when we want to use an additional filter. Say, we want to use only those observations from Lower Austria where the number of voters is larger than 10,000. In the standard notation, we can achieve this with a second condition in the `filter()` command:

```
tst <- filter(EU_elections, str_starts(GKZ, "G3"), Voters > 10000)
```

With piping, we may use the result of the first `filter()`command and pipe it into a second one.

```
tst <- EU_elections %>%
  filter(str_starts(GKZ, "G3")) %>%
  filter(Voters > 10000)
```

When we look at the result, we see that our filter also returns aggregates (districts, regions, the whole state). To eliminate those, we pipe the result through another filter. Now, we check whether "GKZ" ends with "00". This characterizes the aggregates. Since we want to *eliminate* those, we have to negate the result of `str_ends` with an exclamation mark ("!").

```
tst <- EU_elections %>%
  filter(str_starts(GKZ, "G3")) %>%
  filter(Voters > 10000) %>%
  filter(!str_ends(GKZ, "00"))
```

This produces the expected output and also demonstrates the value of piping. The R code is much better readable than when it is written in the standard notation.

4.2 Selecting columns

To select columns from the data frame, we use the command `select()`. To illustrate this command, say, we want to extract only the columns "GKZ", "Name", "Voters", "Votes" and "valid". We do this in the following way

```
tst <- select(EU_elections, "GKZ", "Name", "Voters", "Votes", "valid")
```

Alternatively, we can also identify the columns by number. Above, we selected columns 1 to 3, 5, and 7. Therefore, the above statement is equivalent to

```
tst <- select(EU_elections, 1:3, 5, 7)
```

View the results of both versions to verify that they are identical and only contain these columns. Try to generate a data frame with only these columns and only communes in Lower Austria with more than 10,000 voters.

There are a number of helper functions available that allow us to identify columns by name. Suppose we want to exclude all the columns with "percent" in the name (all the percent variables). We could use the following command (it selects all columns whose name does not contain "percent").

```
tst <- select(EU_elections, !contains("percent"))
```

Note the exclamation mark in front of `contains()`. We could also have used a minus sign and have said 'remove all columns that contain "percent" '.

4.3 Arranging rows

Sometimes you need the observatons in a data frame in a certain order. Observations can be ordered with the `arrange()` command. All this command needs is the name of the variable by which to order. Suppose we want to see the data arranged by "GRÜNE_percent". We use this command:

```
tst <- arrange(EU_elections, GRÜNE_percent)
```

In the results we see immediately that Tschanigraben in Burgenland has the lowest percentage (actually, the first eight communes all have no votes for this party). To find who has the highest percentage (the 7th district in Vienna), we have to scroll all the way to the bottom. If we want to arrange the observations in descending order of this variable, we just use a minus sign.

```
tst <- arrange(EU_elections, -GRÜNE_percent)
```

The function `arrange()` can handle any number of variables and uses them from left to right to sort the observations. Say, we would like to order the observations for every State of Austria descending by the percentage of the party "GRÜNE". We could do this with the following command:

```
tst <- arrange(EU_elections, substr(GKZ, 1, 2), -GRÜNE_percent)
```

We use the function `substr(GKZ, 1, 2)` to extract the first two characters from GKZ. They identify the "Bundesland". Since this is the first parameter after the name of the data frame, we first order the observations by Bundesland and then within every Bundesland by "GRÜNE_percent"; in descending order because of the minus sign.

4.4 Creating new variables

With the command `mutate()` you can create new variables in the data frame. Suppose we want to see how the two parties that are currently in a coalition at the national level, ÖVP and GRÜNE, did in the EU elections 2014. So, we want to create a new variable, "COAL_percent" which is the sum of "ÖVP_percent" and "GRÜNE_percent". To limit the output, we also reduce the number of columns using the previously discussed instruments. Since there is no need for saving the converted data frame, we pipe it to `glimpse()` at the end.

```
EU_elections %>%
  select(1:2, contains("percent")) %>%
  mutate(COAL_percent = ÖVP_percent + GRÜNE_percent) %>%
  arrange(-COAL_percent) %>%
  glimpse()
```

```
Rows: 2,707
Columns: 12
$ GKZ            <chr> "G70815", "G70837", "G70929", "G70610", "G70812", "G708~
$ Name           <chr> "Hinterhornbach", "Zöblen", "Steinberg am Rofan", "Kaun~
$ ÖVP_percent    <dbl> 0.9047619, 0.7777778, 0.8314607, 0.8152174, 0.8095238, ~
$ SPÖ_percent    <dbl> 0.02380952, 0.01851852, 0.05617978, 0.06521739, 0.04761~
$ FPÖ_percent    <dbl> 0.02380952, 0.05555556, 0.02247191, 0.03260870, 0.04761~
$ GRÜNE_percent  <dbl> 0.00000000, 0.11111111, 0.04494382, 0.05434783, 0.04761~
$ BZÖ_percent    <dbl> 0.000000000, 0.000000000, 0.000000000, 0.000000000, 0.0~
$ NEOS_percent   <dbl> 0.04761905, 0.03703704, 0.01123596, 0.02173913, 0.00000~
$ RECOS_percent  <dbl> 0.000000000, 0.000000000, 0.011235955, 0.010869565, 0.0~
$ ANDERS_percent <dbl> 0.000000000, 0.000000000, 0.011235955, 0.000000000, 0.0~
$ EUSTOP_percent <dbl> 0.000000000, 0.000000000, 0.011235955, 0.000000000, 0.0~
$ COAL_percent   <dbl> 0.9047619, 0.8888889, 0.8764045, 0.8695652, 0.8571429, ~
```

5 Some data visualizations

This section is only intended for demonstration purposes. Some of the functions and data requirements are demanding and therefore too time consuming in this general workshop. Graphing and mapping with R would be a good topic for a more specialized workshop.

5.1 Preparing data

We want to look at the EU election data by state ("Bundesland") and by district ("Bezirk"). First, we extract the respective rows from the data-frame. By inspecting the data-frame, we see that the GKZ-column of all state rows ends with "0000". So, we can filter by this condition.

```
bl <- EU_elections %>%
  filter(str_ends(GKZ, "0000")) %>%
  filter(!str_ends(GKZ, "00000"))
```

For the districts, we need to filter all rows that end on "00", but, eliminate those that end with four zeros (Austria and state totals), ald those where the district id is not numeric. for the last criterion we first extract the district id fromthe variable `GKZ` with `str_sub()`. It is between position 2 and 4. Then, we convert the result to a number with `as.numeric()`. When this is not a number, the result is `NA`. Therefore, in the final step, with `!is.na()` we select only those rows where the result is not `NA`.

In the last line, we generate a variable `g_id` with the (numeric) district id. Note that this variable has the same name as the district id in `bez`.

```
bez <- EU_elections %>%
  filter(!is.na(as.numeric(str_sub(GKZ, 2,4))))  %>%
  filter(!str_ends(GKZ, "0000")) %>%
  filter(str_ends(GKZ, "00")) %>%
  mutate(g_id = str_sub(GKZ, 2, 4))
```

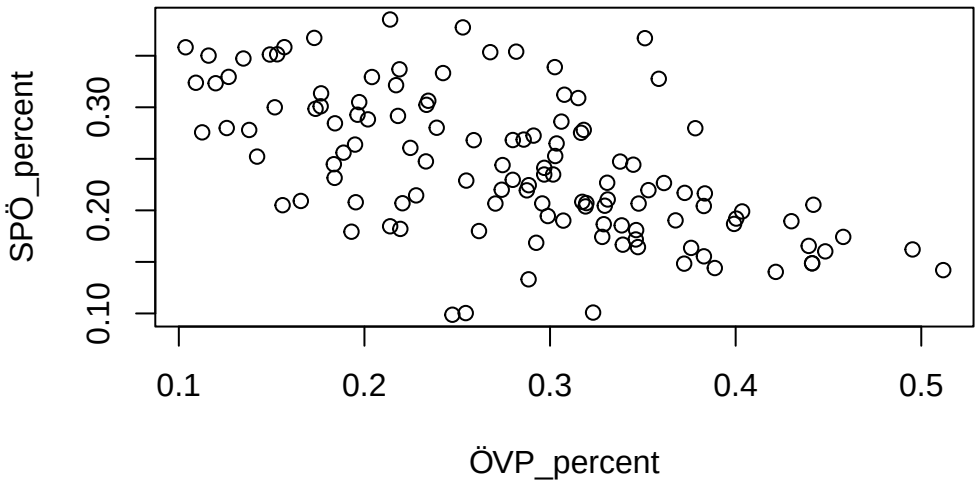
```
Warning: There was 1 warning in `filter()`.
i In argument: `!is.na(as.numeric(str_sub(GKZ, 2, 4)))`.
Caused by warning:
! NAs introduced by coercion
```

5.2 The simple `plot` function

Although `ggplot2` has become the standard tool for graphing in R, the simple `plot` function is still useful for quick views of the data.

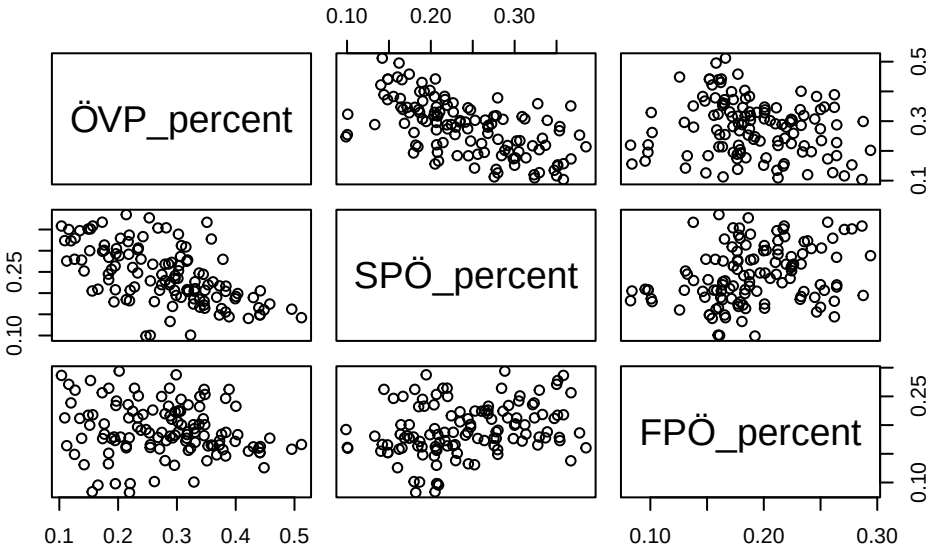
First, we want to plot the variables "ÖVP_percent" and "SPÖ_percent". We select these two variables from the dataframe and pipe them to the `plot` function:

```
bez %>%
  select("ÖVP_percent", "SPÖ_percent") %>%
  plot()
```



Then, we are asked to add the results for FPÖ to our output. When we just add "FPÖ_percent" to our `select` statement, this is what we get:

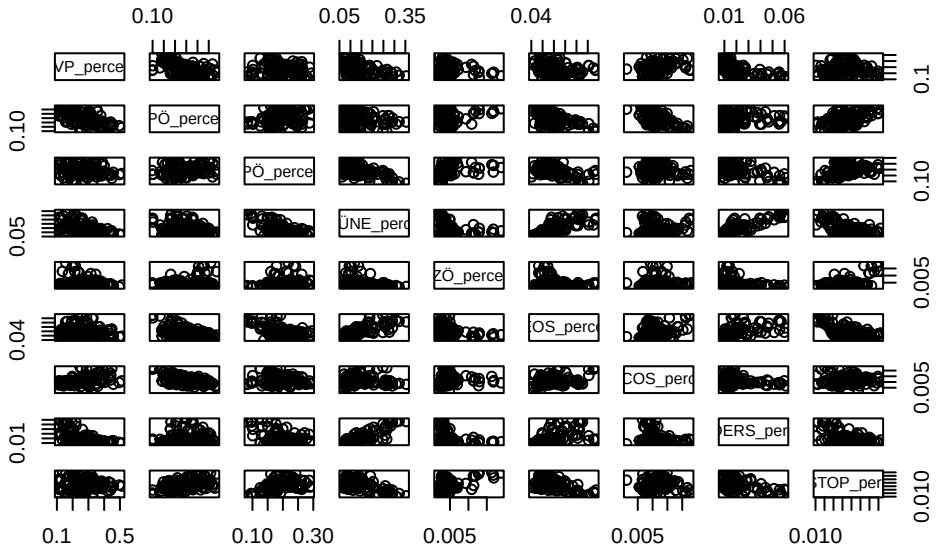
```
bez %>%
  select("ÖVP_percent", "SPÖ_percent", "FPÖ_percent") %>%
  plot()
```



The result is a **matrix of scatterplots** for all the pairs of variables. The corresponding plots above and below the main diagonal are identical, just flipped.

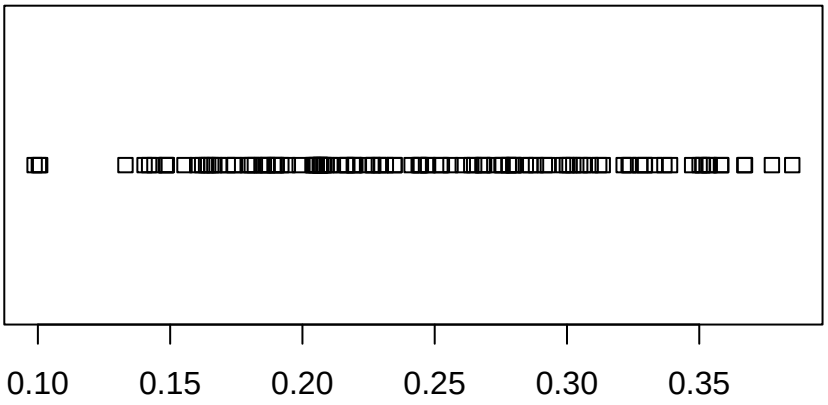
We may then want to see all parties in one matrix. For that, we select the variables that contain "percent" (as above)

```
bez %>%
  select(contains("percent")) %>%
  plot()
```



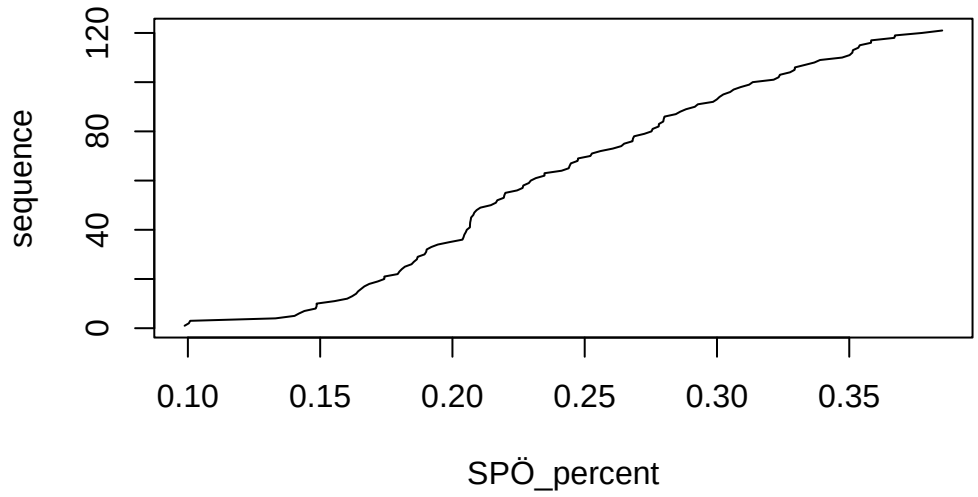
For the plot function, we always need at least two variables. When we provide just one, the output is not what we expect.

```
bez %>%
  select("SPÖ_percent") %>%
  plot()
```



We need to add a sequence and probably want to plot the data by size. Below, we sort the data with `arrange` and then add a variable named "sequence" with values running from 1 to the number of rows in the dataframe (`n()`). We change the plot type to "l" for lines. If we want to see the observations as well, we can use "b" ("both") instead.

```
bez %>%
  select("SPÖ_percent") %>%
  arrange(SPÖ_percent) %>%
  mutate(sequence = 1:n()) %>%
  plot(type = "l")
```



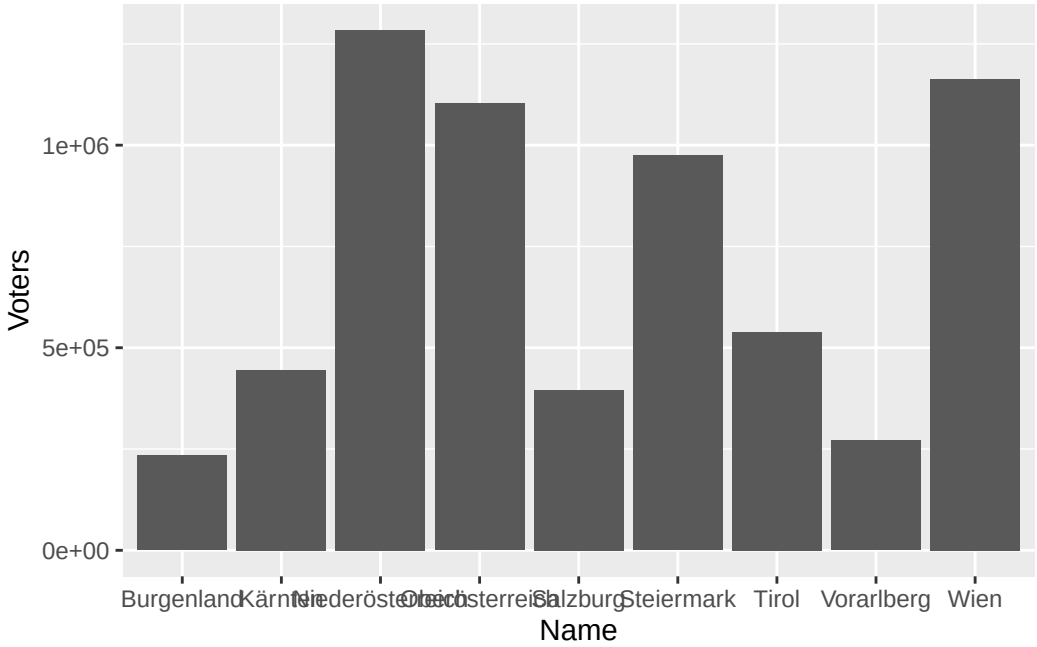
5.3 A glimpse of `ggplot2`

In recent years, `ggplot2` has become something like the standard tool for graphing in R. This package is loaded with the `tidyverse` package and it uses the "geometry of graphics" approach for the creation of graphics. The tool is very powerful (see e.g., this Video), but needs more time to explain. Therefore, I just give a glimpse of the tool.

5.3.1 A bar chart

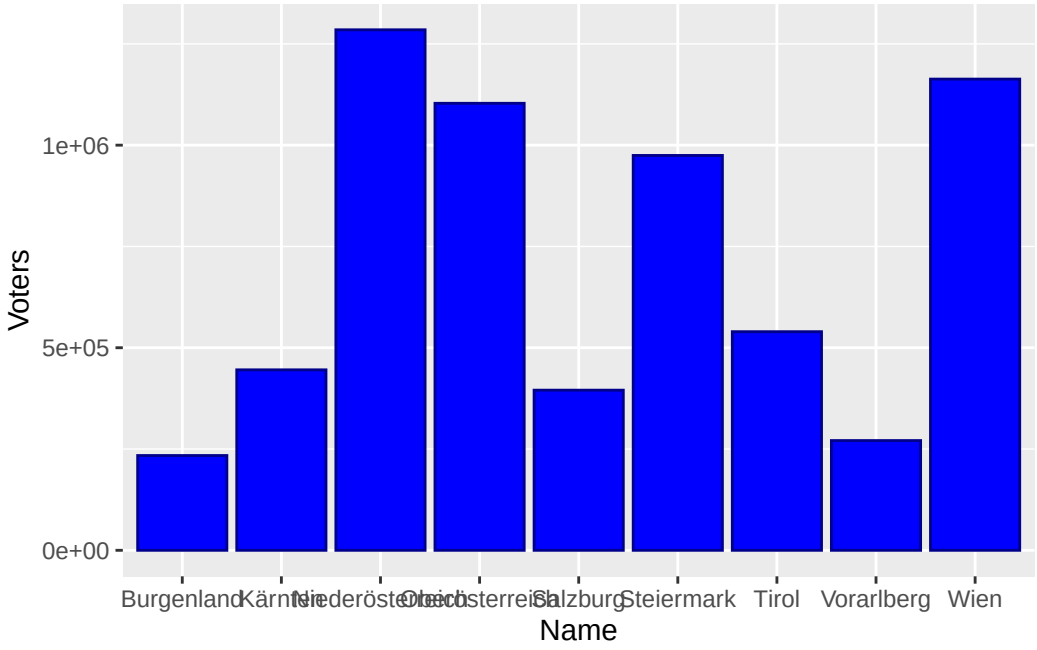
First, we want to plot the number of voters in the Bundesländer in a vertical bars graph.

```
bl %>%
  ggplot(aes(Name, Voters)) +
  geom_col()
```



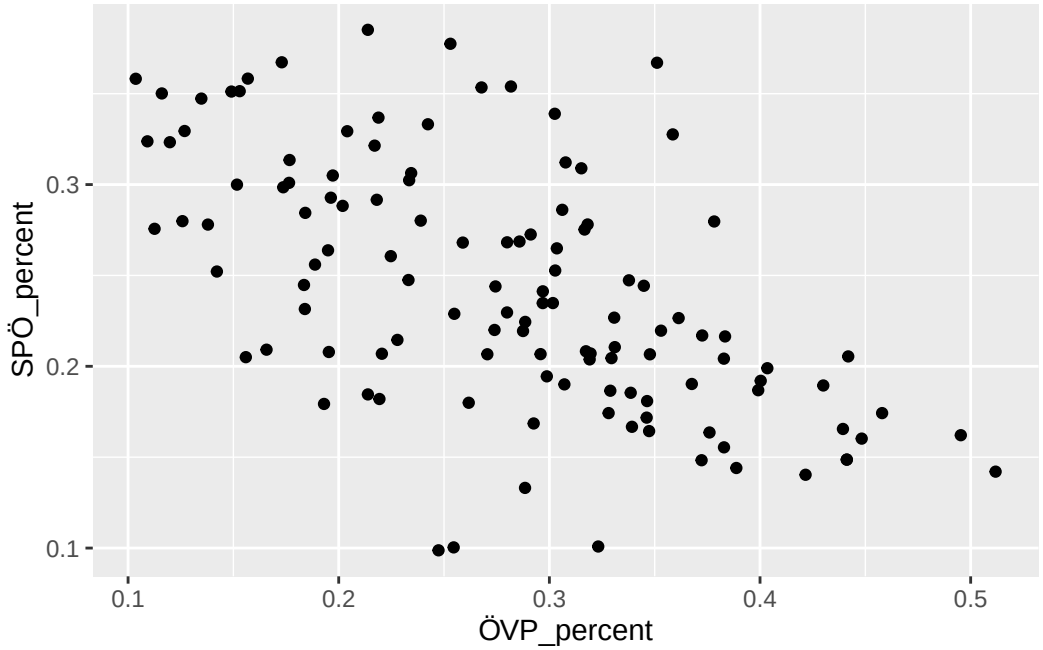
We want the bars in a different color. With `color` we can set the outline of the bars, with `fill` the color of the filling.

```
bl %>%
  ggplot(aes(Name, Voters)) +
  geom_col(color="darkblue", fill="blue")
```



5.3.2 A Scatterplot

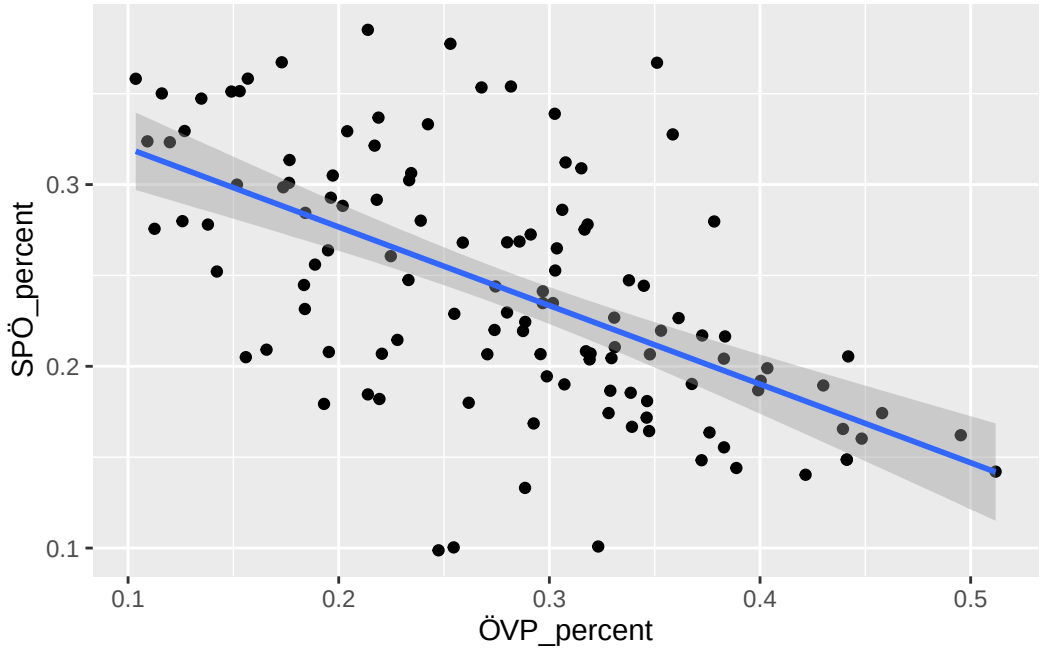
```
bez %>%
  ggplot(aes(ÖVP_percent, SPÖ_percent)) +
  geom_point()
```



We can overlay a regression line that shows the average relation between the two variables. By default, a confidence interval is added. When we specify `se = NULL`, the confidence interval is dropped.

```
bez %>%
  ggplot(aes(ÖVP_percent, SPÖ_percent)) +
  geom_point() +
  geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```



5.3.3 Maps

For a more comprehensive introduction see (Mieno 2023).

We load the package `sf`. It implements the "Simple Feature" concept in R.

We read the shapefile for the Austrian districts into `bez_map`. When we inspect it, we see that there is a column with the district ID and one called `geometry` with the outlines of each of the districts.

```
library(sf)
```

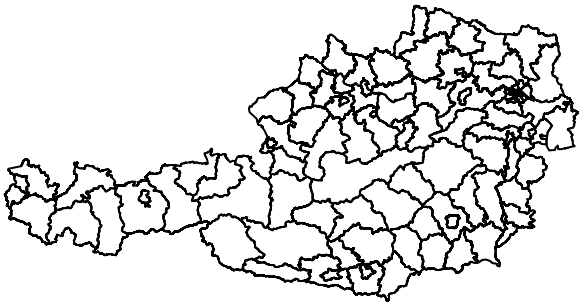
```
Linking to GEOS 3.11.2, GDAL 3.6.2, PROJ 9.2.0; sf_use_s2() is TRUE
```

```
bez_map <- st_read("OGDEXT_POLBEZ_1_STATISTIK_AUSTRIA_20230101/STATISTIK_AUSTRIA_POLBEZ_
```

```
Reading layer `STATISTIK_AUSTRIA_POLBEZ_20230101' from data source
  `C:\Users\gmaier\OneDrive - modul.ac.at\Desktop\MU-R-Introduction\OGDEXT_POLBEZ_1_STATIS
  using driver `ESRI Shapefile'
Simple feature collection with 117 features and 2 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 112518.2 ymin: 275472 xmax: 685444.5 ymax: 570431.1
Projected CRS: MGI / Austria Lambert
```

To see what we got, we select the variable `geometry` and plot it. We get the outlines of the districts.

```
bez_map  %>%
  select(geometry)  %>%
  plot()
```

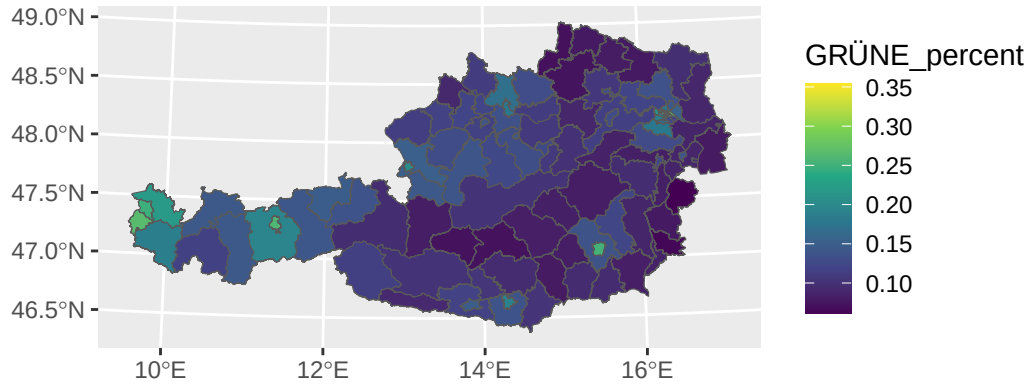


Now, we merge the data in `bez` with the geographical information in `bez_map`. Since we used the same name for the district id, the function uses these values for merging. Then, we pipe the result to the function `st_as_sf()` to convert to "Simple Format".

```
bez_full <- merge(bez_map, bez)  %>%
  st_as_sf()
```

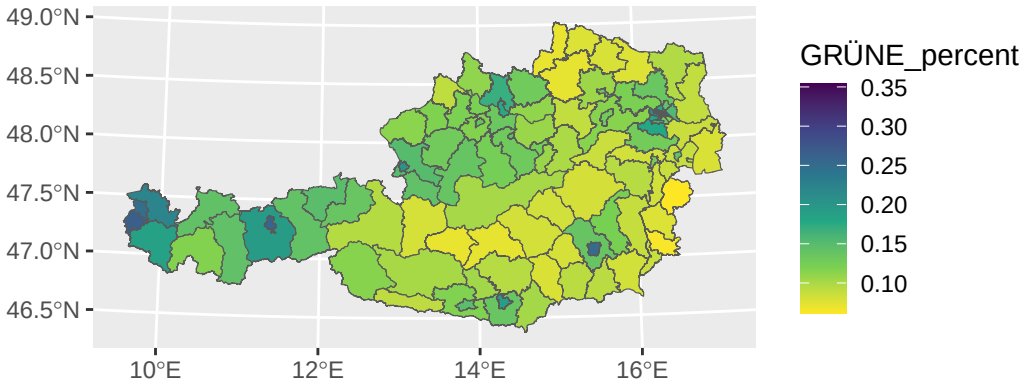
Now, we are ready to plot the map. In the `aes()` function we specify that the areas should be filled according to the values of `GRÜNE_percent`. `geom_sf()` does the mapping, `scale_fill_viridis_c()` sets the color schema.

```
ggplot(bez_full, aes(fill=GRÜNE_percent)) +
  geom_sf() +
  scale_fill_viridis_c()
```



Strangely, by default the lowest values are assigned the darkest colors. To reverse this, we set the parameter `direction` to -1.

```
ggplot(bez_full, aes(fill=GRÜNE_percent)) +
  geom_sf() +
  scale_fill_viridis_c(direction = -1)
```


49.0°N
48.5°N
48.0°N
47.5°N
47.0°N
46.5°N
10°E
12°E
14°E
16°E
GRÜNE_percent
0.35
0.30
0.25
0.20
0.15
0.10

6 Some basic statistical analyses

The EU election data is not very well suited for the types of analysis I want to discuss. Therefore, I want to use the dataset "diamonds" that we loaded with the package "tidyverse". You may want to look into the dataset with `View(diamonds)`.

6.1 Summary statistics

One of the most used functions in R is `summary()`. It is a generic function that produces a summary of the supplied object. Depending on the class of the object, the output of `summary()` looks quite different.

When we call `summary()` with a data frame, the function prints summary statistics for all the variables in the data frame. Here is the result for the data frame "diamonds".

```
summary(diamonds)
```

```
     carat               cut        color        clarity          depth      
 Min.   :0.2000   Fair      : 1610   D: 6775   SI1     :13065   Min.   :43.00  
 1st Qu.:0.4000   Good      : 4906   E: 9797   VS2     :12258   1st Qu.:61.00  
 Median :0.7000   Very Good:12082   F: 9542   SI2     : 9194   Median :61.80  
 Mean   :0.7979   Premium   :13791   G:11292   VS1     : 8171   Mean   :61.75  
 3rd Qu.:1.0400   Ideal     :21551   H: 8304   VVS2    : 5066   3rd Qu.:62.50  
 Max.   :5.0100                     I: 5422   VVS1    : 3655   Max.   :79.00  
                                    J: 2808   (Other): 2531                  
     table           price             x                y         
 Min.   :43.00   Min.   :  326   Min.   : 0.000   Min.   : 0.000  
 1st Qu.:56.00   1st Qu.:  950   1st Qu.: 4.710   1st Qu.: 4.720  
 Median :57.00   Median : 2401   Median : 5.700   Median : 5.710  
 Mean   :57.46   Mean   : 3933   Mean   : 5.731   Mean   : 5.735  
 3rd Qu.:59.00   3rd Qu.: 5324   3rd Qu.: 6.540   3rd Qu.: 6.540  
 Max.   :95.00   Max.   :18823   Max.   :10.740   Max.   :58.900  

       z         
 Min.   : 0.000  
 1st Qu.: 2.910  
 Median : 3.530  
 Mean   : 3.539  
 3rd Qu.: 4.040  
 Max.   :31.800  
```

Note the difference between "cut", "color", and "clarity" on the one hand and all the other variables on the other. The members of the second group are continuous variables. They

can take on a range of values. The numbers we find in the variable "price", for example, represent the prices of the various diamonds. "cut", "color", and "clarity", on the other hand, are categorical variables. They categorize the observations. In R such variables are usually stored as factors with labels and values. The variable "cut", for example, has labels "Fair", "Good", etc. Internally, each of these categories is stored with a specific number. All diamonds with a "Good" cut, for example, may internally have the value 2 stored in this variable.

As you can see in the result above, the output of `summary()` differs for the two types of variables. For continuous variables, the function shows the lowest and highest values, the mean, the median, and the first and third quartile. For categorial variables where those indicators make no sense, the function lists the categories and how many observations fall into each of them (absolute frequencies).

6.2 Grouped summaries

We would like to get some of the statistics that we saw above for the whole dataset broken down by categories. The package "tidyverse" offers tools for that. With the function `group_by()` we can define categories in our data frame. The function `summarize()` (different from `summary()` above) then applies some analysis to each group separately.

Suppose we want to get the mean and the standard deviation of the price for each of the "cut" categories. The following set of commands does exactly that. You can use any number of parameters for `summarize()`.

```
diamonds  %>%
  group_by(cut)   %>%
  summarize(mean(price), sd(price))
```

```
# A tibble: 5 x 3
  cut       `mean(price)` `sd(price)`
  <ord>             <dbl>       <dbl>
1 Fair              4359.       3560.
2 Good              3929.       3682.
3 Very Good         3982.       3936.
4 Premium           4584.       4349.
5 Ideal             3458.       3808.
```

We can define more appropriate headings for the columns:

```
diamonds %>%
  group_by(cut) %>%
  summarize(av_price = mean(price), sd_price = sd(price))
```

```
# A tibble: 5 x 3
  cut       av_price sd_price
  <ord>        <dbl>    <dbl>
1 Fair         4359.    3560.
2 Good         3929.    3682.
3 Very Good    3982.    3936.
4 Premium      4584.    4349.
5 Ideal        3458.    3808.
```

The special function `n()` gives the number of observations in each group.

```
diamonds %>%
  group_by(cut) %>%
  summarize(av_price = mean(price), sd_price = sd(price), count = n())
```

```
# A tibble: 5 x 4
  cut       av_price sd_price count
  <ord>        <dbl>    <dbl> <int>
1 Fair         4359.    3560.  1610
2 Good         3929.    3682.  4906
3 Very Good    3982.    3936. 12082
4 Premium      4584.    4349. 13791
5 Ideal        3458.    3808. 21551
```

When we use more than one categorical variable in the `group_by()` command, we get more and smaller groups. The grouping is first done for the first variable, then within each group for the second variable, and so on.

```
diamonds %>%
  group_by(cut, color) %>%
  summarize(av_price = mean(price), sd_price = sd(price), count = n())
```

```
`summarise()` has grouped output by 'cut'. You can override using the `.groups`
argument.
```

```
# A tibble: 35 x 5
# Groups:   cut [5]
   cut   color av_price sd_price count
   <ord> <ord>    <dbl>    <dbl> <int>
 1 Fair  D        4291.    3286.   163
 2 Fair  E        3682.    2977.   224
 3 Fair  F        3827.    3223.   312
 4 Fair  G        4239.    3610.   314
```

```
 5 Fair  H        5136.    3886.   303
 6 Fair  I        4685.    3730.   175
 7 Fair  J        4976.    4050.   119
 8 Good  D        3405.    3175.   662
 9 Good  E        3424.    3331.   933
10 Good  F        3496.    3202.   909
# i 25 more rows
```

6.3 Crosstabulation

We want to know whether the categories "cut" and "color" are related. Let us first create a two-way table.

```
table(diamonds$cut, diamonds$color)
```

```
             D    E    F    G    H    I    J
  Fair       163  224  312  314  303  175  119
  Good       662  933  909  871  702  522  307
  Very Good 1513 2400 2164 2299 1824 1204  678
  Premium   1603 2337 2331 2924 2360 1428  808
  Ideal     2834 3903 3826 4884 3115 2093  896
```

Since the command `table()` does not expect the data frame as the first parameter, we cannot use a pipe. Instead, we need to inform the function that our variables are in "diamonds". This is done with `dataframe$variable`.

Alternatively, we can `attach()` the data frame "diamonds". When we do this, R will first look for variables in this dataframe. To undo this connection, use the command `detach()`.

```
attach(diamonds)
table(cut, color)
```

```
           color
cut            D    E    F    G    H    I    J
  Fair       163  224  312  314  303  175  119
  Good       662  933  909  871  702  522  307
  Very Good 1513 2400 2164 2299 1824 1204  678
  Premium   1603 2337 2331 2924 2360 1428  808
  Ideal     2834 3903 3826 4884 3115 2093  896
```

Instead of just printing the table, we should store it in an object and then work with this object. When we print the object, we get the same output as above.

```
mytable <- table(cut, color)
mytable
```

```
           color
cut            D    E    F    G    H    I    J
  Fair       163  224  312  314  303  175  119
  Good       662  933  909  871  702  522  307
  Very Good 1513 2400 2164 2299 1824 1204  678
  Premium   1603 2337 2331 2924 2360 1428  808
  Ideal     2834 3903 3826 4884 3115 2093  896
```

With `summary()`, we get meta information about the table and the results of a Chi-square test.

```
mytable <- table(cut, color)
summary(mytable)
```

```
Number of cases in table: 53940
Number of factors: 2
Test for independence of all factors:
    Chisq = 310.32, df = 24, p-value = 1.395e-51
```

Instead of the absolute frequencies, we may want to see relative frequencies. Also, we can request the Chi-square test explicitly from the table object.

```
mytable <- table(diamonds$cut, diamonds$color)
prop.table(mytable)
```

```
                     D           E           F           G           H
  Fair      0.003021876 0.004152762 0.005784205 0.005821283 0.005617353
  Good      0.012272896 0.017296997 0.016852058 0.016147571 0.013014461
  Very Good 0.028049685 0.044493882 0.040118650 0.042621431 0.033815350
  Premium   0.029718205 0.043325918 0.043214683 0.054208380 0.043752317
  Ideal     0.052539859 0.072358176 0.070930664 0.090545050 0.057749351

                     I           J
  Fair      0.003244346 0.002206155
  Good      0.009677419 0.005691509
  Very Good 0.022321098 0.012569522
  Premium   0.026473860 0.014979607
  Ideal     0.038802373 0.016611049
```

```
chisq.test(mytable)
```

```
    Pearson's Chi-squared test

data:  mytable
X-squared = 310.32, df = 24, p-value < 2.2e-16
```

To reduce the number of digits in the table, wrap it in the `round()` function and supply the number of digits you want.

```
mytable <- table(diamonds$cut, diamonds$color)
round(prop.table(mytable),4)
```

```
               D      E      F      G      H      I      J
  Fair      0.0030 0.0042 0.0058 0.0058 0.0056 0.0032 0.0022
  Good      0.0123 0.0173 0.0169 0.0161 0.0130 0.0097 0.0057
  Very Good 0.0280 0.0445 0.0401 0.0426 0.0338 0.0223 0.0126
  Premium   0.0297 0.0433 0.0432 0.0542 0.0438 0.0265 0.0150
  Ideal     0.0525 0.0724 0.0709 0.0905 0.0577 0.0388 0.0166
```

6.4 Analysis of Variance

We want to check whether or not the three categorical variables significantly influence the price of diamonds. The appropriate method is Analysis of Variance, called by `aov()` in R.

For the analysis of variance, we have to specify a model formula. This is the same as in regression analysis. We use a very simple version of the model formula. One can specify very different and quite complex models with this tool.

Our model formula says: use "price" as the dependent variable and "cut", "color" and "clarity" as explanatory variables. Note the "~" between "price" and the other variables and that the explanatory variables are combined with "+".

```
aov(price ~ cut + color + clarity)
```

```
Call:
   aov(formula = price ~ cut + color + clarity)

Terms:
                        cut        color      clarity    Residuals
Sum of Squares   11041745359  25507044246  19997761503 801926584410
Deg. of Freedom            4            6            7        53922
```

```
Residual standard error: 3856.42
Estimated effects may be unbalanced
```

When we use the `aov()` command, the output is limited. We can get more detailed information with `summary()`. We store the result of `aov()` in an object and supply that to `summary()`.

```
anova <- aov(price ~ cut + color + clarity)
summary(anova)
```

```
               Df    Sum Sq   Mean Sq F value Pr(>F)
cut             4 1.104e+10 2.760e+09   185.6 <2e-16 ***
color           6 2.551e+10 4.251e+09   285.9 <2e-16 ***
clarity         7 2.000e+10 2.857e+09   192.1 <2e-16 ***
Residuals   53922 8.019e+11 1.487e+07
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We see that all three categorical variables significantly influence the price.

6.5 Regression Analysis

We want to know how carat and the categories influence the price of diamonds. We run a linear regression with the command `lm()`. We expand the above model formula with "carat".

```
lm(price ~ carat + cut + color + clarity)
```

```
Call:
lm(formula = price ~ carat + cut + color + clarity)

Coefficients:
(Intercept)        carat        cut.L        cut.Q        cut.C        cut^4
  -3710.603     8886.129      698.907     -327.686      180.565       -1.207
    color.L      color.Q      color.C      color^4      color^5      color^6
  -1910.288     -627.954     -171.960       21.678      -85.943      -49.986
  clarity.L    clarity.Q    clarity.C    clarity^4    clarity^5    clarity^6
   4217.535    -1832.406      923.273     -361.995      216.616        2.105
  clarity^7
    110.340
```

When we run this, we get again very limited output. We get more information with the `summary()` function (we store the result of the regression analysis in the object "reg" and supply that to `summary()`).

```
reg <- lm(price ~ carat + cut + color + clarity)
summary(reg)
```

```
Call:
lm(formula = price ~ carat + cut + color + clarity)

Residuals:
     Min       1Q   Median       3Q      Max
-16813.5   -680.4   -197.6    466.4  10394.9

Coefficients:
             Estimate Std. Error  t value Pr(>|t|)
(Intercept) -3710.603     13.980 -265.414  < 2e-16 ***
carat        8886.129     12.034  738.437  < 2e-16 ***
cut.L         698.907     20.335   34.369  < 2e-16 ***
cut.Q        -327.686     17.911  -18.295  < 2e-16 ***
cut.C         180.565     15.557   11.607  < 2e-16 ***
cut^4          -1.207     12.458   -0.097    0.923
color.L     -1910.288     17.712 -107.853  < 2e-16 ***
color.Q      -627.954     16.121  -38.952  < 2e-16 ***
color.C      -171.960     15.070  -11.410  < 2e-16 ***
color^4        21.678     13.840    1.566    0.117
color^5       -85.943     13.076   -6.572 5.00e-11 ***
color^6       -49.986     11.889   -4.205 2.62e-05 ***
clarity.L    4217.535     30.831  136.794  < 2e-16 ***
clarity.Q   -1832.406     28.827  -63.565  < 2e-16 ***
clarity.C     923.273     24.679   37.411  < 2e-16 ***
clarity^4    -361.995     19.739  -18.339  < 2e-16 ***
clarity^5     216.616     16.109   13.447  < 2e-16 ***
clarity^6       2.105     14.037    0.150    0.881
clarity^7     110.340     12.383    8.910  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1157 on 53921 degrees of freedom
Multiple R-squared:  0.9159,    Adjusted R-squared:  0.9159
F-statistic: 3.264e+04 on 18 and 53921 DF,  p-value: < 2.2e-16
```

For every categorical variable there is one coefficient less estimated than the number of categories. I do not know (and could not find out) why the names of the categories look like

that. In a real application, I would need to do that. Since we do not interpret the results, it does not really matter here.

Anyways, we see that most regression coefficients are highly significant. From the t-value and significance indicator of "carat" we see that this variable significantly contributes to explaining the price.

6.5.1 Is the contribution of a categorical variable significant?

For each of the categorical variables we get a number of regression coefficients. Most of them are significant, but not all. To check whether or not a categorical variable, say "color", as a whole is significant, we can run a likelihood ratio test. For that, we first run a regression with this variable dropped from the model formula

```
reg.2 <- lm(price ~ carat + cut + clarity)
summary(reg.2)
```

```
Call:
lm(formula = price ~ carat + cut + clarity)

Residuals:
     Min       1Q   Median       3Q      Max
-16842.5   -636.4   -114.3    474.8  11238.6

Coefficients:
             Estimate Std. Error  t value Pr(>|t|)
(Intercept) -3187.540     14.475 -220.208   <2e-16 ***
carat        8472.026     12.615  671.584   <2e-16 ***
cut.L         713.804     22.511   31.709   <2e-16 ***
cut.Q        -334.503     19.828  -16.871   <2e-16 ***
cut.C         188.482     17.218   10.947   <2e-16 ***
cut^4           1.663     13.794    0.121   0.9040
clarity.L    4011.681     33.931  118.231   <2e-16 ***
clarity.Q   -1821.922     31.870  -57.167   <2e-16 ***
clarity.C     917.658     27.313   33.598   <2e-16 ***
clarity^4    -430.047     21.831  -19.699   <2e-16 ***
clarity^5     257.141     17.821   14.429   <2e-16 ***
clarity^6      26.909     15.539    1.732   0.0833 .
clarity^7     186.742     13.685   13.646   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1281 on 53927 degrees of freedom
Multiple R-squared:  0.8969,    Adjusted R-squared:  0.8969
F-statistic: 3.911e+04 on 12 and 53927 DF,  p-value: < 2.2e-16
```

Now, we have two objects with regression results stored; "reg" and "reg.2". The package "lmtest" provides a function for likelihood ratio tests (`lrtest()`).

```
library(lmtest)
```

```
Loading required package: zoo
```

```
Attaching package: 'zoo'
```

```
The following objects are masked from 'package:base':

    as.Date, as.Date.numeric
```

```
lrtest(reg.2, reg)
```

```
Likelihood ratio test

Model 1: price ~ carat + cut + clarity
Model 2: price ~ carat + cut + color + clarity
  #Df  LogLik Df Chisq Pr(>Chisq)
1  14 -462490
2  20 -456992  6 10998  < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The result shows that the six color variables together have a significant impact on the price.

6.5.2 Checking the residuals

An important aspect of regression analysis is to check the residuals of the "reg" model. The residuals are included in the "reg" object and can be accessed via `reg$residuals`. First, we calculate the mean and the standard deviation of the residuals, the minimum and the maximum.

```
(reg.mean <- mean(reg$residuals))
```

```
[1] 4.1985e-13
```

```
(reg.sd <- sd(reg$residuals))
```

```
[1] 1156.659
```

```
min(reg$residuals)
```

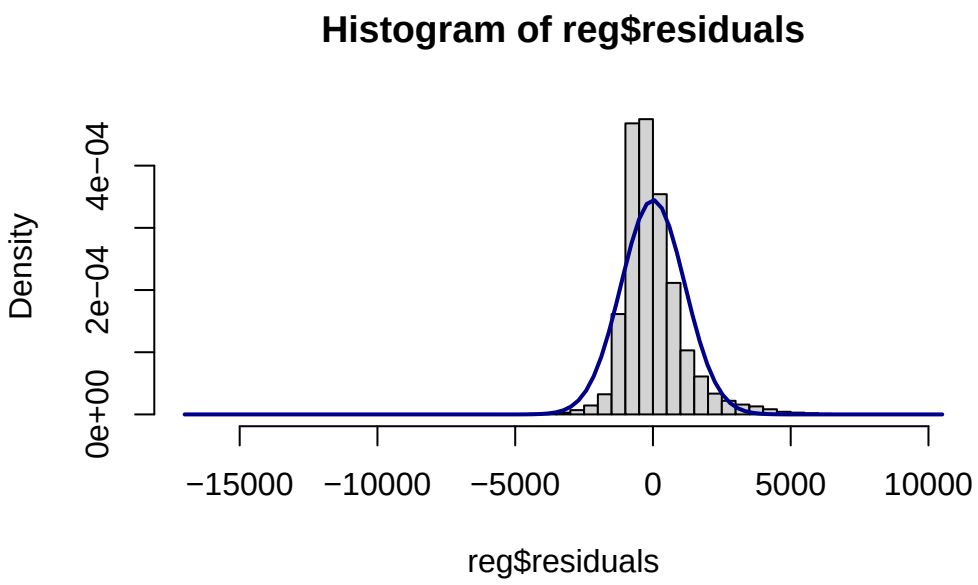
```
[1] -16813.48
```

```
max(reg$residuals)
```

```
[1] 10394.86
```

In a second step, we plot a histogram of the residuals and overlay it with a normal density function with the mean and standard deviation that we calculated above.

```
hist(reg$residuals, breaks=50, freq=FALSE)
curve(dnorm(x, mean=reg.mean, sd=reg.sd),
      col="darkblue", lwd=2, add=TRUE)
```



7 Writing up and publishing your analysis

When we do quantitative analysis, one of the problems is to keep track of our workflow. What dataset did we use for what step? After all the different trials, what is our final model version? Did we exclude observations from the analysis and which ones? Such questions can be particularly troubling when we work with one of those easy to use menu driven statistics programs.

At a more general level, these questions raise issues of reproducibility and replicability that are discussed in the scientific literature.

R and RStudio offer various tools for keeping track of your workflow. I cannot go into much detail here. I will just sketch the main issues.

7.1 Saving the history

A very valuable tool in this context is the "history" that R collects. Whenever you submit a command to R via the console, R adds this command to the bottom of the history. To inspect the history, go to the History pane in the Environment window.

With the "disk" icon in the tools bar of the History pane you can save the whole history to a file. You can do the same also from the console with the command `savehistory()`. The saved history is plain text. So you can give it the extension ".txt" or ".R", if you want to develop an R-script from it.

7.2 R scripts

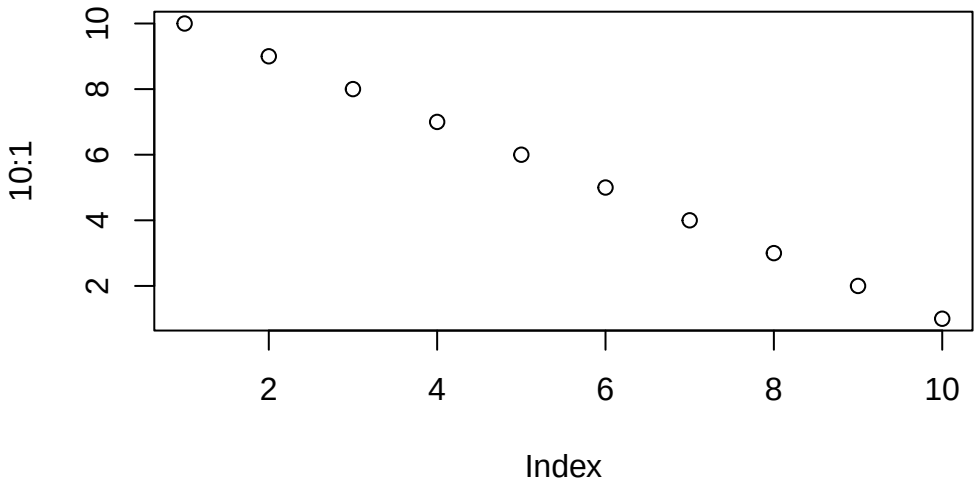
As mentioned before, you can collect R commands in R scripts either manually or by saving the history. R scripts have the file extension ".R". You can run the script (i.e., all the included commands in sequence) from the command line of your operating system with the program "Rscript". Alternatively, you can open the script in RStudio (File - Open File) and then execute it via the Source button. That is equivalent to typing `source("<path to script>")`in the console. For example, the following line runs the script "myscript.R" and echoes the commands in the script and their output:

```
source("myscript.R", echo=TRUE)
```

```
> 5 + 7
[1] 12

> 5 + 9
[1] 14

> plot(10:1)
```



A third option for running a script is the function `Rscript()` in the package "xfun".

When you just collect the R commands in your script, as you get when you save the history, you will not achieve much improvement in terms of reproducibility and replicability. Without any further information, it will be difficult to understand your workflow after a few months. Comments can resolve some of these issues. I suggest that you structure your scripts with empty lines and comments that explain what is the purpose of the following set of commands. In an R script, everything following a `#` is a comment. You can add lines or blocks of lines of comments and append comments to individual commands.

7.3 RMarkdown and Quarto

When you properly comment R scripts, you may reach a point where the comments become something like the backbone of a report, an article, or even a book. Instead of adding text in the form of comments to a set of R commands, you may want to add R commands to some text that you prepare. It would be great if you could even run the R commands from your text and embed their results into your text.

This is exactly what you can achieve with RMarkdown and Quarto. Quarto is a more comprehensive version of RMarkdown. For our purpose, however, we do not have to worry about their differences. We will just talk about Quarto.

According to the Quarto homepage, Quarto is

> An open-source scientific and technical publishing system

Quarto (Tierney 2022) is a standard that allows you to combine formatted text with R code. Text formatting uses a markdown dialect. The syntac is quite simple. A line starting with one hash (`#`) followed by a blank, for example, creates a top level heading. Two hashes create a second level header, three a third level, and so on. One or more blank lines separate the paragraphs. Text enclosed in single asteriscs (`*text*`) is set in italics, double asteriscs (`**text**`) make the text bold. The markdown dialect of Quarto lets you include footnotes, figures, hyperlinks, references, and mathematical expressions using LaTeX syntax.

Here are a few examples using LaTeX mathematics:

- pure LaTeX (equation environment)

$$E = mc^2 \tag{1}$$

- pure LaTeX style (eqnarray environment)

$$\begin{aligned} Y &\sim X\beta_0 + X\beta_1 + \epsilon \\ \epsilon &\sim N(0, \sigma^2) \end{aligned} \tag{2}$$

- Quarto style (double dollar signs)

$$Y \sim X\beta_0 + X\beta_1 + \epsilon$$

$$\epsilon \sim N(0, \sigma^2)$$

- Mathematics in the text: In the statement above, Y is the dependent variable, X the independent variable. β_0 and β_1 are coefficients that we want to estimate. All the random influence is symbolized by ϵ.

7.3.1 Code chunks and inline code

A unique feature of Quarto and RMarkdown are "code chunks". Code chunks are snippets of R code embedded in your document. Usually, this code is executed by R and the result is inserted into the document underneath the code chunk.

Code chunks start with ```` ```{r} ```` and end with ```` ``` ````. Everything between those markers is considered R code or a code chunk option. With code chunk options you can determine, for example, whether the R code should be shown and whether it should be executed. Quarto defines a large number of code chunk options.

You can also embed code directly into your text. The syntax for that is the following:

```
`r <code>`
```

This is most useful to extract specific results from the analysis. For example, we can use

```
`r reg$coefficients["carat"]`
```

to extract the estimate for the variable "carat" from the first regression. This could be used in the following text, for example: Between the full and the constrained regression, the coefficient of "carat" changes by -414.1 units from 8886.13 to 8472.03.

7.3.2 How to use a Quarto document

Quarto documents are identified with the file extension ".qmd". You can start, open, and edit Quarto documents in the Source window of RStudio, Whenever a Quarto document is open and active in RStudio, your tools will look somewhat different. In addition to a Source pane you will also see a Visual pane. When you switch to that pane, you will see the formatted version of your text. The visual pane is also an editor that helps you with the markdown syntax.

Irrespective whether you use the visual or the text editor, each code chunk will be shown with a light gray background and with two buttons on the top right. Clicking the first one (a downward pointing triangle with a green bar) will execute all code chunks above the current one. Clicking the second one (a green triangle pointing right) will execute the current chunk. When a code chunk generates output, it will be inserted after the code chunk.

You do not have to click through all the code chunks. The button "Run" in the toolbar offers a set of options. When you click the last one("Run All"), RStudio will go through your Quarto document and execute all code chunks in sequence. You can follow the execution in the Console. Any warnings or error messages will also be displayed there.

In the toolbar there is also a button "Render". When you click this button, RStudio will execute all the code chunks and in addition generate some output from your Quarto document. Some possible output formats are **HTML**, **PDF**, **Word**, **OpenOffice**, and **ePub**. Which format to use is specified in the so called YAML-code at the top of a Quarto document. There you can specify a set of parameters that let you generate books, articles, presentations, blog entries, web pages, and so on. For the purpose of illustration, I will first produce a HTML-page and then a PDF-file with the same content.

8 Conclusion

This presentation provides a quick overview of R, RStudio, and some of their exciting new features like RMarkdown and Quarto. None of the aspects was covered comprehensively. Practically all R commands that I have mentioned offer more parameters that influence their behavior. We could go deeper in almost every aspect. Options that come to my mind are

- statistical procedures
- R data types
- graphics with R (esp. "ggplot")
- writing R functions
- details of the markdown language
- specifics of Quarto
- challenges of reproducibility and replicability

9 References

Mieno, Taro. 2023. *R as GIS for Economists*. https://tmieno2.github.io/R-as-GIS-for-Economists/index.html.

Tierney, Nicholas. 2022. *Quarto for Scientists*. https://qmd4sci.njtierney.com/.