Modelling Criminological Data LAWS20452

Reka Solymosi and Eon Kim (based on material developed with Juanjo Medina

2022-02-15

Contents

Preface

This is the main text you will be using during the labs for the module *Modelling Criminological Data*. Every week you will have to use these materials during the lab sessions. The idea is that you read this book and try to run the code we provide in your own machine. Along the way you will see you have to complete a series of exercises to check that you are correctly understanding the materials that we introduce. We hope you find these materials useful. They are a work in process, so please if you have any suggestions please don't hesitate to get in touch via juanjo.medina@manchester.ac.uk.

Chapter 1

A first lesson about R

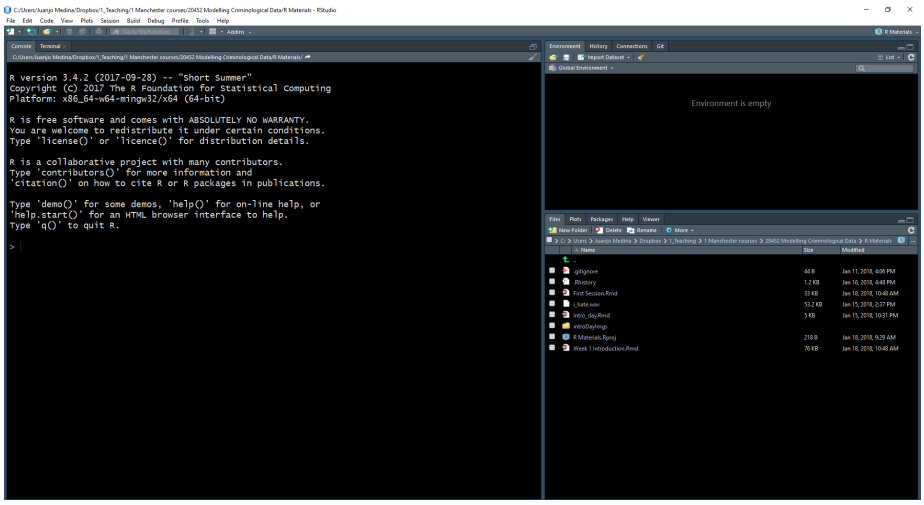
1.1 Install R & RStudio

We recommend that you use your own laptops for this course. If you have not already, then please download and install R and R Studio onto your laptops. - click here for instructions using Windows or - here for instructions using a Mac. If you are using a Mac it would be convenient that you use the most up to date version of OS or, at least one compatible with the most recent version of R. Read this if you want to check how to do that.

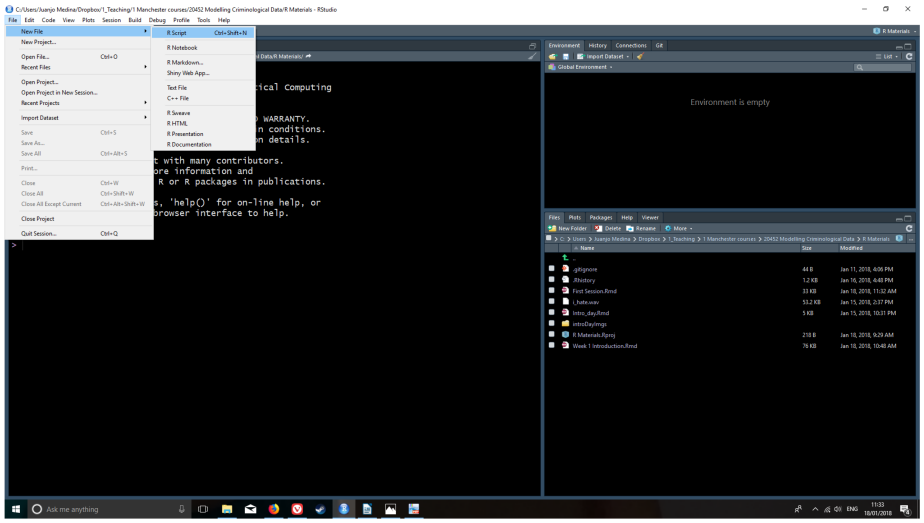
If you prefer, you can always use any of the PCs in the computer cluster. All of them already have the software installed.

1.2 Open up and explore RStudio

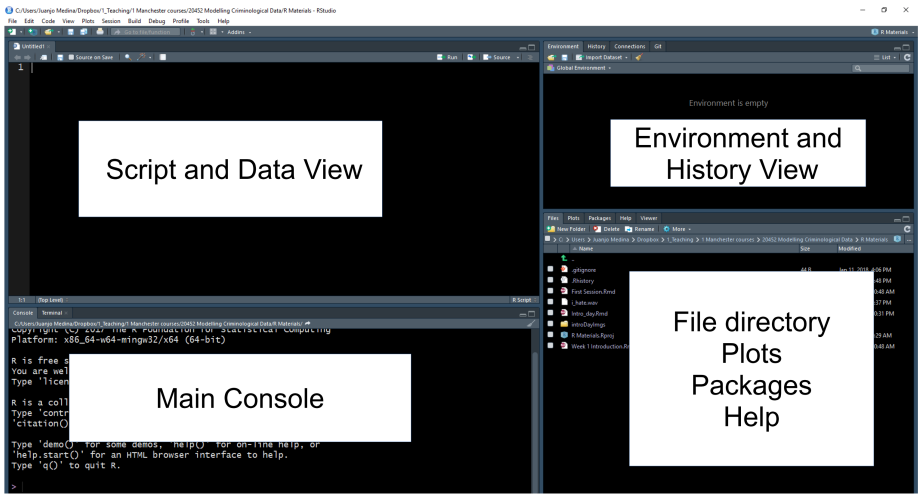
In this session we will focus in developing basic familiarity with R Studio. You can use R without using R Studio, but R Studio is an app that makes it easier to work with R. R Studio automatically runs R in the background. We will be interacting with R in this course unit via R Studio.



When you first open R Studio, you will see (as in the image above) that there are 3 main panes. The bigger one to your left is the console. If you read the text in the console you will see that R Studio is indeed opening R and you can see what version of R you are running. Depending on whether you are using the cluster machines or your own installation this may vary, but don't worry too much about it.



The view in R Studio is structured so that you have 4 open panes in a regular session. Click in the *File* drop down Menu, select *New File*, then *R Script*. You will now see the 4 window areas in display. On each of these areas you can shift between different views and panels. You can also use your mouse to re-size the different windows if that is convenient.



Look for example at the bottom right area. Within this area you can see that there are different tabs, which are associated with different views. You can see in the tabs in this section that there are different views available: *Files*, *Plots*, *Packages*, *Help*, and *Viewer*. The **Files** allow you to see the files in the physical directory that is currently set up as your working environment. You can think of it like a window in Windows Explorer that lets you see the content of a folder.

In the **plots** panel you will see any data visualisations or graphical displays of data that you produce. We haven't yet produced any, so it is empty at the moment. If you click in **packages** you will see the packages that are currently available in your installation. What is a "package" in this context?

Packages are modules that expand what R can do. There are thousands of them. A few come pre-installed when you do a basic R installation. Others you pick and install yourself. This term we will introduce some important packages we recommend and you should install.

The other really useful panel in this part of the screen is the **Help** viewer. Here you can access the documentation for the various packages that make up R. Learning how to use this documentation will be essential if you want to be able to get the most from R.

In the diagonally opposite corner, the top left, you should now have an open script window. The **script** is where you write your programming code, the instructions you send to your computer. A script is nothing but a text file where you can write. Unlike other programs for data analysis you may have used in the past (Excel, SPSS), you need to interact with R by means of writing down instructions and asking R to evaluate those instructions. R is an *interpreted* programming language: you write instructions (code) that the R engine has to interpret in order to do something. And all the instructions we write can and should be saved in a script, so that you can return later to what you did.

One of the key advantages of doing data analysis this way is that you are producing a written record of every step you take in the analysis. The challenge though is that you need to learn this language in order to be able to use it. That will be the main focus of this course, teaching you to write R code for data analysis purposes.

As with any language the more you practice it, the easier it will become. More often than not you will be doing a lot of cutting and pasting from chunks of code we will give you. But we will also expect you to develop a basic understanding of what these bits of code do. It is a bit like cooking. At first you will just follow recipes as they are given to you, but as you become more comfortable in your "kitchen" you will feel more comfortable experimenting.

The advantage of doing analysis this way is that once you have written your instructions and saved them in a file, you will be able to share it with others and run it every time you want in a matter of seconds. This creates a *reproducible* record of your analysis: something that your collaborators or someone else anywhere (including your future self, the one that will have forgotten how to do the stuff) could run and get the same results than you did at some point earlier. This makes science more transparent and transparency brings with it many advantages. For example, it makes your research more trustworthy. Don't underestimate how critical this is. **Reproducibility** is becoming a key criteria to assess good quality research. And tools like R allow us to enhance it. You may want to read more about reproducible research here.

1.3 Customising the RStudio look

RStudio allows you to customise the way it looks. For example, working with white backgrounds is not generally a good idea if you care about your eyesight. If you don't want to end up with dry eyes not only it is good you follow the 20-20-20 rule (every 20 minutes look for 20 seconds to an object located 20 feet away from you), but it may also be a good idea to use more eye friendly screen displays.

Click in the *Tools* menu and select *Global options*. This will open up a pop up window with various options. Select *Appearance*. In this section you can change the font type and size, but also the kind of theme background that R will use in the various windows. I suffer from poor sight, so I often increase the font type. I also use the *Tomorrow Night Bright* theme to prevent my eyes to go too dry from the effort of reading a lightened screen, but you may prefer a different one. You can preview them and then click apply to select the one you like. This will not change your results or analysis. This is just something you may want to do in order to make things look better and healthier for your.

1.4 Getting organised: R Projects

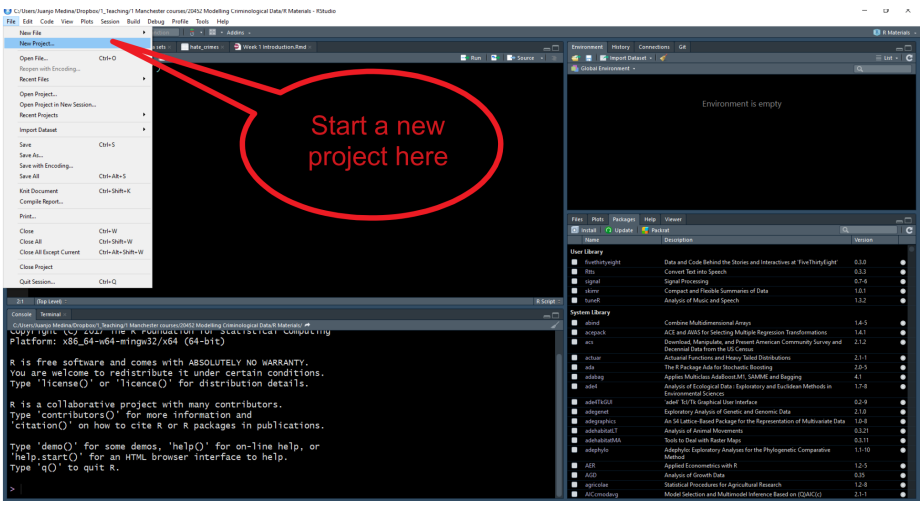
Whenever you do analysis you will be working with a variety of files. You may have an Excel dataset (or some other type of dataset file, like csv for example), a Microsoft Word file where you are writing down the essay with your results, but also a script with all the progamming code you have been using. R needs to know where all these files sit in your computer. Often you will get error messages because you are expecting R to find one of these files in the wrong location. **It is absolutely critical thus that you understand how your computer organises and store files.** Please watch the video below to understand the basics of file management and file paths:

Windows users MAC users

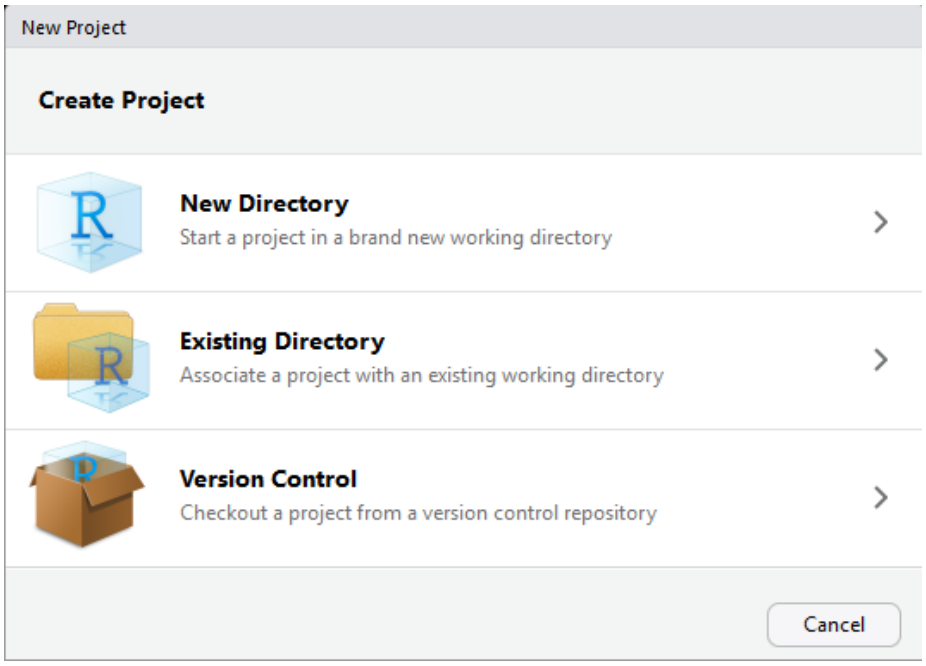
The best way to avoid problems with file management in R is using what RStudio calls **R Projects**.

Technically, a RStudio project is just a directory (a folder) with the name of the project, and a few files and folders created by R Studio for internal purposes. This is where you should hold your scripts, your data, and reports. You can manage this folder with your own operating system manager (eg., Windows Explorer) or through the R Studio file manager (that you access in the bottom right corner set of windows in R Studio).

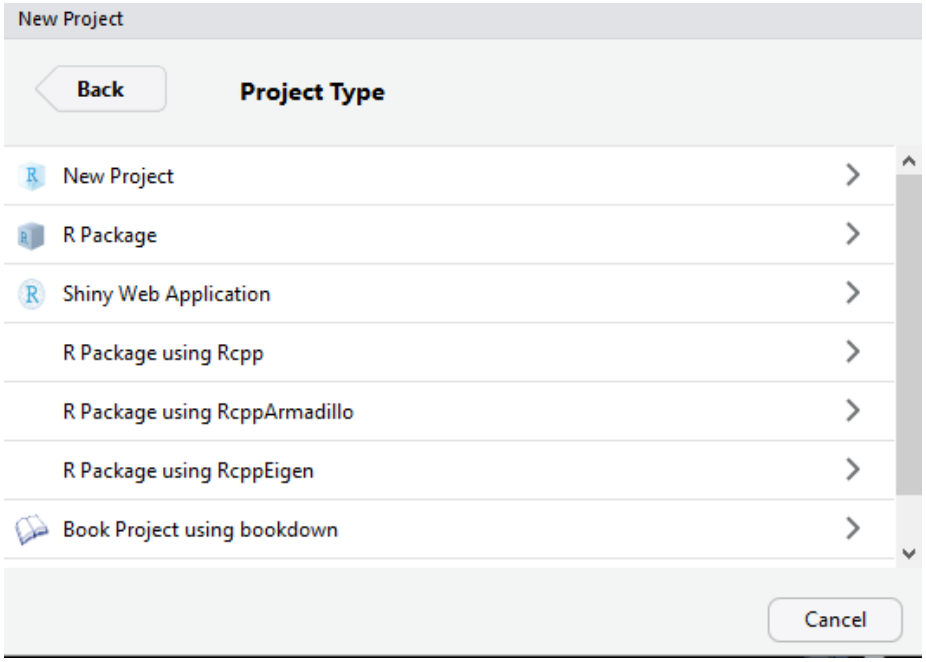
When a project is reopened, R Studio opens every file and data view that was open when the project was closed last time around. Let's learn how to create a project. Go to the *File* drown menu and select *New Project*.



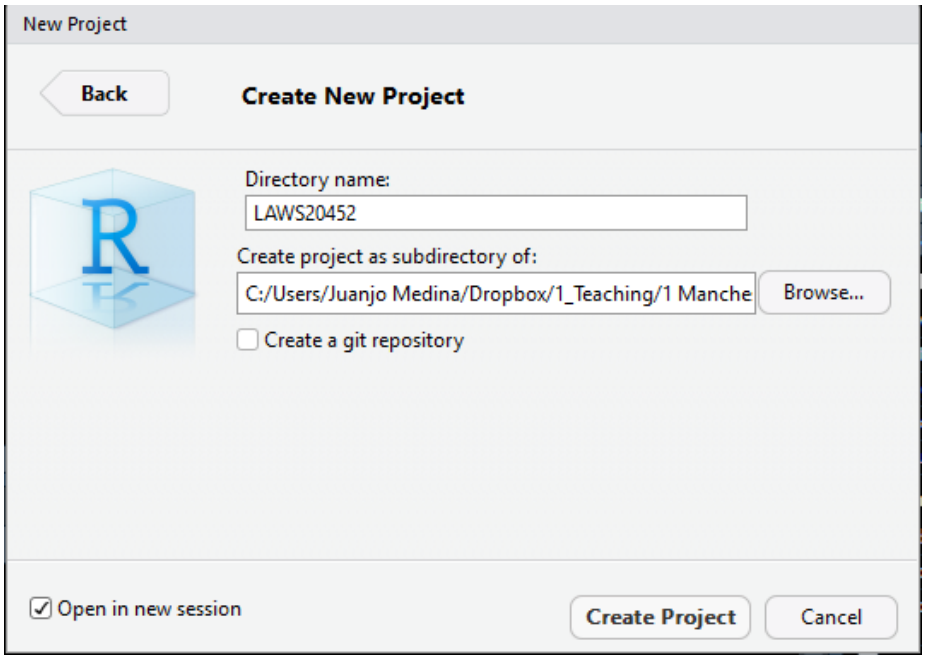
That will open a dialog box where you ask to specify what kind of directory you want to create. Select new working directory in this dialog box.



Now you get another dialog box (at least you have an older version of R Studio) where you have to specify what kind of project you want to create. Select the first option *New Project*.



Finally, you get to select a name for your project (in the image below I use the code for this course unit, but you can use any sensible name you prefer) and you will need to specify the folder/directory in which to place this directory. If you are using a cluster machine use the P: drive, otherwise select what you prefer in your laptop (preferably, to avoid problems later, not your desktop in Windows machines).



With simple projects a single script file and a data file is all you may have. But with more complex projects, things can rapidly become messy. So you may want to create subdirectories within this project folder. I typically use the following structure in my own work to put all files of a certain type in the same subdirectory:

- *Scripts and code*: Here I put all the text files with my analytic code, including rmarkdown files which is something we will introduce much later in the semester.
- *Source data*: Here I put the original data. I tend not to touch this once I have obtained the original data.
- *Documentation*: This is the subdirectory where I place all the data documentation (e.g., codebooks, questionnaires, etc.)
- *Modified data*: All analysis involve doing transformations and changing things in the original data files. You don't want to mess up the original data files, so what you should do is create new data files as soon as you

start changing your source data. I go so far as to place them in a different subdirectory.

- *Literature*: Analysis is all about answering research questions. There is always a literature about these questions. I place the relevant literature for the analytic project I am conducting in this subdirectory.
- *Reports and write up*: Here is where I file all the reports and data visualisations that are associated with my analysis.

You can create these subdirectories using Windows Explorer or the Files window in R Studio.

1.5 Talk to your computer

So far we hace covered an introduction to the main interface you will be using and talk about RStudio projects. In this unit you will be using this interface and using and creating files within your RStudio projects to produce analysis based on programming code that you will need to write using the R language.

Let's write some very simple code using R to talk to your computer. First open a new script within the project you just created. Type the following instructions in the script window. After you are done click in the top right corner where it says *Run* (if you prefer quick shortcuts, you can select the text and then press Ctrl + Enter):

```
print("I hate computers")
```

```
## [1] "I hate computers"
```

Congratulations!!! You just run your first line of R code!

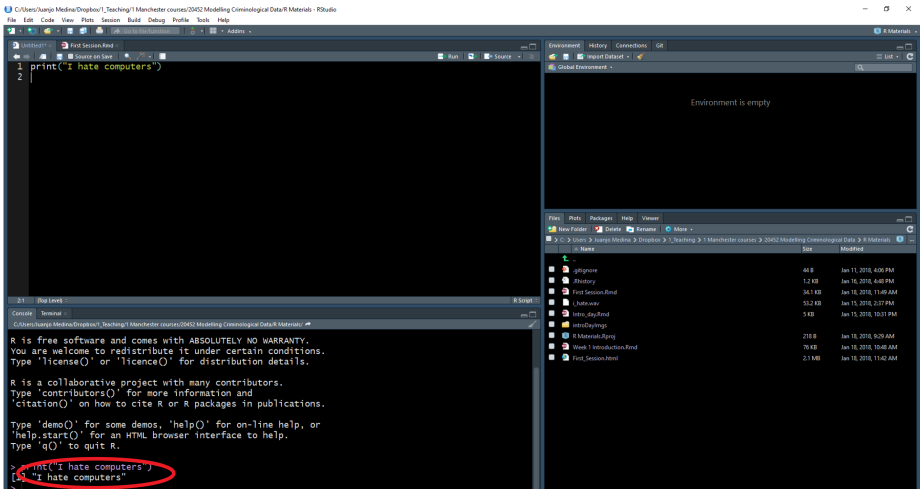
In these handouts you will see grayed boxes with bit of code on it. You can cut and paste this code into your script window and run the code from it to reproduce our results. As we go along we will be covering new bits of code.

Sometimes in these documents you will see on them the results of running the code, what you see printed in your console or in your plot viewer. The results will appear enclosed in a box as above.

The R languages uses **functions** to tell the computer what to do. In the R *language* functions are the *verbs*. You can think of functions as predefined commands that somebody has already programmed into R and tell R what to do. Here you learnt your first R function: *print*. All this function does is to ask R to print whatever it is you want in the main console (see the window in the bottom left corner).

In R, you can pass a number of **arguments** to any function. These arguments control what the function will do in each case. The arguments appear between brackets. Here we passed the text "I hate computers" as an argument. Once you execute the program, by clicking on *Run*, the R engine sends this to the CPU of your machine in the form of binary code and this produces a result. In this case we see that result printed in the main console.

Every R function admits different kind of arguments. Learning R involves not only learning different functions but also learning what are the valid arguments you can pass to each function.



As indicated above, the window in the bottom left corner is the main **console**.You will see that the words "I hate computers" appear printed there. If rather than using R Studio you were working directly from R, that's all you would get: the main console where you can write code interactively (rather than all the different windows you see in R Studio). You can write your code directly in the main console and execute it line by line in an interactive fashion. However, we will be running code from scripts, so that you get used to the idea of properly documenting all the steps you take,

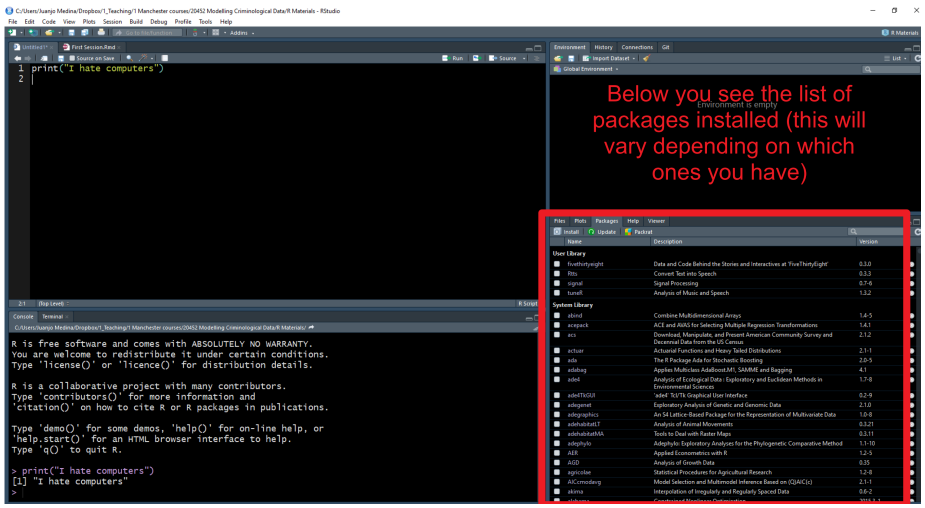
1.6 More on packages

Before we described packages as elements that add the functionality of R. What most packages do is they introduce new functions that allow you to ask R to do new different things.

Anybody can write a package, so consequently R packages vary on quality and complexity. You can find packages in different places, as well, from official repositories (which means they have passed a minimum of quality control), something called GitHub (a webpage where software developers post work in

progress), to personal webpages (danger danger!). In early 2017 we passed the 10,000 mark just in the main official repository, so the number of things that can be done with R grows exponentially every day as people keep adding new packages.

When you install R you only install a set of basic packages, not the full 10,000 plus. So if you want to use any of these added packages that are not part of the basic installation you need to first install them. You can see what packages are availabe in your local install by looking at the *packages* tab in the bottom right corner panel. Click there and check. We are going to install a package that is not there so that you see how the installation is done.



If you just installed R in your laptop you will see a shortish list of packages that constitute the basic installation of R. If you are using one of the machines in the computer cluster this list is a bit longer, because we asked IT to install some of the most commonly used packages. But knowing how to install packages is pretty essential, since you will want to do it very often.

We are going to install a package called "cowsay" to demonstrate the process. In the Packages panel there is an *Install* menu that would open a dialog box and allows you to install packages. Instead we are going to use code to do this. Just cut and paste the code below into your script and then run it:

```
install.packages("cowsay")
```

Here we are introducing a new function "install.packages" and what we have passed as an argument is the name of the package that we want to install. This is how we install a package *that is available in the official CRAN repository.* Given that you are connecting to an online repository you will need an internet connection every time you want to install a package. CRAN is an official repository that has a collection of R packages that meet a minimum set of quality

criteria. It's a fairly safe place to get packages from. If we wanted to install a package from somewhere else we would have to adapt the code. Later this semester you will see how we install packages from GitHub.

This line of code (as it is currently written) will install this package in a personal library that will be located in your P: drive if you are using a cluster machine. If you are using a Windows machine this code will place this package within a personal library in your Documents folder. Once you install a package, it will remain in the machine/location where you install it until you physically delete it.

How do you find out what a package does? You look at the relevant documentation. In the Packages window scroll down until you find the new package we installed listed. Here you will see the name of the package (cowsay), a brief description of what the program is about, and the version you have installed (an indication that a package is a good package is that it has gone through several versions, that means that someone is making sure the package gets regular updates and improvements). The version I have for cowsay is 0.7.0. Yours may be older or newer. It doesn't matter much at this point.

Click in the name *cowsay*. You will see that R Studio has now brought you to the Help tab. Here is where you find the help files for this package, including all the available documentation.

Every beginner in R will find these help files a bit confusing. But after a while, their format and structure will begin to make sense to you. Click where it says *User guides, package vignettes, and other documentation*. Documentation in R has become much better since people started to write **vignettes** for their packages. They are little tutorials that explain with examples what each package does. Click in the *cowsay::cowsay_tutorial* that you see listed here. What you will find there is an html file that gives you a detailed tutorial on this package. You don't need to read it now, but remember that this is one way to find help when using R.

Let's try to use some of the functions of this package. We will use the "say" function:

```
say("I hate computers")
```

You will get an error message telling you that this function could not be found. What happened?? This will be the first of many error messages you will get. An error message is the computer way of telling you that your instructions are somehow incomplete or problematic and, thus, is unable to do what you ask. It is frustrating to get these messages and a critical skill for you this semester will be to overcome that frustration and try to understand why the computer cannot do what you ask. Finding out the source of the error and solving it is what these labs are all about. There is nothing wrong with getting errors. The problem is if you give up and let your frustration get the best of you.

So why are we getting this error? Installing a package is only the first step. The next step, when you want to use it in a given session, is to **load** it.

Think of it as a pair of shoes. You buy it once, but you have to take them from your closet and put them on when you want to use them. Same with packages, you only install once, but need to load it from your library every time you want to use it -within a given session (once loaded it will remain loaded until you finish your session).

To see what packages you currently have **loaded** in your session, you use the `search()` function (you do not need to pass it any arguments in this case).

```
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

If you run this code, you will see that `cowsay` is not in the list of loaded packages. Therefore your computer cannot use any of the functions associated with it until you load it. To load a package we use the **library** function. So if we want to load the new package we installed in our machine we would need to use the following code:

```
library("cowsay")
```

Run the `search` function again. You will see now this package is listed. So now we can try using the function "say" again.

```
say("I hate computers")
```

```
##
##  --------------
## I hate computers
##  --------------
##     \
##       \
##         \
##             |\___/|
##           ==) ^Y^ (==
##             \  ^  /
##              )=*=(
##             /     \
##             |     |
##            /| | | |\
```

```
##            \| | |_|/\
##       jgs  //_// ___/
##                \_)
##
```

You get a random animal in the console repeating the text we passed as an argument. If we like a different animal we could pass an argument to select it. So, say, we want to have cow rather than a random animal, then we would pass the following arguments to our function.

```
say("I hate computers", "cow")
```

```
##
##  -----
## I hate computers
##  ------
##     \   ^__^
##      \  (oo)\ ________
##         (__)\         )\ /\
##              ||------w|
##              ||      ||
```

This is an important feature of arguments in functions. We said how different functions admits different arguments. Here by specifying `cow` the function is printing that particular animal. But who is it that when we didn't specify a particular kind of animal we still got a result? That happened because functions always have default arguments that are necessary for them to run and that you do not have to make explicit. Default arguments are implicit and do not have to be typed. The `say` function has a default argument `random` which will print a random character or animal. It is only when you want to change the default that you need to make an alternative animal explicit.

Remember, you only have to install a package that is not already installed once. But if you want to use it in a given session you will have to load it within that session using the `library` function. Once you load it within a session the package will remain loaded until you terminate your session (for example, by closing R Studio). Do not forget this.

1.7 Using objects

We have seen how the first argument that the "say" function takes is the text that we want to convert into speech for our given animal. We could write the text directly into the function (as we did above), but now we are going to do something different. We are going to create an object to store the text.

An **object**? What do I mean? In the same way that everything you do in R you do with functions (your verbs), everything that exist in R is an object. You can think of objects as boxes where you put stuff. In this case we are going to create an object called *my_text* and inside this object we are going to store the text "I hate computers". How do you do this? We will use the code below:

```
my_text <- "I hate computers."
```

This bit of code is simply telling R we are creating a new object with the assigned name ("my_text"). We are creating a box with such name and inside this box we are placing a bit of text ("I hate computers"). The arrow you see is the **assignment operator**. This is an important part of the R language that tells R what we are including inside the object in question.

Run the code. Look now at the *Environment* window in the right top corner. We see that this object is now listed there. You can think of the Environment as a warehouse where you put stuff in, your different objects. Is there a limit to this environment? Yes, your RAM. R works on your RAM, so you need to be aware that if you use very large objects you will need loads of RAM. But that won't be a problem you will encounter in this course unit.

Once we put things into these boxes or objects we can use them as arguments in our functions. See the example below:

```
say(my_text, "cow")
```

```
##
##  -----
## I hate computers.
##  ------
##     \   ^__^
##      \  (oo)\ ________
##         (__)\         )\ /\
##              ||------w|
##              ||      ||
```

1.8 More on objects

Now that we have covered some of the preliminaries we can move to talk about data. In Excel you are used to see your data in a spreadsheet format. If you did the prep for this session, you should have reviewed some of the materials we covered in *Making Sense of Criminological Data* last semester. You should be familiar with the notion of a data set, levels of measurement, and tidy data. If you have not. This is your chance to do it in this link.

R is considerably more flexible than Excel. Most of the work we do here will use data sets or **dataframes** as they are called in R. But as you have seen earlier you can have *objects* other than data frames in R. These objects can relate to external files or simple textual information ("I hate computers"). This flexibility is a big asset because among other things it allow us to break down data frames or the results from doing analysis on them to its constitutive parts (this will become clearer as we go along).

Technically R is an *Object Oriented language*. Object-oriented programming (OOP) is a programming language model organized around objects rather than "actions" and data rather than logic.

As we have seen earlier, to create an object you have to give it a name, and then use the assignment operator (the `<-` symbol) to assign it some value.

For example, if we want to create an object that we name "x", and we want it to represent the value of 5, we write:

```
x <- 5
```

We are simply telling R to create a **numeric object**, called `x`, with one element (5) or of length 1. It is numeric because we are putting a number inside this object. The length is 1 because it only has one element on it, the number 5.

You can see the content of the object `x` in the main console either by using the print function we used earlier or by auto-printing, that is, just typing the name of the object and running that as code:

```
x
```

```
## [1] 5
```

When writing expressions in R is very important you understand that **R is case sensitive**. This could drive you nuts if you are not careful. More often than not if you write an expression asking R to do something and R returns an error message, chances are that you have used lower case when upper case was needed (or vice-versa). So always check for the right spelling. For example, see what happens if I use a capital 'X':

```
X
```

```
## Error in eval(expr, envir, enclos): object 'X' not found
```

You will get the following message: `"Error in eval(expr, envir, enclos): object 'X' not found"`. R is telling us that `X` does not exist. There isn't an object `X` (upper case), but there is an object `x` (lower case). Error messages in R are pretty good at telling you exactly what went wrong.

Remember computers are very literal. They are like dogs. You can tell a dog "sit" and if it has been trained it will sit. But if you tell a dog "would you be so kind as to relax a bit and lay down in the sofa?", it won't have a clue what you are saying and will stare at you like you have gone mad. Error messages are computers ways of telling us "I really want to help you but I don't really understand what you mean" (never take them personal, computers don't hate you).

When you get an error message or implausible results, you want to look back at your code to figure out what is the problem. This process is called **debugging**. There are some proper systematic ways to write code that facilitate debugging, but we won't get into that here. R is very good with automatic error handling at the levels we'll be using it at. Very often the solution will simply involve correcting the spelling.

A handy tip is to cut and paste the error message into Google and find a solution. If anybody had given me a penny for every time I had to do that myself, I would be Bill Gates by now. You're probably not the first person to make your mistake, after all, and someone on the internet has surely already found a solution to your issue. People make mistakes all the time. It's how we learn. Don't get frustrated, don't get stuck. Instead look for a solution. These days we have Google. We didn't back in the day. Now you have the answer to your frustration within quick reach. Use it to your advantage.

##Vectors

In R there are different kind of objects. We will start with **vectors**.

What is a vector? A vector is simply a set of elements *of the same class* (typically these classes are: character, numeric, integer, or logical -as in True/False). Vectors are the basic data structure in R.

Typically you will use the `c()` function (c stands for concatenate) to create vectors. The code below exemplifies how to create vectors of different classes (numeric, logical, etc.). Notice how the listed elements (to simplify there are two elements in each vector below) are separated by commas:

```
my_1st_vector <- c(0.5, 0.6) #creates a numeric vector with two elements
my_2nd_vector <- c(1L, 2L) #creates an integer vector
my_3rd_vector <- c(TRUE, FALSE) #creates a logical vector
my_4th_vector <- c(T, F) #creates a logical vector using abbreviations of True and Fal
my_5th_vector <- c("a", "b", "c") #creates a character vector
my_6th_vector <- c(1+0i, 2+4i) #creates a complex vector (we won't really use this cla
```

Cut and paste this code into your script and run it. You will see how all these vectors are added to your global environment and stored there.

The beauty of an object oriented statistical language like R is that one you have these objects you can use them as **inputs** in functions, use them in operations, or to create other objects. This makes R very flexible. See some examples below:

```
class(my_1st_vector) #a function to figure out the class of the vector
```

```
## [1] "numeric"
```

```
length(my_1st_vector) #a function to figure out the length of the vector
```

```
## [1] 2
```

```
my_1st_vector + 2 #Add a constant to each element of the vector
```

```
## [1] 2.5 2.6
```

```
my_7th_vector <- my_1st_vector + 1 #Create a new vector that contains the elements of my1stvector
my_1st_vector + my_7th_vector #Adds the two vectors and auto-print the results (note how the sum
```

```
## [1] 2.0 2.2
```

As indicated earlier, when you create objects you will place them in your working memory or workspace. Each R session will be associated to a workspace (called "global environment" in R Studio). In R Studio you can visualise the objects you have created during a session in the **Global Environment** screen. But if you want to produce a list of what's there you can use the `ls()` function (the results you get my differ from the ones below depending on what you actually have in your global environment).

```
ls() #list all objects in your global environment
```

```
## [1] "my_1st_vector" "my_2nd_vector" "my_3rd_vector" "my_4th_vector"
## [5] "my_5th_vector" "my_6th_vector" "my_7th_vector" "my_text"
## [9] "x"
```

If you want to delete a particular object you can do so using the `rm()` function.

```
rm(x) #remove x from your global environment
```

It is also possibly to remove all objects at once:

```
rm(list = ls()) #remove all objects from your global environment
```

If you mix in a vector elements that are of a different class (for example numerical and logical), R will **coerce** to the minimum common denominator, so that every element in the vector is of the same class. So, for example, if you input a number and a character, it will coerce the vector to be a character vector -see the example below and notice the use of the `class()` function to identify the class of an object.

```
my_8th_vector <- c(0.5, "a")
class(my_8th_vector) #The class() function will tell us the class of the vector
```

```
## [1] "character"
```

1.9 On comments

In the bits of code above you will have noticed parts that were grayed out. See for example in the last example provided. You can see that after the hashtag all the text is being grayed out. What is this? What's going on?

These are **comments**. Comments are simply annotations that R will know is not code (and therefore doesn't attempt to understand and execute). We use the hash-tag symbol to specify to R that what comes after is not programming code, but simply bits of notes that we write to remind ourselves what the code is actually doing. Including these comments will help you to understand your code when you come back to it.

To create a comment you use the hashtag/ number sign `#` followed by some text. Whenever the R engine sees the number sign it knows that what follows is not code to be executed. You can use this sign to include *annotations* when you are coding. These annotations are a helpful reminder to yourself (and others reading your code) of **what** the code is doing and (even more important) **why** you are doing it.

It is good practice to often use annotations. You can use these annotations in your code to explain your reasoning and to create "scannable" headings in your code. That way after you save your script you will be able to share it with others or return to it at a later point and understand what you were doing when you first created it -see here for further details on annotations and in how to save a script when working with the basic R interface.

Just keep in mind: + You need one `#` per line, and anything after that is a comment that is not executed by R.

- You can use spaces after (its not like a hashtag on twitter).

1.10 Factors

An important thing to understand in R is that categorical (ordered, also called ordinal, or unordered, also called nominal) data are *typically* encoded as **factors**, which are just a special type of vector. A factor is simply an integer vector that can contain *only predefined values* (this bit is very important), and is used to store categorical data. Factors are treated specially by many data analytic and visualisation functions. This makes sense because they are essentially different from quantitative variables.

Although you can use numbers to represent categories, *using factors with labels is better than using integers to represent categories* because factors are self-describing (having a variable that has values "Male" and "Female" is better than a variable that has values "1" and "2" to represent male and female). When R reads data in other formats (e.g., comma separated), by default it will automatically convert all character variables into factors. If you rather keep these variables as simple character vectors you need to explicitly ask R to do so. We will come back to this next week with some examples.

Factors can also be created with the `factor()` function concatenating a series of *character* elements. You will notice that is printed differently from a simply character vector and that it tells us the levels of the factor (look at the second printed line).

```
the_smiths <- factor(c("Morrisey", "Marr", "Rourke", "Joyce")) #create a new factor
the_smiths #auto-print the factor
```

```
## [1] Morrisey Marr     Rourke   Joyce
## Levels: Joyce Marr Morrisey Rourke
```

```
#Alternatively for similar result using the as.factor() function
the_smiths_bis <- c("Morrisey", "Marr", "Rourke", "Joyce") #create a character vector
the_smiths_f <- as.factor(the_smiths_bis) #create a factor using a character vector
the_smiths_f #auto-print factor
```

```
## [1] Morrisey Marr     Rourke   Joyce
## Levels: Joyce Marr Morrisey Rourke
```

Factors in R can be seen as vectors with a bit more information added. This extra information consists of a record of the distinct values in that vector, called **levels**. If you want to know the levels in a given factor you can use the `levels()` function:

```
levels(the_smiths)
```

```
## [1] "Joyce"    "Marr"     "Morrisey" "Rourke"
```

Notice that the levels appear printed by alphabetical order. There will be situations when this is not the most convenient order. Later on we will discuss in these tutorials how to reorder your factor levels when you need to.

1.11 Naming conventions for objects in R

You may have noticed the various names I have used to designate objects (`my_1st_vector`, `the_smiths`, etc.). You can use almost any names you want for your objects. Objects in R can have names of any length consisting of letters, numbers, underscores ("_") or the period (".") and should begin with a letter. In addition, when naming objects you need to remember:

- *Some names are forbidden.* These include words such as FALSE and TRUE, logical operators, and programming words like Inf, for, else, break, function, and words for special entities like NA and NaN.
- *You want to use names that do not correspond to a specific function.* We have seen, for example, that there is a function called `print()`, you don't want to call an object "print" to avoid conflicts. To avoid this use nouns instead of verbs for naming your variables and data.
- *You don't want them to be too long* (or you will regret it every time you need to use that object in your analysis: your fingers will bleed from typing).
- *You want to make them as intuitive to interpret as possible.*
- *You want to follow consistent naming conventions.* R users are terrible about this. But we could make it better if we all aim to follow similar conventions. In these handouts you will see I follow the `underscore_separated` convention -see here for details.

It is also important to remember that R will always treat numbers as numbers. This sounds straightforward, but actually it is important to note. We can name our variables almost anything. EXCEPT they cannot be numbers. Numbers are **protected** by R. 1 will always mean 1.

If you want, give it a try. Try to create a variable called 12 and assign it the value "twelve". As we did last week, we can assign something a meaning by using the "<-" characters.

```
12 <- "twelve"
```

```
## Error in 12 <- "twelve": invalid (do_set) left-hand side to assignment
```

You get an error!

1.12 Dataframes

Ok, so now that you understand some of the basic types of objects you can use in R, let's start taking about data frames. One of the most common objects you will work with in this course are **data frames**. Data frames can be created with the `data.frame()` function.

Data frames are *multiple vectors* of possibly different classes (e.g., numeric, factors), but of the same length (e.g., all vectors, or variables, have the same number of rows). This may sound a bit too technical but it is simply a way of saying that a data frame is what in other programmes for data analysis gets represented as data sets, the tabular spreadsheets you have seen when using Excel.

Let's create a data frame with two variables:

```
#We create a dataframe called mydata.1 with two variables, an integer vector called foo and a log
mydata_1 <- data.frame(foo = 1:4, bar = c(T,T,F,F))
mydata_1
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

Or alternatively for the same result:

```
x <- 1:4
y <- c(T, T, F, F)
mydata_2 <- data.frame (foo = x, bar = y)
mydata_2
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

As you can see in R, as in any other language, there are multiple ways of saying the same thing. Programmers aim to produce code that has been optimised: it is short and quick. It is likely that as you develop your R skills you find increasingly more efficient ways of asking R how to do things. What this means too is that when you go for help, from your peers or us, we may teach you slightly different ways of getting the right result. As long as you get the right result that's what at this point really matters.

These are silly toy examples of data frames. In this course, we will use real data. Next week we will learn in greater detail how to read data into R. But you should also know that R comes with pre-installed data sets. Some packages in fact are nothing but collections of data frames.

Let's have a look at some of them. We are going to look at some data that are part of the *fivethirtyeight* package. This package contains data sets and code behind the stories in this particular online newspaper. This package is not part of the base installation of R, so you will need to install it first. I won't give you the code for it. See if you can figure it out by looking at previous examples.

Done? Ok, now we are going to look at the data sets that are included in this package. Remember first we have to load the package if we want to use it:

```
library("fivethirtyeight")
```

```
## Some larger datasets need to be installed separately, like senators and
## house_district_forecast. To install these, we recommend you install the
## fivethirtyeightdata package by running:
## install.packages('fivethirtyeightdata', repos =
## 'https://fivethirtyeightdata.github.io/drat/', type = 'source')
```

```
data(package="fivethirtyeight") #This function will return all the data frames that ar
```

Notice that this package has some data sets that relate to stories covered in this journal that had a criminological angle. Let's look for example at the hate_crimes data set. How do you that? First we have to load the data frame into our global environment. To do so use the following code:

```
data("hate_crimes")
```

This function will search among all the *loaded* packages and locate the "hate_crimes" data set. Notice that it now appears in the global environment, although it also says "promise" next to it. To see the data in full you need to do something to it first. So let's do that.

Every object in R can have **attributes**. These are: names; dimensions (for matrices and arrays: number of rows and columns) and dimensions names;

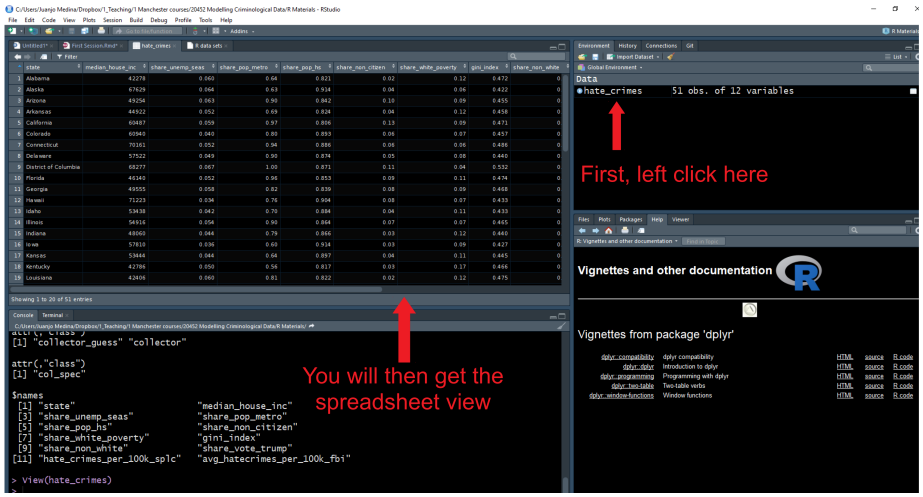
class of object (numeric, character, etc.); length (for a vector this will be the number of elements in the vector); and other user-defined. You can access the attributes of an object using the `attributes()` function. Let's query R for the attributes of this data frame.

```
attributes(hate_crimes)
```

```
## $row.names
##  [1]  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
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
## [51] 51
##
## $class
## [1] "tbl_df"     "tbl"        "data.frame"
##
## $names
##  [1] "state"                          "state_abbrev"
##  [3] "median_house_inc"               "share_unemp_seas"
##  [5] "share_pop_metro"                "share_pop_hs"
##  [7] "share_non_citizen"              "share_white_poverty"
##  [9] "gini_index"                     "share_non_white"
## [11] "share_vote_trump"               "hate_crimes_per_100k_splc"
## [13] "avg_hatecrimes_per_100k_fbi"
```

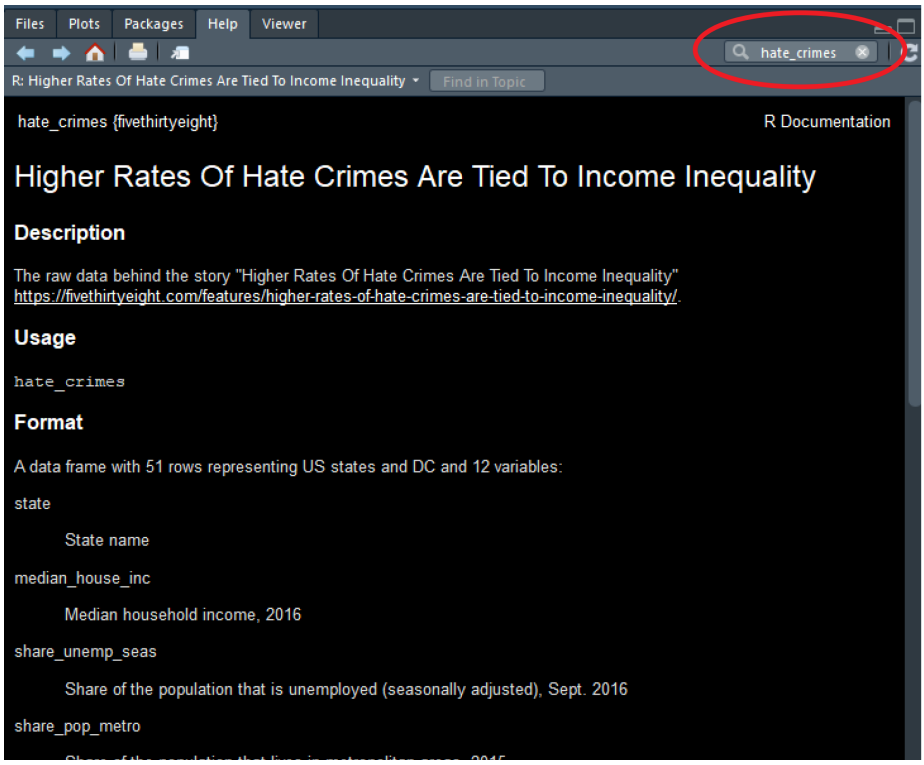
These results printed in the may console may not make too much sense to you at this point. We will return to this next week, so do not worry.

Go now to the global environment panel and left click on the data frame "hate_crimes". This will open the data viewer in the top left section of R Studio. What you get there is a spreadsheet with 12 variables and 51 observations. Each variable in this case is providing you with information (demographics, voting patterns, and hate crime) about each of the US states.



1.13 Exploring data

Ok, let's now have a quick look at the data. There are so many different ways of producing summary stats for data stored in R that is impossible to cover them all! We will just introduce a few functions that you may find useful for summarising data. Before we do any of that it is important you get a sense for what is available in this data set. Go to the help tab and in the search box input the name of the data frame, this will take you to the documentation for this data frame. Here you can see a list of the available variables.



Let's start with the *mean*. This function takes as an argument the numeric variable for which you want to obtain the mean. Because of the way that R works you cannot simply put the name of the variable you have to tell R as well in which data frame is that variable located. To do that you write the name of the data frame, the dollar sign(`$`), and then the name of the variable you want to summarise. If you want to obtain the mean of the variable that gives us the proportion of people that voted for Donald Trump you can use the following expression:

```
mean(hate_crimes$share_vote_trump)
```

```
## [1] 0.49
```

This code is saying look inside the "hate_crimes" dataset object and find the "share_vote_trump" variable, then print the mean. The `$` is used when you want to find a particular component of an object. In the case of dataframes that component typically will be one of the vectors (variables). But we will see other uses for other kind of objects as we move through the course.

Another function you may want to use with numeric variables is `summary()`:

```
summary(hate_crimes$share_vote_trump)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.040   0.415   0.490   0.490   0.575   0.700
```

This gives you the five number summary (minimum, first quartile, median, third quartile, and maximum, plus the mean and the count of missing values if there are any).

You don't have to specify a variable you can ask for these summaries from the whole data frame:

```
summary(hate_crimes)
```

```
##     state           state_abbrev       median_house_inc share_unemp_seas
##  Length:51          Length:51          Min.   :35521    Min.   :0.02800
##  Class :character   Class :character   1st Qu.:48657    1st Qu.:0.04200
##  Mode  :character   Mode  :character   Median :54916    Median :0.05100
##                                        Mean   :55224    Mean   :0.04957
##                                        3rd Qu.:60719    3rd Qu.:0.05750
##                                        Max.   :76165    Max.   :0.07300
##
##  share_pop_metro   share_pop_hs    share_non_citizen share_white_poverty
##  Min.   :0.3100   Min.   :0.7990   Min.   :0.01000   Min.   :0.04000
##  1st Qu.:0.6300   1st Qu.:0.8405   1st Qu.:0.03000   1st Qu.:0.07500
##  Median :0.7900   Median :0.8740   Median :0.04500   Median :0.09000
##  Mean   :0.7502   Mean   :0.8691   Mean   :0.05458   Mean   :0.09176
##  3rd Qu.:0.8950   3rd Qu.:0.8980   3rd Qu.:0.08000   3rd Qu.:0.10000
##  Max.   :1.0000   Max.   :0.9180   Max.   :0.13000   Max.   :0.17000
##                                    NA's   :3
##    gini_index     share_non_white  share_vote_trump hate_crimes_per_100k_splc
##  Min.   :0.4190   Min.   :0.0600   Min.   :0.040    Min.   :0.06745
##  1st Qu.:0.4400   1st Qu.:0.1950   1st Qu.:0.415    1st Qu.:0.14271
##  Median :0.4540   Median :0.2800   Median :0.490    Median :0.22620
##  Mean   :0.4538   Mean   :0.3157   Mean   :0.490    Mean   :0.30409
##  3rd Qu.:0.4665   3rd Qu.:0.4200   3rd Qu.:0.575    3rd Qu.:0.35694
##  Max.   :0.5320   Max.   :0.8100   Max.   :0.700    Max.   :1.52230
##                                                     NA's   :4
##  avg_hatecrimes_per_100k_fbi
##  Min.   : 0.2669
##  1st Qu.: 1.2931
##  Median : 1.9871
##  Mean   : 2.3676
##  3rd Qu.: 3.1843
```

```
##  Max.   :10.9535
##  NA's   :1
```

So you see how now we are getting this info for all variables in one go.

There are multiple ways of getting results in R. Particularly for basic and intermediate-level statistical analysis many core functions and packages can give you the answer that you are looking for. For example, there are a variety of packages that allow you to look at summary statistics using functions defined within those packages. You will need to install these packages before you can use them.

I am only going to introduce one of them here *skimr*. It is neat and is maintained by one of my former stats teachers, the criminologist Elin Waring, an example of kindness and dedication to her students.

You will need to install it before anything else. Use the code you have learnt to do so and then load it. I won't be providing you the code for it, by now you should now how to do this.

Once you have loaded the *skimr* package you can use it. Its main function is *skim*. Like *summary* for data frames, skim presents results for all the columns and the statistics will depend on the class of the variable. However, the results are displayed and stored in a nicer way - though we won't get into the details of this right now.

```
skim(hate_crimes)
```

skim_type	skim_variable	n_missing	complete_rate	character.min	character.max	cha
character	state	0	1.0000000	4	20	
character	state_abbrev	0	1.0000000	2	2	
numeric	median_house_inc	0	1.0000000	NA	NA	
numeric	share_unemp_seas	0	1.0000000	NA	NA	
numeric	share_pop_metro	0	1.0000000	NA	NA	
numeric	share_pop_hs	0	1.0000000	NA	NA	
numeric	share_non_citizen	3	0.9411765	NA	NA	
numeric	share_white_poverty	0	1.0000000	NA	NA	
numeric	gini_index	0	1.0000000	NA	NA	
numeric	share_non_white	0	1.0000000	NA	NA	
numeric	share_vote_trump	0	1.0000000	NA	NA	
numeric	hate_crimes_per_100k_splc	4	0.9215686	NA	NA	
numeric	avg_hatecrimes_per_100k_fbi	1	0.9803922	NA	NA	

Apart from summary statistics, last semester we discussed a variety of ways to graphically display variables. In week 3 we covered scatterplots, a graphical device to show the relationship between two quantitative variables. I don't know if you remember the amount of point and click you had to do in Excel for getting this done. If not you can review the notes here.

1.14 Quitting RStudio

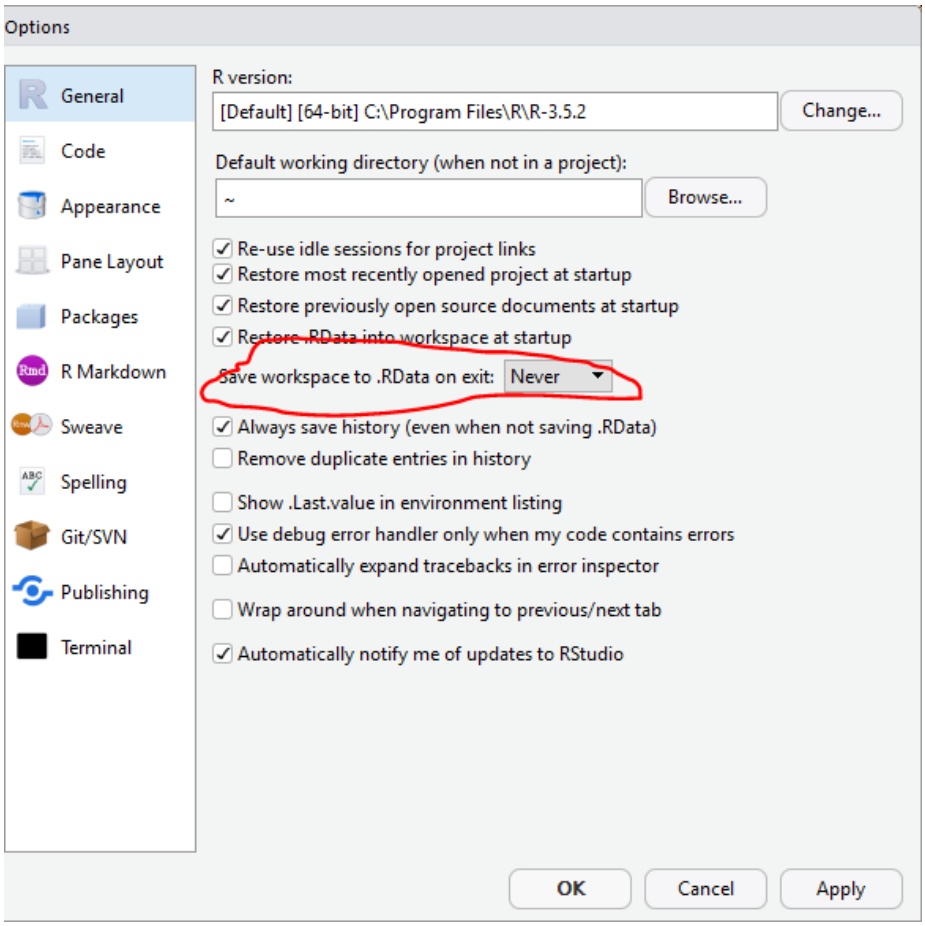
At some point, you will quit your R/R Studio session. I know, hard to visualise, right? Why would you want to do that? Anyhow, when that happens R Studio will ask you a hard question: "Save work space image to bla bla bla/.RData?" What to do? What does that even mean?

If you say "yes" what will happen is that all the objects you have in your environment will be preserved, alongside the *History* (which you can access in the top right set of windows) listing all the functions you have run within your session. So, next time you open this project all will be there. If you think that what is *real* is those objects and that history, well then you may think that's what you want to do.

Truth is what is real is your scripts and the data that your scripts use as inputs. You don't need anything that is in your environment, because you can recreate those things by re-running your scripts. I like keeping things tidy, so when I am asked whether I want to save the image, my answer is always no. Most long time users of R never save the workspace, nor care about saving the history either. Remember what is real is your scripts and the data.

Keep in mind though that you should not then panic if you open your next R Studio session and you don't see any objects in your environment. The good news is you can generate them quickly enough (if you really need them) by re-running your scripts. I would suggest that at this point it may be helpful for you to get into this habit as well. I suspect otherwise you will be in week 9 of the semester and have an environment full of garbage you don't really need.

What is more. I would suggest you go to the Tools drop down menu, select Global Options, and make sure you select "Never" where it says "Save workspace". Then click "Apply". This way you will never be asked to save what is in your global environment when you terminate a session.


Options
General
Code
Appearance
Pane Layout
Packages
R Markdown
Sweave
Spelling
Git/SVN
Publishing
Terminal
R version:
[Default] [64-bit] C:\Program Files\R\R-3.5.2
Change...
Default working directory (when not in a project):
~
Browse...
Re-use idle sessions for project links
Restore most recently opened project at startup
Restore previously open source documents at startup
Restore .RData into workspace at startup
Save workspace to .RData on exit: Never
Always save history (even when not saving .RData)
Remove duplicate entries in history
Show .Last.value in environment listing
Use debug error handler only when my code contains errors
Automatically expand tracebacks in error inspector
Wrap around when navigating to previous/next tab
Automatically notify me of updates to RStudio
OK
Cancel
Apply

Chapter 2

Getting to know your data

2.1 Causality in social sciences

In today's session we will refresh themes you have explored in previous research methods courses, specifically causality. This is a central concept in empirical research. We often do research because we want to make causal inferences. We want to be in a position where we establish whether an intervention or a social process is causally linked to crime or some other relevant criminological outcome.

Making causal inferences often involves making comparisons. For example, between cases that have been subject to an intervention and cases that have not being subject to the causal process we are trying to investigate. But you should already know that not all kind of research comparisons are the same. In previous methods courses you must have discussed the differences between experimental and observational studies. This different kinds of research designs have a bearing on your ability to make causal inference.

Let's think about a specific case so that this makes more sense. Is there discrimination against former offenders in the labor market? In other words, are offenders less likely to find employment after release because of prejudice among employers? Or can we say that the fact that former offenders are less likely to be in employment may be due to other factors? Perhaps, they are less skilled. Perhaps they have less social capital, people they know that can help them to get jobs or to learn about job opportunities. Perhaps, they are less interested in finding employment?

Only in comparisons when other things are equal you can make causal inferences. It would only be fair to compare John, an ex-offender with a high school degree and X number of personal connections and Y numbers of professional experience, with Peter, a non-ex offender, with the same educational and pro-

fessional credentials than John (and everything else that matters when getting jobs also being equal between Peter and John).

How can you do that? How can you create situations when other things are equal? Well, that's what courses in research design are oriented to teach you. What is important for you to remember is that the way data are generated, the way you do your study, will, of course, affect the kind of interpretations that you make from your statistical comparisons. And not all studies are created equal. Some research designs put you in a better position than others to make causal inference.

You should have learnt by now that the "bronze" standard for establishing causality in the social sciences is the randomised experiment. In a randomised trial, the researchers change the causal variable of interest for a group using something like a coin toss. As Angrist and Pischke (2015: xiii) highlight:

"By changing circumstances randomly, we make it highly likely that the variable of interest is unrelated to the many other factors determining the outcomes we mean to study… Random manipulation makes other things equal hold on average across the groups that did and did not experience manipulation"

So, say you want to establish whether arresting a perpetrator may have a deterrent effect on subsequent domestic abuse. You could randomise, basically use the equivalent of a lottery, to decide whether the police officer is going to arrest the perpetrator or not and then compare those you arrest with those you don't arrest. Because you are randomising your treatment (the arrest) on average the treatment and the control group on the long run should be fairly similar and any differences you observe between them in the outcome of interest (domestic abuse recidivism) you could link it to your intervention -if you are interested in the answer to this you can read about it here.

In this session we will look at data from a randomised trial that tried to establish whether there is discrimination in the labour market against former offenders. In doing so, we will also learn various functions used in R to read data, transform data, and to obtain summary statistics for groups. We will also very quickly introduce a plotting function used in R to generate graphics.

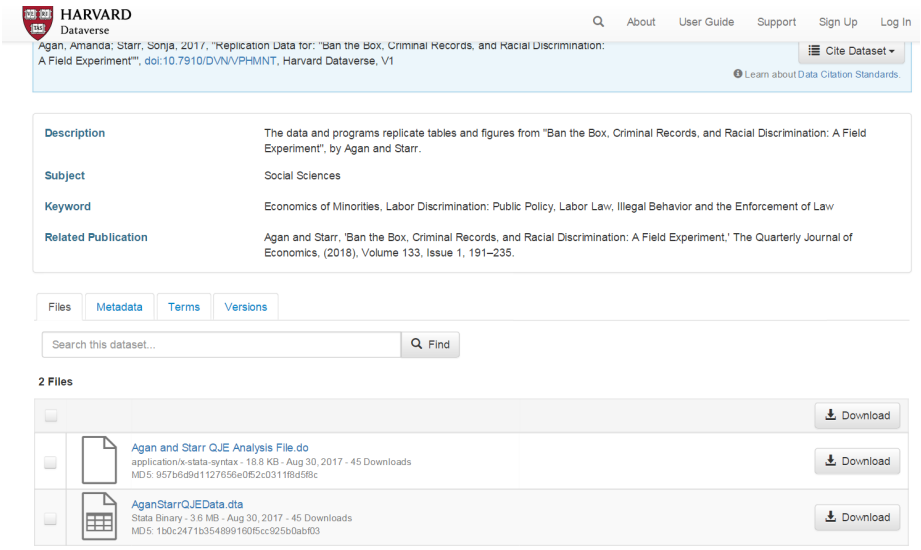
2.2 Getting data thanks to reproducibility

Last week we introduced the notion of reproducible research and said that using and publishing code (particularly if using open source tools like R) is the way that many researchers around the world think that science ought to be done. This way of operating makes research more open, more credible, and more legitimate. It also means that we can more easily access the data used in published research. For this session we are going to use the data from this and this paper study. In this research project, the authors tried to answer the question of whether criminal antecedents and other personal characteristics have

an impact on access to employment. You can find more details about this work in episode 8 of *Probable Causation*, the criminology podcast.

Amanda Agan and Sonja Starr developed a randomised experiment in which they created 15,220 fake resumes randomly generating these critical characteristics (such as having a criminal record) and used these resumes to send online job applications to low-skill, entry level job openings in New Jersey and New York City. All the fictitious applicants were male and about 21 to 22 years old. These kind of experiments are very common among researchers that want to explore through these "audits" whether some personal characteristics are discriminated against in the labor market.

Because Amanda Agan and Sonja Starr conformed to reproducible standards when doing their research we can access this data from the *Harvard Dataverse* (a repository for open research data). Click here to locate the data.



In this page you can see a download section and some files that can be accessed. One of them contains analytic code pertaining to the study and the other contains the data. You also see a link called **metadata**. Metadata is data about data, it simply provides you with some information about the data. If you clink in metadata you will see at the bottom a reference to the software the authors used (STATA). So we know these files are in STATA proprietary format. Let's download the data file and then read the data into R.

You could just click "download" and then place the file in your project directory. Alternatively, and preferably, you may want to use code to make your whole work more reproducible. Think of it this way, every time you click or use drop down menus you are doing things that others cannot reproduce because there won't be a written record of your steps. You will need though to do some clicking for finding out the required url you will use for writing your code. The

file we want is the `AganStarrQJEData.dta`. Click in the name of this file. You will be taken to another webpage. On it you will see the download url. Copy and paste this url in your code below.

```
#First, let's create an object with the link, paste the copied address here:
data_url <- "https://dataverse.harvard.edu/api/access/datafile/3036350"
```

This data file is a STATA .dta file in our working directory. To read STATA files we will need the *haven* package. This is a package developed for importing different kind of data files into R. If you don't have it you will need to install it. And then load it.

```
library(haven)
#Now we can use the read_dta function, within this function we will
#pass an argument, url, specifying to the read_dta function the need to
#make a url connection. The url function takes as an argument the url
#we are using and that we encoded in the data_url object in our case.
banbox <- read_dta(url(data_url))
```

You will need to pay attention to the file extension, to find the appropriate function to read in your file. For example, if something has the extension `.sav` that is a file used by the software SPSS. To read this, you would use the `read_spss()` function, also in the haven package.

Some other file types need other packages. For example, to read in comma separated values or `.csv` files, you can use the `read_csv()` function from the `readr` package. To read in excel files, you would use the appropriate function from the `readxl` package, which might be `read_xls()` or `read_xlsx()` depending on the file extension.

2.3 Getting a sense for your data

2.3.1 First steps

What is the first thing you need to ask yourself when you look at a dataset? Data are often too big to look at the whole thing. It is almost always impossible to eyeball the entire dataset and see what you have in terms of interesting patterns or potential problems. It is often a case of information overload and we want to be able to extract what is relevant and important about it. But where do you start? Here you can find a brief but very useful overview put together by Steph de Silva. Read it before we carry on.


STARTING A DATA ANALYSIS
Rex Analytics' tips and tricks for beginning the process.
Data analysis is part science, part art. There are no one-size-fits-many solutions. But here are some questions to ask the first time you look at your data.
www.rex-analytics.com
QUESTIONS TO ASK YOUR DATA
WHERE DO YOU COME FROM?
- Where does your data come from?
- Who collected it?
- To what purpose?
- Is there associated documentation? If not, why not?
- How are you preserving this information?
WHAT IS INSIDE YOU?
- Is there a single file?
- How many variables/features?
- Does the labelling match your expectations and documentation?
WHAT ARE YOU?
- Are you dealing with integers, continuous numbers, strings of information, dates? Combinations thereof? Other?
HOW DIRTY ARE YOU?
- For each variable/feature, do the values fit your expectations?
- Do the values you observe fit the information in documentation?
- What's the missing value code? - What is the proportion of missing data?
- Are there other forms of missing or dirty data, e.g. blanks?
HOW AM I GOING TO MANAGE YOU?
- How will you keep track of changes you make?
- How will you keep track of your analyses?
ARE YOU EVERYTHING I NEED?
- What is the purpose of your project?
- Does this data represent everything you need to complete your project?
- What are the specific outcomes you are trying to achieve?
WHERE TO FROM HERE?
Congratulations! You've begun exploring your data. Data analysis is a series of questions- and these are just the start.
- Rex Analytics

As Mara Averick (somebody you want to follow in twitter at @dataanndme if you want to be in top of R related resources) suggests this also makes for good relationship advice!

Here we are going to introduce a few functions that will help you to start making sense for what you have just downloaded. Summarising the data is the first step in any analysis and it is also used for finding out potential problems with the data. Regarding the latter you want to look out for: missing values; values outside the expected range (e.g., someone aged 200 years); values that seem to be in the wrong units; mislabeled variables; or variables that seem to be the wrong class (e.g., a quantitative variable encoded as a factor).

Lets start by the basic things you always look first in a datasets. You can see in the Environment window that banbox has 14813 observations (rows) of 62 variables (columns). You can also obtain this information using code. Here you want the **DIM**ensions of the dataframe (the number of rows and columns) so you use the `dim()` function:

```
dim(banbox)
```

```
## [1] 14813    62
```

Looking at this information will help you to diagnose whether there was any trouble getting your data into R (e.g., imagine you know there should be more cases or more variables). You may also want to have a look at the names of the columns using the `names()` function. We will see the names of the variables.

```
names(banbox)
```

```
##  [1] "nj"                    "nyc"                   "app_date_d"
```

```
##  [4] "pre"                  "post"                 "storeid"
##  [7] "chain_id"             "center"               "crimbox"
## [10] "crime"                "drugcrime"            "propertycrime"
## [13] "ged"                  "empgap"               "white"
## [16] "black"                "remover"              "noncomplier_store"
## [19] "balanced"             "response"             "response_date_d"
## [22] "daystoresponse"       "interview"            "cogroup_comb"
## [25] "cogroup_njnyc"        "post_cogroup_njnyc"   "white_cogroup_njnyc"
## [28] "ged_cogroup_njnyc"    "empgap_cogroup_njnyc" "box_white"
## [31] "pre_white"            "post_white"           "white_nj"
## [34] "post_remover_ged"     "post_ged"             "remover_ged"
## [37] "post_remover_empgap"  "post_empgap"          "remover_empgap"
## [40] "post_remover_white"   "post_remover"         "remover_white"
## [43] "raerror"              "retail"               "num_stores"
## [46] "avg_salesvolume"      "avg_num_employees"    "retail_white"
## [49] "retail_post"          "retail_post_white"    "percblack"
## [52] "percwhite"            "tot_crime_rate"       "nocrimbox"
## [55] "nocrime_box"          "nocrime_pre"          "response_white"
## [58] "response_black"       "response_ged"         "response_hsd"
## [61] "response_empgap"      "response_noempgap"
```

As you may notice, these names may be hard to interpret. If you open the dataset in the data viewer of RStudio (using `View`) you will see that each column has a variable name and underneath a longer and more meaningful **variable label** that tells you what each variable means.

```
View(banbox)
```

2.3.2 On tibbles and labelled vectors

You also want to understand what the banbox object actually is. You can do that using the `class()` function:

```
class(banbox)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

What does `tbl` stands for? It refers to **tibbles**. This is essentially a new type of data structure introduced into R. Tibbles *are* data frames, but a particular type. Not all data frames we encounter in R are tibbles.

The R language has been around for a while and sometimes things that made sense a couple of decades ago, make less sense now. A number of programmers are trying to create code that is more modern and more useful today. They

are doing this by introducing a set of packages that speak to each other in order to modernise R without breaking existing code. You can think of it as an easier and more efficient modern dialect of R. This set of packages is called the **tidyverse**. Tibbles are dataframes that have been optimised to be used with this new set of packages. You can read a bit more about tibbles here.

You can also look at the class of each individual column. As discussed, class of the variable lets us know, for example, if its an integer (number), character, or factor.

To get the class of one variable, you pass it to the `class()` function. For example:

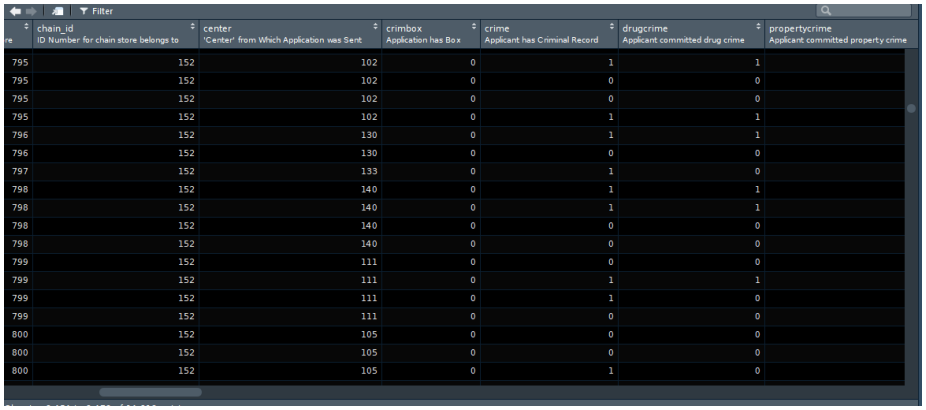
```
class(banbox$crime)
```

```
## [1] "haven_labelled" "vctrs_vctr"     "double"
```

```
class(banbox$num_stores)
```

```
## [1] "numeric"
```

We talked about numeric vectors in week one. It is simply a collection of numbers. But what is a labelled vector? This is a new type of vector introduced by the *haven* package. **Labelled vectors** are categorical variables that have labels. Go to the *Environment* panel and left click in the banbox object. This should open the data browser in the top left quadrant of RStudio.



If you look carefully you will see that the various columns that include categorical variables only contain numbers. In many statistical environments, such as STATA or SPSS, this is a common standard. The variables have a numeric value for each observation and then each of these numeric values is associated with a label. This kind of made sense when computer memory was an issue - for

this was an efficient way of saving resources. It also made manual data input quicker. These days it makes perhaps less sense. But labelled vectors give you a chance to reproduce data from other statistical environments without losing any fidelity in the import process. See what happens if we try to summarise this labelled vector. We will use the `table()` to provide a count of observations on the various valid values of the *crime* variable. It is a function to obtain your frequency distribution.

```
table(banbox$crime)
```

```
##
##    0    1
## 7323 7490
```

So, we see that we have 7490 observations classed as 1 and 7323 classed as 2. If only we knew what those numbers represent! Well, we actually do. We will use the `attributes()` function to see the different "compartments"" within your "box", your object.

```
attributes(banbox$crime)
```

```
## $label
## [1] "Applicant has Criminal Record"
##
## $format.stata
## [1] "%9.0g"
##
## $class
## [1] "haven_labelled" "vctrs_vctr"     "double"
##
## $labels
## No Crime    Crime
##        0        1
```

So this object has different compartments. The first one is called label and it provides a description of what the variable measures. This is what you saw in the RStudio data viewer earlier. The second compartment explains the original format in which it was. The third one identifies the class of the vector. Whereas the final one, *labels*, provides the labels that allows us to identify what the meaning of 0 and 1 mean in this context.

2.3.3 Turning variables into factors and changing the labels

Last week we said that many R functions expect factors when you have categorical data, so typically after you import data into R you may want to coerce your labelled vectors into factors. To do that you need to use the `as_factor()` function of the *haven* package. Let's see how we do that.

```
#This code asks R to create a new column in your banbox tibble
#that is going to be called crime_f. Typically, when you alter
#variables you can to create a new one so that the original gets
#preserved in case you do something wrong. Then we use the
#as_factor() function to explain to R that what we want to do
#is to get the original crime variable and mutate it into
#a factor, this resulting factor is what will be stored in
#the new column.
banbox$crime_f <- as_factor(banbox$crime)
```

You will see now that you have 63 variables in your dataset, look at the environment to check. Let's explore the new variable we have created (you can also look for the new variable in the data browser and see how it looks different to the original crime variable):

```
class(banbox$crime_f)
```

```
## [1] "factor"
```

```
table(banbox$crime_f)
```

```
##
## No Crime    Crime
##     7323     7490
```

```
attributes(banbox$crime_f)
```

```
## $levels
## [1] "No Crime" "Crime"
##
## $class
## [1] "factor"
##
## $label
## [1] "Applicant has Criminal Record"
```

So far we have a looked at single columns in your dataframe one at the time. But there is a way that you can apply a function to all elements of a vector (list or dataframe). You can use the functions `sapply()`, `lapply()`, and `mapply()` . To find out more about when to use each one see here.

For example, we can use the `lapply()` function to look at each column and get its class. To do so, we have to pass two arguments to the `lapply()` function, the first is the name of the dataframe, to tell it what to look through, and the second is the function we want it to apply to every column of that function.

So we want to type "lapply(" + "name of dataframe" + "," + "name of function" + ")"

Which is:

```
lapply(banbox, class)
```

As you can see many variables are classed as labelled. This is common with survey data. Many of the questions in social surveys measure the answers as categorical variables (e.g., these are nominal or ordinal level measures). In fact, with this dataset there are many variables that are encoded as numeric that really aren't. Welcome to real world data, where things can be a bit messy and need tidying!

See for example the variable black:

```
class(banbox$black)
```

```
## [1] "numeric"
```

```
table(banbox$black)
```

```
##
##    0    1
## 7406 7407
```

We know that this variable measures whether someone is black or not. When people use 0 and 1 to code binary responses, typically they use a 1 to denote a positive response, a yes. So, I think it is fair to assume that a 1 here means the respondent is black. Because this variable is of class numeric we cannot simply use as_factor to assign the pre-existing labels and create a new factor. In this case we don't have preexisting labels, since this is not a labelled vector. So what can we do to tidy this variable? We'll we need to do some further work.

```
#We will use a slightly different function as.factor()
banbox$black_f <- as.factor(banbox$black)
#You can check that the resulting column is a factor
class(banbox$black_f)
```

```
## [1] "factor"
```

```
#But if you print the frequency distribution you will see the
#data are still presented in relation to 0 and 1
table(banbox$black_f)
```

```
##
##    0    1
## 7406 7407
```

```
#You can use the levels function to see the levels, the
#categories, in your factor
levels(banbox$black_f)
```

```
## [1] "0" "1"
```

So, all we have done is create a new column that is a factor but still refers to 0 and 1. If we assume (rightly) that 1 mean black we have 7407 black applicants. Of course, it makes sense we only get 0 and 1 here. What else could R do? This is not a labelled vector, so there is no way for R to know that 0 and 1 mean anything other than 0 and 1, which is why those are the levels is using. But now that we have the factor we can rename those levels. We can use the following code to do just that:

```
#We are using the levels function to access them and change
#them to the levels we specify with the c() function. Be
#careful here, because the order we specify here will map
#out to the order of the existing levels. So given that 1 is
#black and black is the second level (as shown when printing
#the results above) you want to make sure that in the c()
#you write black as the second level.
levels(banbox$black_f) <- c("White", "Black")
table(banbox$black_f)
```

```
##
## White Black
##  7406  7407
```

This gives you an idea of the kind of transformations you often want to perform to make your data more useful for your purposes. But let's keep looking at functions you can use to explore your dataset.

2.3.4 Looking for missing data and other anomalies

You can, for example, use the `head()` function if you just want to visualise the values for the first few cases in your dataset. The next code for example ask for the values for the first two cases. If you want a different number to be shown you just need to change the number you are passing as an argument.

```
head(banbox, 2)
```

In the same way you could look at the last two cases in your dataset using `tail()`:

```
tail(banbox, 2)
```

It is good practice to do this to ensure R has read the data correctly and there's nothing terribly wrong with your dataset. If you have access to STATA you can open the original file in STATA and check if there are any discrepancies, for example. Glimpsing your data in this way can also give you a first impression for what the data looks like.

One thing you may also want to do is to see if there are any **missing values**. For that we can use the `is.na()` function. Missing values in R are coded as NA. The code below, for example, asks for NA values for the variable *response_black* in the *banbox* object for observations 1 to 10:

```
is.na(banbox$response_black[1:10])
```

```
##  [1] FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE
```

The result is a logical vector that tells us if it is true that there is missing (NA) data for each of those first ten observations. You can see that there are 6 observations out of those 10 that have missing values for this variable.

```
sum(is.na(banbox$response_black))
```

```
## [1] 7406
```

This is asking R to sum how many cases are TRUE NA in this variable. When reading a logical vector as the one we are creating, R will treat the FALSE elements as 0s and the TRUE elements as 1s. So basically the sum() function will count the number of TRUE cases returned by the is.na() function.

You can use a bit of a hack to get the proportion of missing cases instead of the count:

```
mean(is.na(banbox$response_black))
```

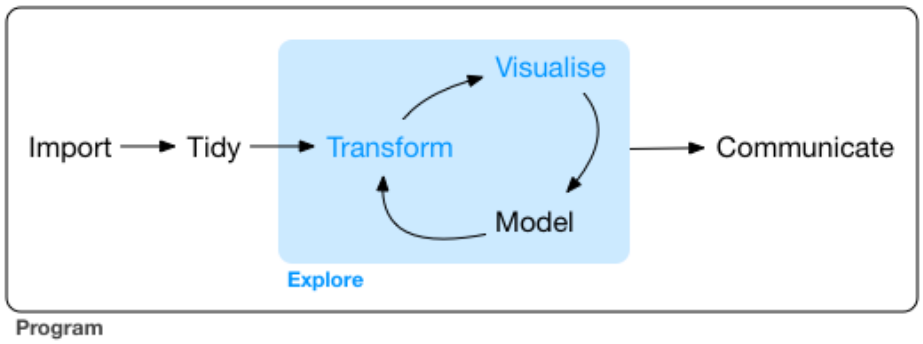
```
## [1] 0.4999662
```

This code is exploiting the mathematical fact that the mean of binary outcomes (0 or 1) gives you the proportion of ones in your data. As a rule of thumb, if you see more than 5% of the cases declared as NA, you need to start thinking about the implications of this. Beware of formulaic application of rules of thumb such as this though! In this case, we know that 49%% of the observations have missing values in this variable. When you see things like this the first thing to do is to look at the codebook or documentation to try to get some clues as to why there are so many missing cases. With survey data you often have questions that are simply not asked to everybody, so it's not necessarily that something went very wrong with the data collection, but simply that the variable in question was only used with a subset of the sample. And that, therefore, any analysis you do using this question will only relate to that particular subset of cases.

There is a whole field of statistics devoted to doing analysis when missing data is a problem. R has extensive capabilities for dealing with missing data -see for example here. For the purpose of this introductory course, however, we only explain how to do analysis that ignore missing data. This is often referred to a **full/complete case analysis**, because you only use observations for which you have full information in all the variables you employ. You would cover techniques for dealing with this sort of issues in more advanced courses.

2.4 Data wrangling with dplyr

The data analysis workflow has a number of stages. The diagram below (produced by Hadley Wickham) is a nice illustration of this process:



We have started to see different ways of bringing data into R. And we have also started to see how we can explore our data. It is now time we start discussing one of the following stages, **transform**. A good deal of time and effort in data

analysis is devoted to this. You get your data and then you have to do some transformations to it so that you can answer the questions you want to address in your research. We have already seen, for example, how to turn variables into factors, but there are other things you may want to do.

R offers a great deal of flexibility in how to transform your data, Here we are going to illustrate some of the functionality of the *dplyr* package for data carpentry (a term people use to refer to this kind of operations). This package is part of the tydiverse and it aims to provide a friendly and modern take on how to work with dataframes (or tibbles) in R. It offers, as the authors of the package put it, "a flexible grammar of data manipulation".

Dplyr aims to provide a function for each basic verbs of data manipulation:

- `*filter()*` to select cases based on their values.
- `*arrange()*` to reorder the cases.
- `*select()*` and `*rename()*` to select variables based on their names.
- `*mutate()*` and `*transmute()*` to add new variables that are functions of existing variables.
- `*summarise()*` to condense multiple values to a single value.
- `*sample_n()*` and `*sample_frac()*` to take random samples.

In this session we will introduce and practice some of these. But we won't have time to cover everything. There is, however, a very nice set of vignettes for this package in the help files, so you can try to go through those if you want a greater degree of detail or more practice.

Now let's load the package:

```
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
```

Notice that when you run this package you get a series of warnings in the conseole. It is telling us that some functions from certain packages are being "masked". One the things with a language like R is that sometimes packages

introduce functions that have the same name than others that are already loaded into your session. When that happens the newly loaded ones will over-ride the previous ones. You can still use them but you will have to refer to them explicitly. Otherwise R will assume you are using the function most recently loaded:

```
#Example:
#If you use load dplyr and then invoke the *filter()* function
#R will assume you are using the filter function from dplyr
#rather than the *filter()* function that exist in the *stats*
#package, which is part of the basic installation of R. If
#after loading dplyr you want to use the filter function from
#the stats package you will have to invoke it like this:
stats::filter()
#Notice the grammar, first you write the name of the package,
#then colon twice, and then the name of the function. Don't
#run this code. You would need to pass some valid arguments
#for this to produce meaningful results.
```

2.5 Using dplyr single verbs

One of the first operations you may want to carry out when working with dataframes is to subset them based on values of particular variables. Say we want to replicate the results reported by Agan and Starr in 2017. In this earlier paper, these researchers only used data from the period prior to the introduction of Ban the Box legislation and only used data from businesses that asked about criminal records in their online applications. How can we recreate this dataset?

For this kind of operations we use the `filter()` function. Like all single verbs in dplyr, the first argument is the tibble (or data frame). The second and subsequent arguments refer to variables within that data frame, selecting rows where the expression is TRUE.

Ok, so if we look at the dataset we can see that there is a variable called "crimbox" that identifies applications that require information about criminal antecedents and there is a variable called "pre" that identifies whether the application was sent before the legislation was introduced. In this dataset the value 1 is being used to denote positive responses. So, if we want to create the 2017 dataset we would start by selecting only data where the value in these two variables equals 1 as shown below.

```
#We will store the results of filtering the data in a new
#object that I am calling aer (short for the name of the
#journal in which the paper was published)
```

```
aer2017<- filter(banbox, crimbox == 1, pre == 1)
```

Notice that the number of cases equals the number of cases reported by the authors in their 2017 paper. That's cool! So far we are replicating with same results.

You may have noticed in the code above that I wrote "==" instead of "=". Logical operators in R are not written exactly the same way than in normal practice. Keep this in mind when you get error messages from running your code. Often the source of your error may be that you are writing the logical operators the wrong way (as far as R is concerned). Look here for valid logic operators in R.

Sometimes you may want to select only a few variables. Earlier we said that real life data may have hundreds of variables and only a few of those may be relevant for your analysis. Say you only want "crime", "ged" (a ged is a high school equivalence diploma rather than a proper high school diploma and is sometimes seen as inferior), "empgap" (a gap year on employment), "black_f", "response", and "daystoresponse" from this dataset. For this kind of operations you use the `select()` function.

The syntax of this function is easy. First we name the dataframe object ("aer2017") and then we list the variables. The order in which we list the variables within the select function will determine the order in which those columns appear in the new dataframe we are creating. So this is a handy function to use if you want to change the order of your columns for some reason. Since I am pretty confident I am not making a mistake I will transform the original "aer2017" tibble rather than creating an entirely new object.

```
aer2017 <- select(aer2017, crime, ged, empgap, black_f, response, daystoresponse)
```

If you now look at the global environment you will see that the "aer2017" tibble has reduced in size and now only has 6 columns. If you view the data you will see these are the 6 variables we selected.

2.6 Using dplyr for grouped operations

So far we have used `dplyr` single verbs for ungrouped operations. But we can also use some of the functionality of `dplyr` for obtaining answers to questions that relate to groups of cases within our dataframe. Imagine that you want to know if applicants with a criminal record are less likely to receive a positive response from employers. How could you figure that one out? For answering this kind of questions we can use the `group_by()` function in conjunction with

other `dplyr` functions. In particular we are going to look at the `summarise` function.

First we group the observations by criminal record in a new object, by using as_factor in the call to the crime variable the results will be labelled later on (even though we are not changing the crime variable in the aer2017 dataframe. Keep in mind we are using as_factor because the column crime is a labelled vector rather than a factor or a character vector, and we do this to aid interpretation (it is easier to interpret labels than 0 and 1).

```
by_antecedents <- group_by(aer2017, as_factor(crime))

#Then we run the summarise function to provide some useful
#summaries of the groups we are using: the number of cases
#and the mean of the response variable
results <- summarise(by_antecedents,
  count = n(),
  outcome = mean(response, na.rm = TRUE))
#autoprint the results stored in the newly created object
results
```

```
## # A tibble: 2 x 3
##   `as_factor(crime)` count outcome
##   <fct>              <int>   <dbl>
## 1 No Crime            1319  0.136
## 2 Crime               1336  0.0846
```

Let's look at the code in the summarise function above. First we are asking R to place the results in an object we are calling "results". Then we are specifying that we want to group the data in the way we specified in our `group_by()` function before, that is by criminal record. Then we pass two arguments. Each of these arguments is creating a new variable in the resulting object called "results". The first variable we are creating is called "*count*" by saying this equals "n()" we are specifying to R that this new variable simply counts the number of cases in each of the grouping categories. The second variable we are creating is called "*outcome*" and to compute this variable we are asking R to compute the mean of the variable response for each of the two groups of applicants defined in "by_antecedents" (those with records, those without). Remember that the variable response in the "aer2017" dataframe was coded as numeric variable, even though in truth is categorical in nature (there was a response, or not, from the employers). It doesn't really matter. Taking the mean of a binary variable in this case is mathematically equivalent to computing a proportion as we discussed earlier.

So, what we see here is that about 13.6% of applicants with no criminal record received a positive response from the employers, whereas only 8% of those with

criminal records did receive such a response. Given that the assignation of a criminal record was randomised to the applicants, there's a pretty good chance that no other **confounders** are influencing this outcome. And that is the beauty of randomised experiments. You may be in a position to make stronger claims about your results.

2.7 Making comparisons with numerical outcomes

We have been looking at relationships so far between categorical variables, specifically between having a criminal record (yes or no), race (black or white), and receiving a positive response from employers (yes or no). Often we may be interested in looking at the impact of a factor on a numerical outcome. In the banbox object we have such an outcome measured by the researchers. The variable "daystoresponse" tells us how long it took the employers to provide a positive response. Let's look at this variable:

```
summary(banbox$daystoresponse)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##    0.00    3.00   10.00   19.48   28.00  153.00   14361
```

The `summary()` function provides some useful stats for numerical variables. We obtain the minimum and maximum value, the 25th percentile, the median, the mean, the 75th percentile, and the number of missing data (NA). You can see the number of missing data here is massive. Most cases have missing data on this variable. Clearly this is a function of, first and foremost, the fact that the number of days to receive a positive response will only be collected in cases where there was a positive response! But even accounting for that, it is clear that this information is also missing in many cases that received a positive response. So given all of this, we need to be very careful when interpreting this variable. Yet, because it is the only numeric variable here we will use it to illustrate some handy functions.

We could do as before and get results by groups. Let's look at the impact of race on days to response:

```
by_race <- group_by(banbox, black_f)
results_3 <- summarise(by_race,
  avg_delay = mean(daystoresponse, na.rm = TRUE))
results_3
```

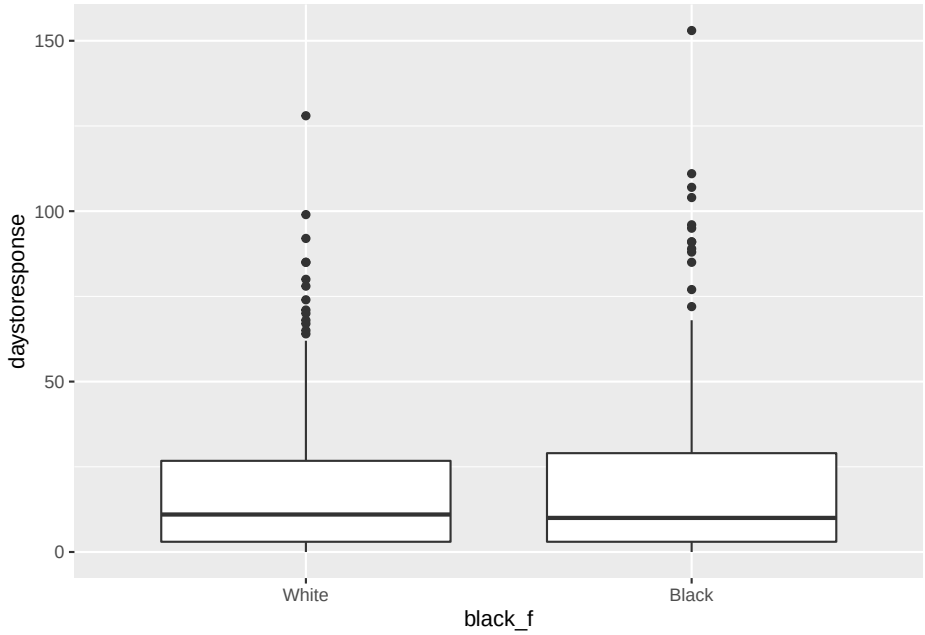
```
## # A tibble: 2 x 2
```

```
##   black_f avg_delay
##   <fct>       <dbl>
## 1 White        18.7
## 2 Black        20.4
```

We can see that the average delay seems to be longer for Black applicants than White applicants.

But we could also try to represent these differences graphically. The problem with comparing groups on quantitative variables using numerical summaries such as the mean, is that these comparisons hide more than they show. We want to see the full distribution, not just the mean. For this we are going to use `ggplot2` the main graphical package we will use this semester. We won't get into the details of this package or what the code below means, but just try to run it. We will cover graphics in R in the next section. This is just a taster for it.

```
library(ggplot2)
ggplot(banbox, aes(y = daystoresponse, x = black_f)) +
  geom_boxplot()
```



Watch this video and see if you can interpret the results portrayed here. What do you think?

Overall the outcomes are worse for Black applicatns, and in fact, the authors find that employers substantially increase discrimination on the basis of race

after ban the box goes into effect.You can now replicate these findings with the data provided. by applying the new skills you've learned this week!

Chapter 3

Data visualisation with R

3.1 Introduction

A picture is worth a thousand words; when presenting and interpreting data this basic idea also applies. There has been, indeed, a growing shift in data analysis toward more visual approaches to both interpretation and dissemination of numerical analysis. Part of the new data revolution consists in the mixing of ideas from visualisation of statistical analysis and visual design. Indeed data visualisation is one of the most interesting areas of development in the field.

Good graphics not only help researchers to make their data easier to understand by the general public. They are also a useful way for understanding the data ourselves. In many ways it is very often a more intuitive way to understand patterns in our data than trying to look at numerical results presented in a tabular form.

Recent research has revealed that papers which have good graphics are perceived as overall more clear and more interesting, and their authors perceived as smarter (see this presentation)

The preparation for this session includes many great resources on visualising quantitative information, and if you have not had time to go through them, we recommend that you take some time to do so.

As with other aspects of R, there are a number of core functions that can be used to produced graphics. However these offer limited possibilities for building graphs.

The package we will be using throughout this tutorial is `ggplot2`. The aim of `ggplot2` is to implement the grammar of graphics. The `ggplot2` package has excellent online documentation and is becoming an industry standard in some sectors. Here for example you can read about how the BBC uses as part of their News service.

If you don't already have the package installed (check you do), you will need to do so using the `install.packages()` function.

You will then need to load up the package

```
library(ggplot2)
```

The grammar of graphics upon which this package is based on defines various components of a graphic. Some of the most important are:

-**The data**: For using `ggplot2` the data has to be stored as a data frame or tibble.

-**The geoms**: They describe the objects that represent the data (e.g., points, lines, polygons, etc..). This is what gets drawn. And you can have various different types layered over each other in the same visualisation.

-**The aesthetics**: They describe the visual characteristics that represent data (e.g., position, size, colour, shape, transparency).

-**Facets**: They describe how data is split into subsets and displayed as multiple small graphs.

-**Stats**: They describe statistical transformations that typically summarise data.

Let's take it one step at the time.

3.2 Anatomy of a plot

Essentially the philosophy behind this is that all graphics are made up of layers. The package `ggplot2` is based on the grammar of graphics, the idea that you can build every graph from the same few components: a data set, a set of geoms—visual marks that represent data points, and a coordinate system.

Take this example (all taken from *Wickham, H. (2010). A layered grammar of graphics. Journal of Computational and Graphical Statistics, 19(1), 3-28.*)

You have a table such as:

Table 3. Simple dataset with variables mapped into aesthetic space.

x	y	Shape
25	11	circle
0	0	circle
75	53	square
200	300	square

You then want to plot this. To do so, you want to create a plot that combines the following layers:

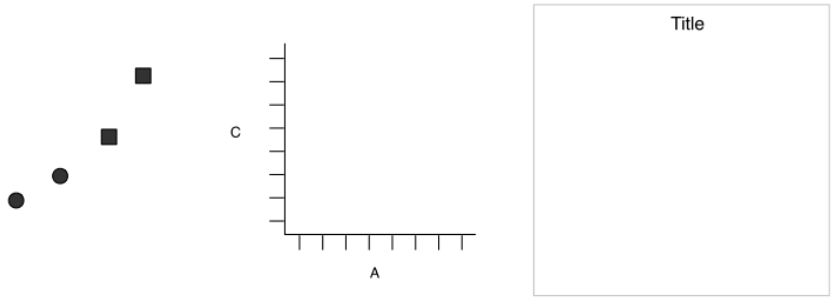


Figure 1. Graphics objects produced by (from left to right): geometric objects, scales and coordinate system, plot annotations.

This will result in a final plot:

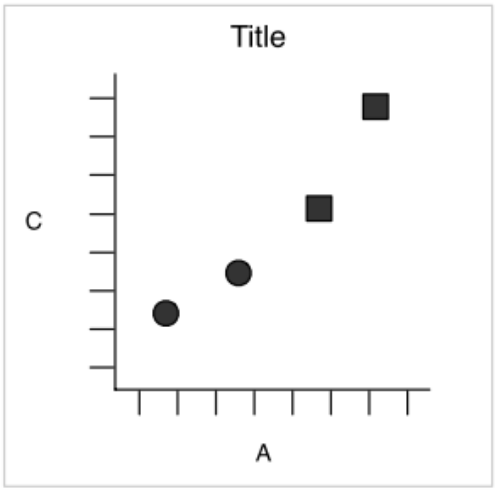


Figure 2. The final graphic, produced by combining the pieces in Figure 1.

Let's have a look at what this looks like for a graph.

Let's have a look at some data about banning orders for different football clubs.

First you need to read the data. We keep this data in a website and you can download it with the following code:

```
# save URL into an object
fbo_url <- "https://raw.githubusercontent.com/maczokni/modelling_book/master/datasets/l

# load readr library and use read_csv() function
library(readr)
fbo <- read_csv(url(fbo_url))
```

```
## New names:
## * `` -> ...1

## Rows: 119 Columns: 4

## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): Club.Supported, League.of.the.Club.Supported
## dbl (2): ...1, Banning.Orders

##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

You can also find this on the Blackboard page for this week's learning materials. If you download from there, make sure to save this file in your project directory, possibly in a subfolder called "Datasets". Then you can read in from there.

One thing from the first lab we mentioned is conventions in the naming ob objects. This also applies to the names of your variables (i.e. your column names) within your data. If you look at the fbo dataframe, either with the `View()` function, or by printing the names of the columns with the `names()` function, you can see, this dataset violates that requirement:

```
names(fbo)
```

```
## [1] "...1"                          "Club.Supported"
## [3] "Banning.Orders"                "League.of.the.Club.Supported"
```

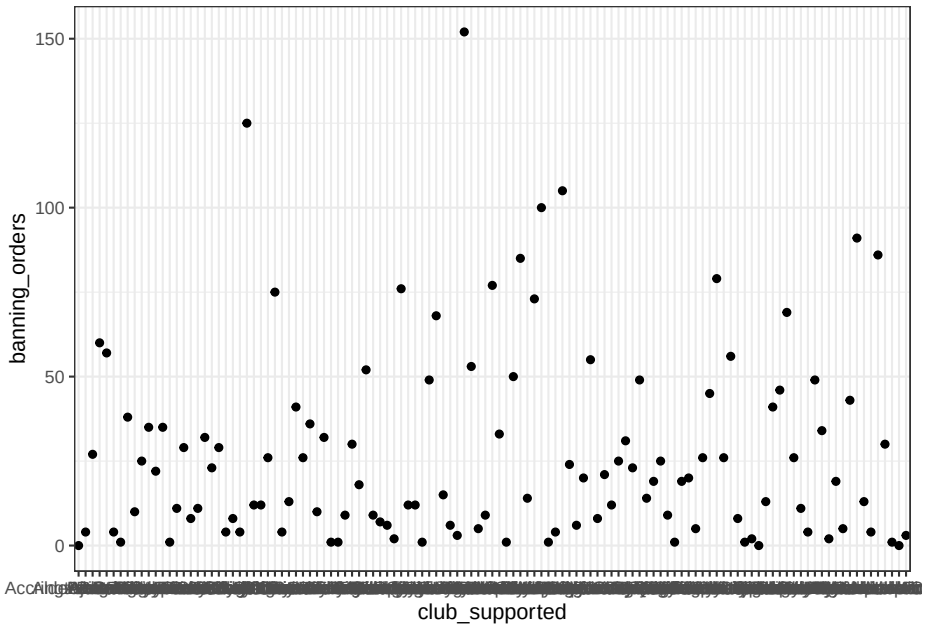
To address this, we can use a function called `clean_names()` which lives inside the `janitor` package. This will replace any spaces with an underscore, and also turn any capital letters into lowercase. Much more tidy!

```
library(janitor)

fbo <- clean_names(fbo)
```

Now let's explore the question of number of banning orders for clubs in different leagues. But as a first step, let's just plot the number of banning orders for each club. Let's build this plot:

```
ggplot(data = fbo, aes(x = club_supported, y=banning_orders)) + #data
   geom_point() +                  #geometry
  theme_bw()                      #backgroud coordinate system
```

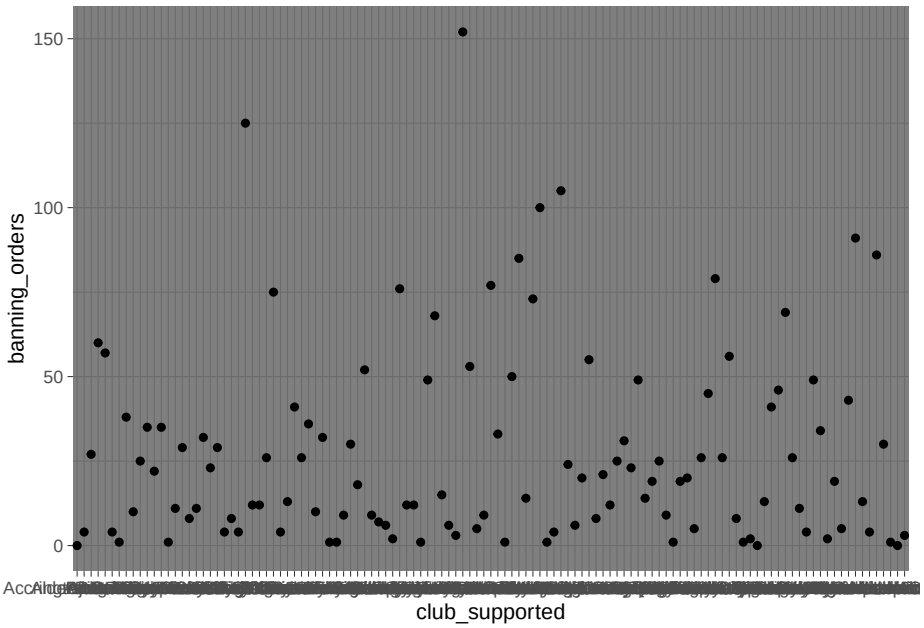


The first line above begins a plot by calling the `ggplot()` function, and putting the data into it. You have to name your dataframe witg the `data` argument, and then, within the `aes()` command you pass the specific variables which you want to plot. In this case, we only want to see the distribution of one variable, banning orders, in the y axis and we will plot the club supported in the x axis.

The second line is where we add the *geometry*. This is where we tell R what we want the graph to be. Here we say we want it to be points by using `geom_points`. You can see a list of all possible geoms here.

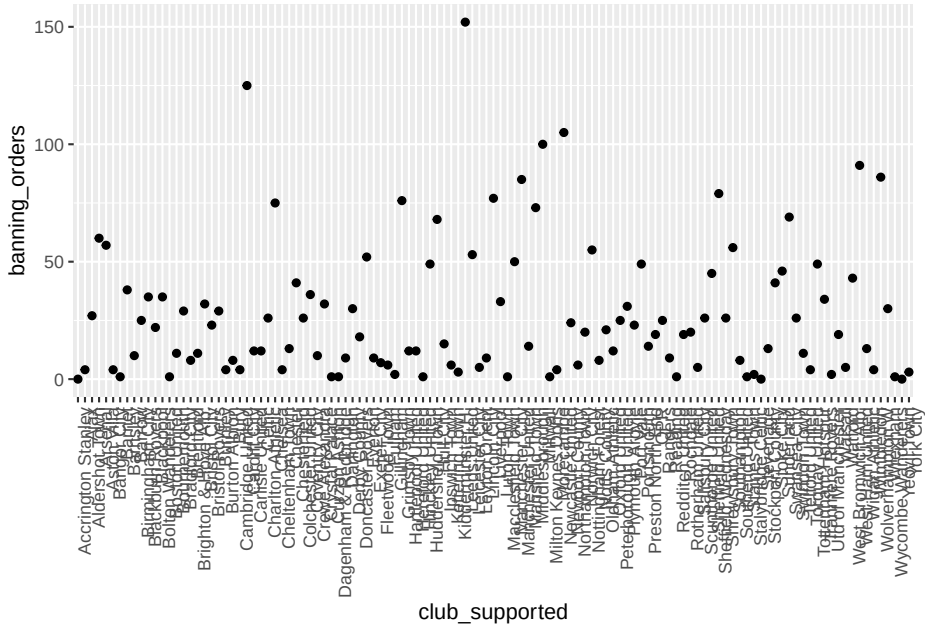
The third line is where we can tweak the display of the graph. Here I used `theme_bw()` which is a nice clean theme. You can try with other themes. To get a list of themes you can also see the resource here. If you want more variety you can explore the package `ggthemes`.

```
ggplot(data = fbo, aes(x = club_supported, y=banning_orders)) +     #data
   geom_point() +                                                   #geometry
  theme_dark()                                                      #backgroud coordin
```



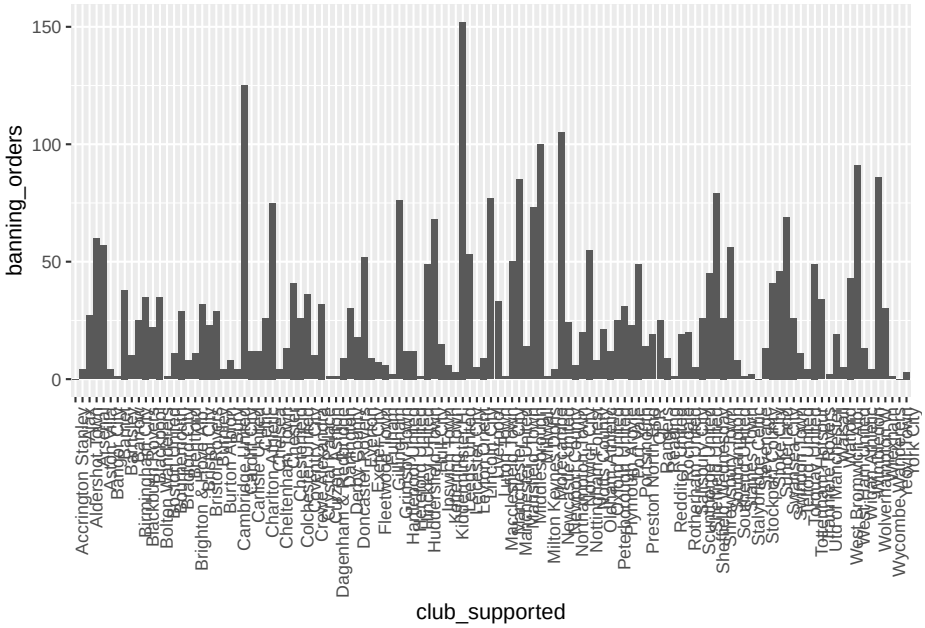
Changing the theme is not all you can do with the third element. For example here you can't really read the axis labels, because they're all overlapping. One solution would be to rotate your axis labels 90 degrees, with the following code: `axis.text.x = element_text(angle = 90, hjust = 1)`. You pass this code to the `theme` argument.

```
ggplot(data = fbo, aes(x = club_supported, y=banning_orders)) +
   geom_point() +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```



OK what if we don't want it to be points, but instead we wanted it to be a bar graph?

```
ggplot(data = fbo, aes(x = club_supported, y=banning_orders)) +   #data
   geom_bar(stat = "identity") +                                  #geometry
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```

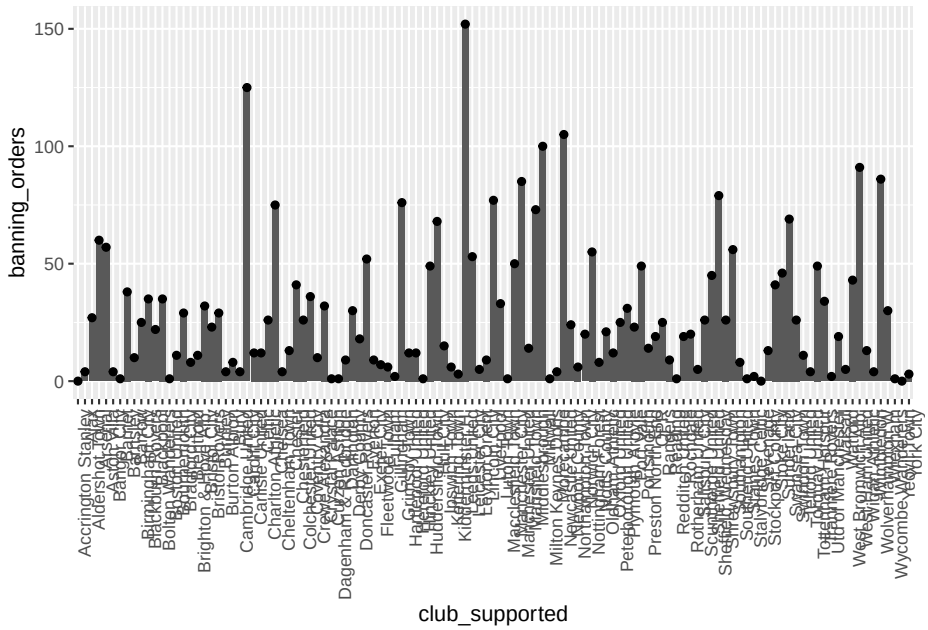


You might notice here we pass an argument `stat = "identity"` to `geo_bar()` function. This is because you can have a bar graph where the height of the bar shows frequency (`stat = "count"`), or where the height is taken from a variable in your dataframe (`stat = "identity"`). Here we specified a y-value (height) as the banning_orders variable.

So this is cool! But what if I like both?

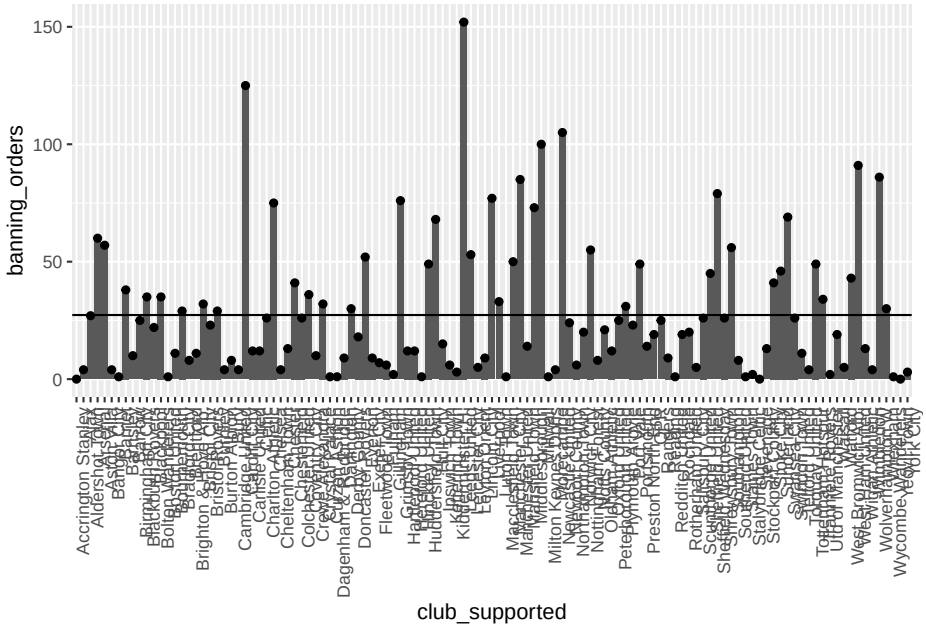
Well this is the beauty of the layering approach of `ggplot2`. You can layer on as many geoms as your little heart desires! XD

```
ggplot(data = fbo, aes(x = club_supported, y=banning_orders)) +  #data
  geom_bar(stat = "identity") +                                  #geometry 1
  geom_point()+                                                  #geometry 2
theme(axis.text.x = element_text(angle = 90, hjust = 1))
```



You can add other things too. For example you can add the mean number of *banning_orders*:

```
ggplot(data = fbo, aes(x = club_supported, y=banning_orders)) +  #data
  geom_bar(stat = "identity") +                                  #geometry 1
  geom_point() +                                                 #geometry 2
  geom_hline(yintercept = mean(fbo$banning_orders)) +            #mean line
theme(axis.text.x = element_text(angle = 90, hjust = 1))
```

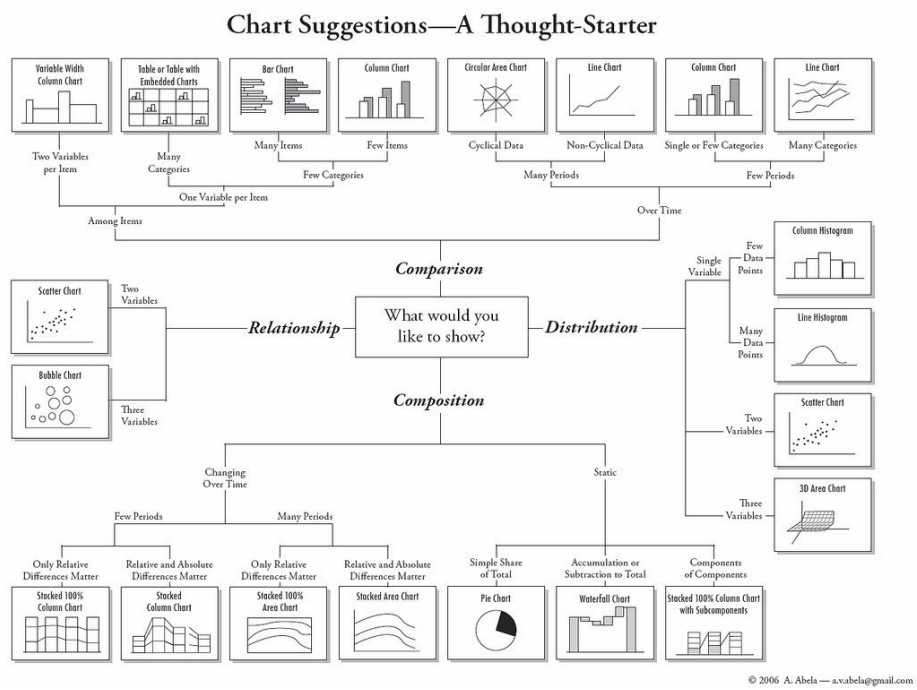


This is basically all you need to know to build a graph! So far we have introduced a lot of code some of which you may not fully understand. Don't worry too much, we just wanted to give you a quick introduction to some of the possibilities. Later in the session we will go back to some of these functions in a slower way. The take away from this section is to understand the basic elements of the grammar of graphics.

3.3 What graph should I use?

There are a lot of points to consider when you are choosing what graph to use to visually represent your data. There are some best practice guidelines, but at the end of the day, you need to consider what is best for your data. What do you want to show? What graph will best communicate your message? Is it a comparison between groups? Is it the frequency distribution of 1 variable?

As some guidance, you can use the below cheatsheet, taken from Nathan Yau's blog Flowingdata:



However, keep in mind that this is more of a guideline, aimed to nudge you in the right direction. There are many ways to visualise the same data, and sometimes you might want to experiment with some of these, see what the differences are.

There is also a vast amount of research into what works in displaying quantitative information. The classic book is by Edward Tufte [1], but since him there are many other researchers as well who focus on approaches to displaying data. Two useful books you may want to consider are Few (2012) [2] and Cairo (2016)

[1] Although we would like to think of our samples as random, it is in fact very difficult to generate random numbers in a computer. Most of the time someone is telling you they are using random numbers they are most likely using pseudo-random numbers. If this is the kind of thing that gets you excited you may want to read the wiki entry. If you want to know how R generates these numbers you should ask for the help pages for the Random.Seed function.

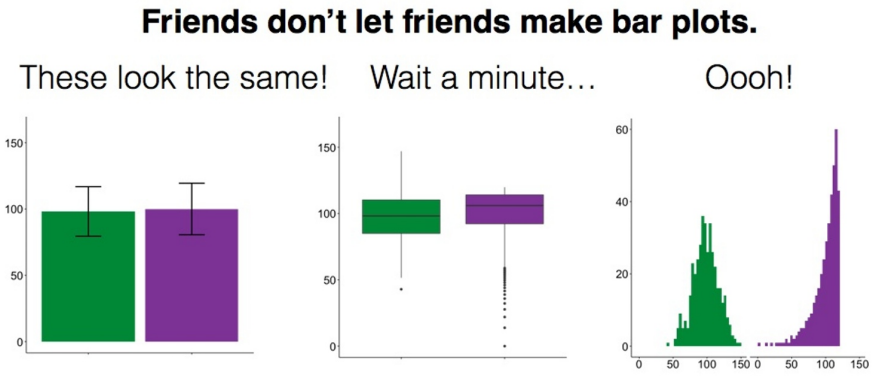
[2] As an aside, you can use this Java applet to see what happens when one uses different parameters with confidence intervals. In the right hand side you will see a button that says "Sample". Press there. This will produce a horizontal line representing the confidence interval. The left hand side end of the line represents the lower limit of the confidence interval and the right hand side end of the line represents the upper limit of the confidence interval. The red point in the middle is your sample mean, your point estimate. If you are lucky the line will be black and it will cross the green vertical line. This means that your CI covers the population mean. There will be a difference with your point estimate (i.e., your red point is unlikely to be just in top of the green vertical line). But, at least, the population parameter will be included within the range of plausible values for it that our confidence interval is estimating. If you keep pressing the "Sample" button (please do 30 or 50 times), you will see that most confidence intervals include the population parameter: most will cross the green line and will be represented by a black line. Sometimes your point estimate (the red point at the centre of the horizontal lines) will be to the right of the population mean (will be higher) or to the left

[3]. Claus Wilke is also producing a textbook freely available in the internet. These authors tend to produce recommendations on what to use (and not use) in certain contexts.

For example, most data visualisation experts agree that you should not use 3D graphics unless there is a meaning to the third dimension. So using 3D graphics just for decoration, as in this case is normally frowned upon. However there are cases when including a third dimension is vital to communicating your findings. See this example.

Also often certain chart types are vilified. For example, the *pie chart* is one such example. A lot of people (including your course leader) really dislike pie charts, e.g. see here or here. If you want to display proportion, research indicates that a square pie chart is more likely to be interpreted correctly by viewers see here.

Also, in some cases bar plots (if used to visualise quantitative variables) can hide important features of your data, and might not be the most appropriate means for comparison:



This has lead to a kickstarter campaign around actually banning bar plots…!

So choosing the right plot and how to design the different elements of a plot is somehow of an art that requires practice and a good understanding of the data visualisation literature. Here we can only provide you with an introduction to some of these issues. At the end of the chapter we will also highlight additional resources you may want to explore on your own.

An important consideration is that the plot that you use depends on the data

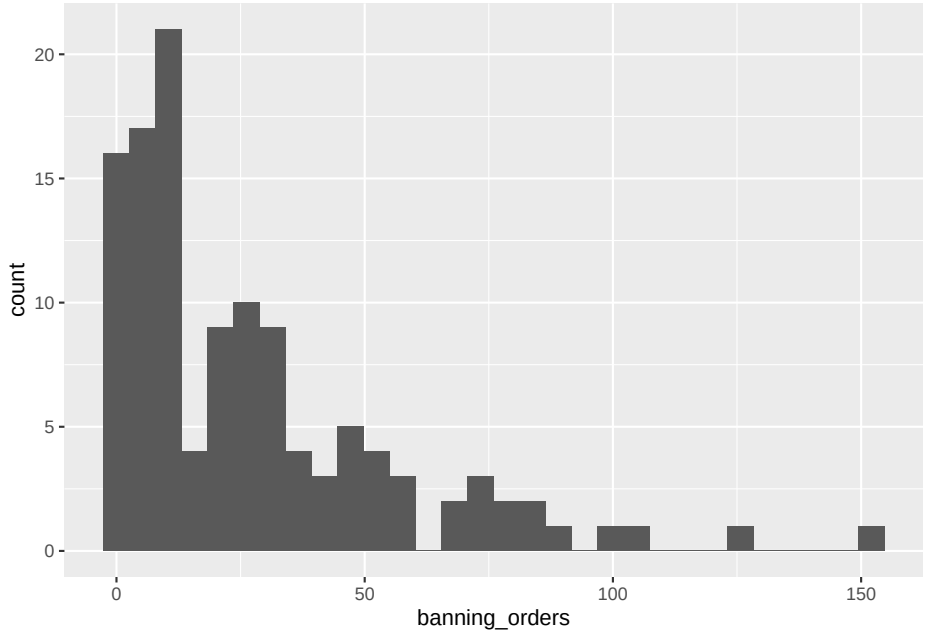
(will be lower), but the confidence interval will include the population mean (and cross the green vertical line).

[3]Cairo, Alberto (2016) *The truthful art: data, charts, and maps for communication.* New Riders.

you are plotting, as well as the message you want to convey with the plot, the audience that is intended for, and even the format in which it will be presented (a website, a printed report, a power point presentation, etc.). So for example, returning again to the difference between number of banning orders between clubs in different leagues, what are some ways of plotting these?

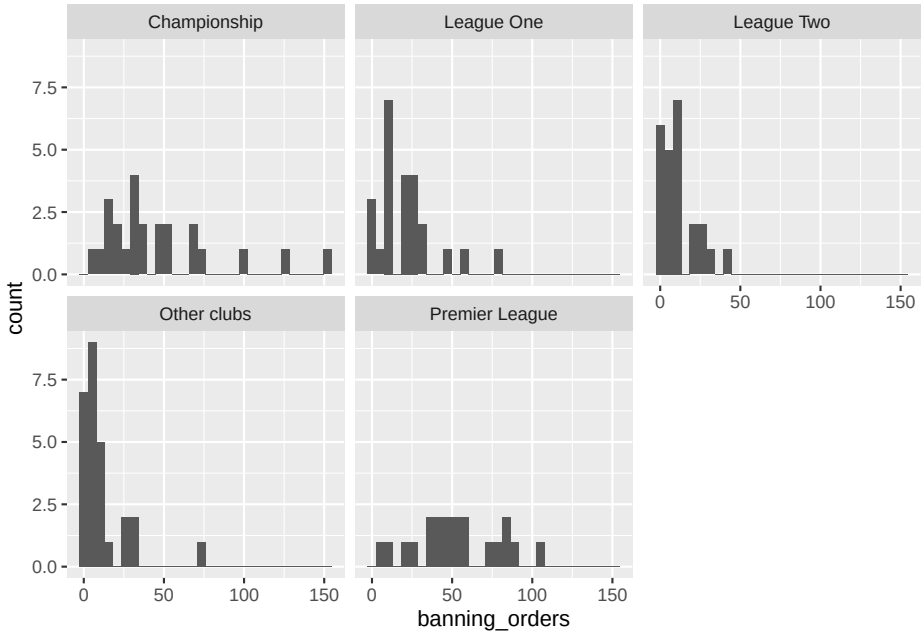
One suggestion is to make a histogram for each one. You can use ggplot's `facet_wrap()` option to split graphs by a grouping variable. For example, to create a histogram of banning orders you write:

```
ggplot(data = fbo, aes(x = banning_orders)) +
  geom_histogram()
```



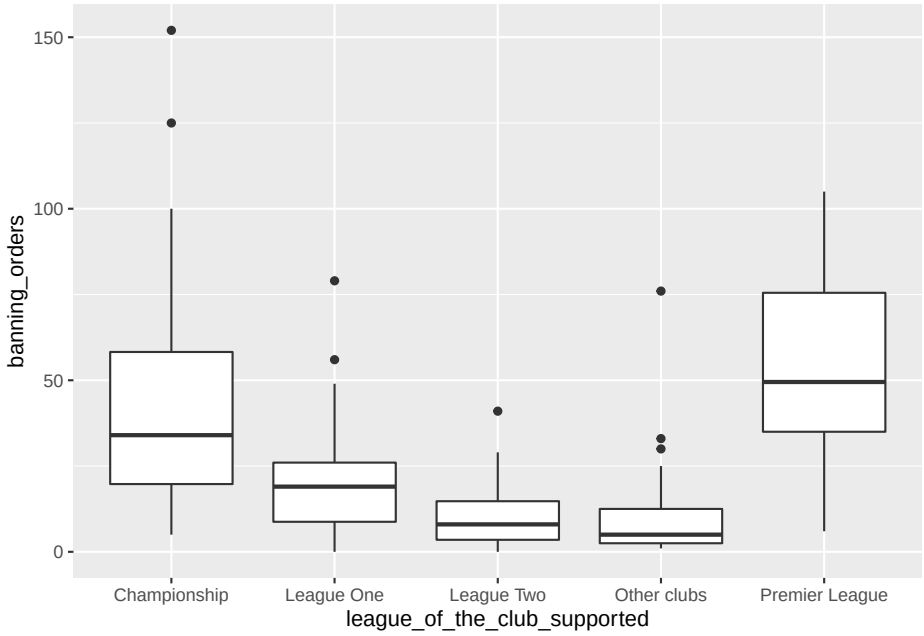
Now to split this by *league_of_the_club_supported*, you use `facet_wrap()` in the coordinate layer of the plot.

```
ggplot(data = fbo, aes(x = banning_orders)) +
  geom_histogram() +
  facet_wrap(~league_of_the_club_supported)
```



Well you can see there's different distribution in each league. But is this easy to compare? Maybe another approach would make it easier? Personally I like boxplots (we will explain them in greater detail below) for showing distribution. So let's try:

```
ggplot(data = fbo, aes(x = league_of_the_club_supported, y = banning_orders)) +
  geom_boxplot()
```

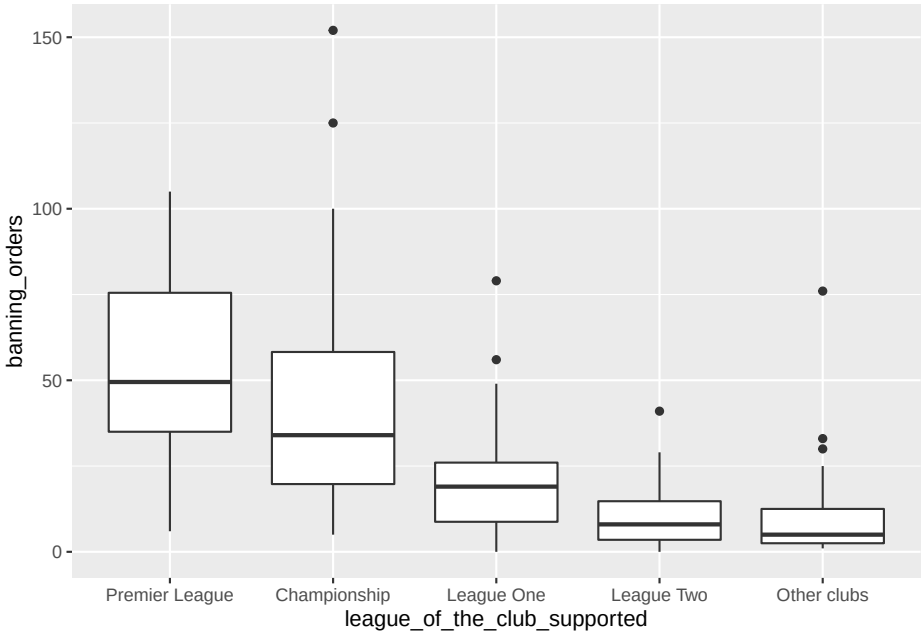


This makes the comparison significantly easier, right? But the order is strange! Remember we talked about factors in previous weeks? Well the good thing about factors is that we can arrange them in their natural order. If we don't describe an order, then R uses the alphabetical order. So let's reorder our factor. To do that we are specifying the levels in the order in which we want to be embedded within the factor. We use code we introduced last week to do this.

```
fbo$league_of_the_club_supported <- factor(fbo$league_of_the_club_supported, levels = c
```

And now create the plot again!

```
ggplot(data = fbo, aes(x = league_of_the_club_supported, y = banning_orders)) +
  geom_boxplot()
```



Now this is great! We can see that the higher the league the more banning orders they have. Any ideas why?

We'll now go through some examples of making graphs using `ggplot2` package stopping a bit more on each of them.

3.4 Visualising numerical variables: Histograms

Histograms are useful ways of representing quantitative variables visually.

As mentioned earlier, we will emphasise in this course the use of the `ggplot()` function. With `ggplot()` you start with a blank canvass and keep adding specific layers. The `ggplot()` function can specify the dataset and the aesthetics (the visual characteristics that represent the data).

To get the data we're going to use here, load up the package `MASS` and then call the *Boston* data into your environment.

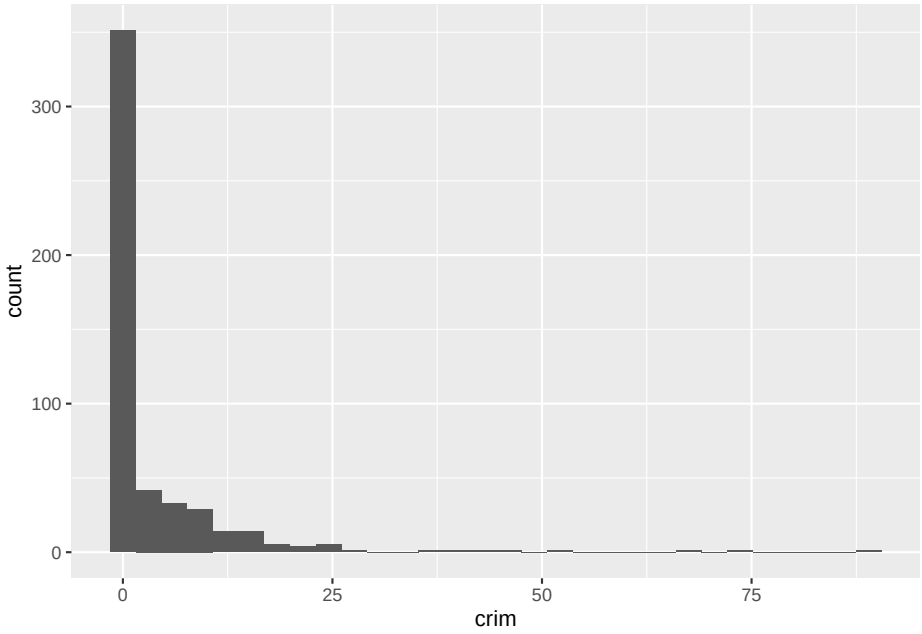
```
library(MASS)
data(Boston)
```

This package has a dataframe called *Boston*. This has data about Housing Values in suburbs of Boston (USA). To access the codebook (how you find out what variables are) use the "?".

OK so let's make a graph about the variable which represents the per capita crime rate by town (*crim*).

If you want to produce a histogram with the `ggplot` function you would use the following code:

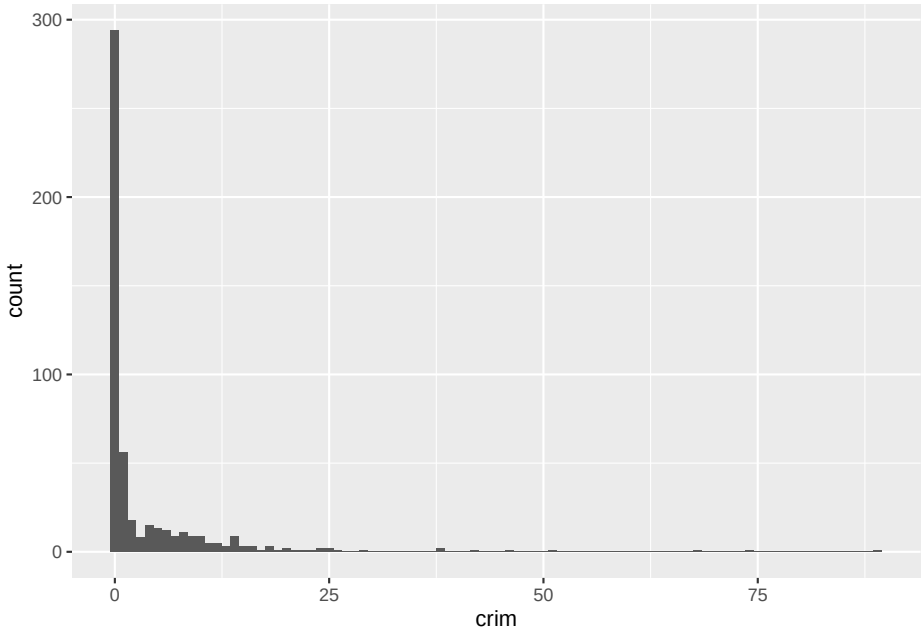
```
ggplot(Boston, aes(x = crim)) +
  geom_histogram()
```



So you can see that `ggplot` works in a way that you can add a series of additional specifications (layers, annotations). In this simple plot the `ggplot` function simply maps *crim* as the variable to be displayed (as one of the **aesthetics**) and the dataset. Then you add the `geom_histogram` to tell R that you want this variable to be represented as a histogram. Later we will see what other things you can add.

A histogram is simply putting cases in "bins" and then creates a bar for each bin. You can think of it as a *visual grouped frequency distribution.* The code we have used so far has used a bin-width of size range/30 as R kindly reminded us in the output. But you can modify this parameter if you want to get a rougher or a more granular picture. In fact, you should *always* play around with different specifications of the bin-width until you find one that tells the full story in a parsimonious way.

```
ggplot(Boston, aes(x = crim)) +
  geom_histogram(binwidth = 1)
```



We can pass arguments to the the geoms, as you see. Here we are changing the size of the bins (for further details on other arguments you can check the help files). Using bin-width of 1 we are essentially creating a bar for every one unit increase in the percent rate of crime. We can still see that most towns have a very low level of crime.

Let's sum the number of towns with a value lower than 1 in the per capita crime rate. We use the sum function for this, specifying we are only interested in adding cases where the value of the variable *crim* is lower than 1.

```
sum(Boston$crim < 1)
```

```
## [1] 332
```

We can see that the large majority of towns, 332 out of 506, have a per capita crime rate below 1%. But we can also see that there are some towns that have a high concentration of crime. This is a spatial feature of crime; it tends to concentrate in particular areas and places. You can see how we can use visualisations to show the data and get a first feeling for how it may be distributed.

When plotting a continuous variable we are interested in the following features:

- **Asymmetry**: whether the distribution is skewed to the right or to the left, or follows a more symmetrical shape.
- **Outliers**: Are there one or more values that seem very unlike the others?

- **Multimodality**: How many peaks has the distribution? More than one peak may indicate that the variable is measuring different groups.
- **Impossibilities or other anomalies**: Values that are simply unrealistic given what we are measuring (e.g., somebody with an age of a 1000 years). Sometimes you may come across a distribution of data with a very high (and implausible) frequency count for a particular value. Maybe you measure age and you have a large number of cases aged 99 (which is often a code used for missing data).
- **Spread**: this gives us an idea of variability in our data.

Often we visualise data because we want to compare distributions. **Most of data analysis is about making comparisons**. We are going to explore whether the distribution of crime in this dataset is different for less affluent areas. The variable *medv* measures in the Boston dataset the median value of owner-occupied homes. For the purposes of this illustration I want to dichotomise[4] this variable, to create a group of towns with particularly low values versus all the others. For further details in how to recode variables with R you may want to read the relevant sections in Quick R or the R Cookbook. We will learn more about recoding and transforming variables in R soon.

How can we create a categorical variable based on information from a quantitative variable? Let's see the following code and pay attention to it and the explanation below.

```
Boston$lowval[Boston$medv <= 17.02] <- "Low value"
Boston$lowval[Boston$medv > 17.02] <- "Higher value"
```

First we tell R to create a new vector (*lowval*) in the Boston data frame. This vector will be assigned the character value "Low value" when the condition within the square brackets is met. That is, we are saying that whenever the value in *medv* is below 17.02 then the new variable *lowval* will equal "Low value". I have chosen 17.02 as this is the first quartile for *medv*. Then we tell R that when the value is greater than 17.02 we will assign those cases to a new textual category called "Higher Value".

The variable we created was a character vector (as we can see if we run the `class` function). so we are going to transform it into a factor using the `as.factor` function (many functions designed to work with categorical variables expect a factor as an input, not just a character vector). If we rerun the `class` function we will see we changed the original variable

```
class(Boston$lowval)
```

```
## [1] "character"
```

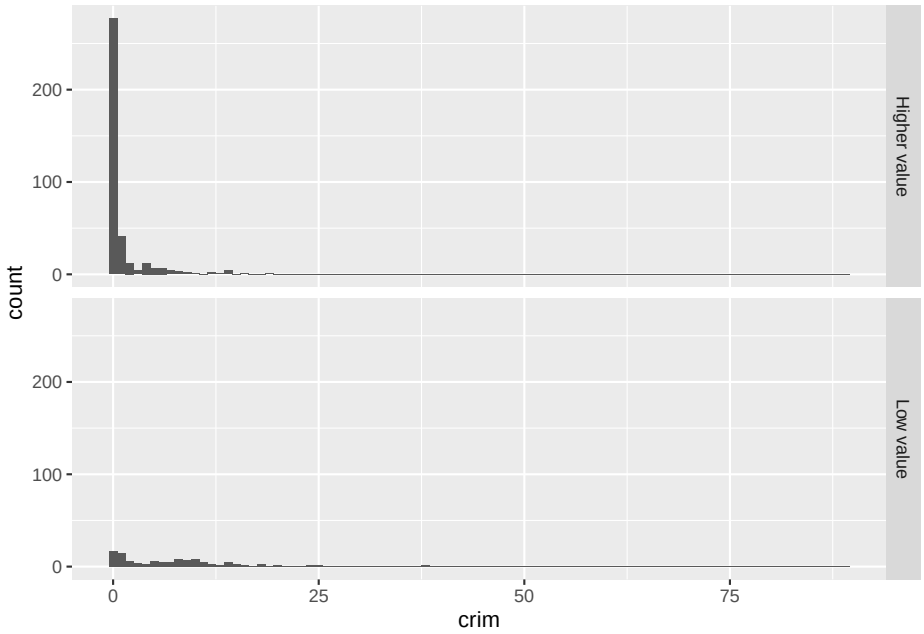
[4] Split into two groups.

```
Boston$lowval <- as.factor(Boston$lowval)
class(Boston$lowval)
```

```
## [1] "factor"
```

Now we can produce the plot. We will do this using **facets**. Facets are another element of the grammar of graphics, we use it to define subsets of the data to be represented as multiple groups, here we are asking R to produce two plots defined by the two levels of the factor we just created.

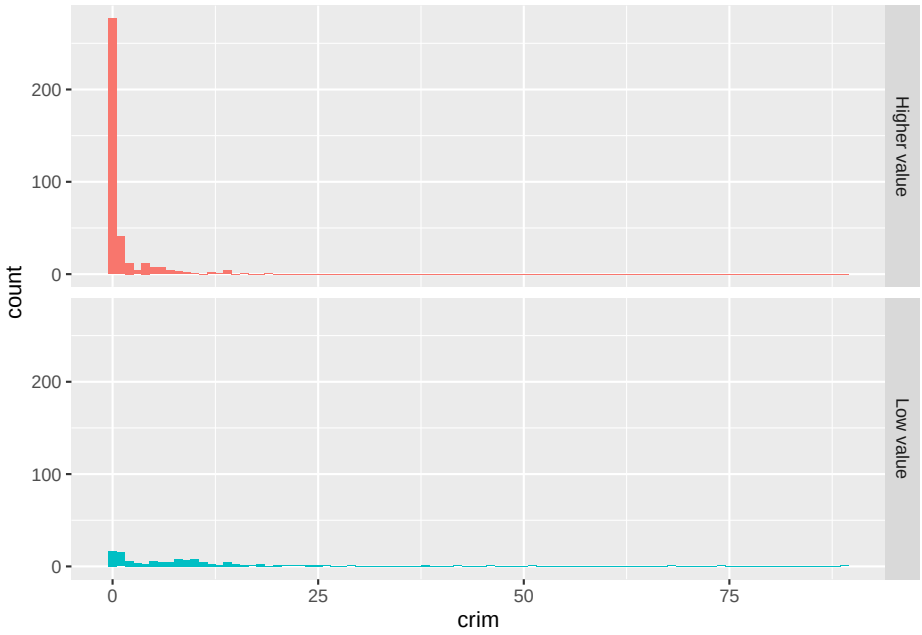
```
ggplot(Boston, aes(x = crim)) +
  geom_histogram(binwidth = 1) +
  facet_grid(lowval ~ .)
```



Visually this may not look great, but it begins to tell a story. We can see that there is a considerable lower proportion of towns with low levels of crime in the group of towns that have cheaper homes. It is a flatter, less skewed distribution. You can see how the `facet_grid()` expression is telling R to create the histogram of the variable mentioned in the `ggplot` function for the groups defined by the categorical input of interest (the factor *lowval*).

We could do a few things that may perhaps help to emphasise the comparison like, for example, adding colour to each of the groups.

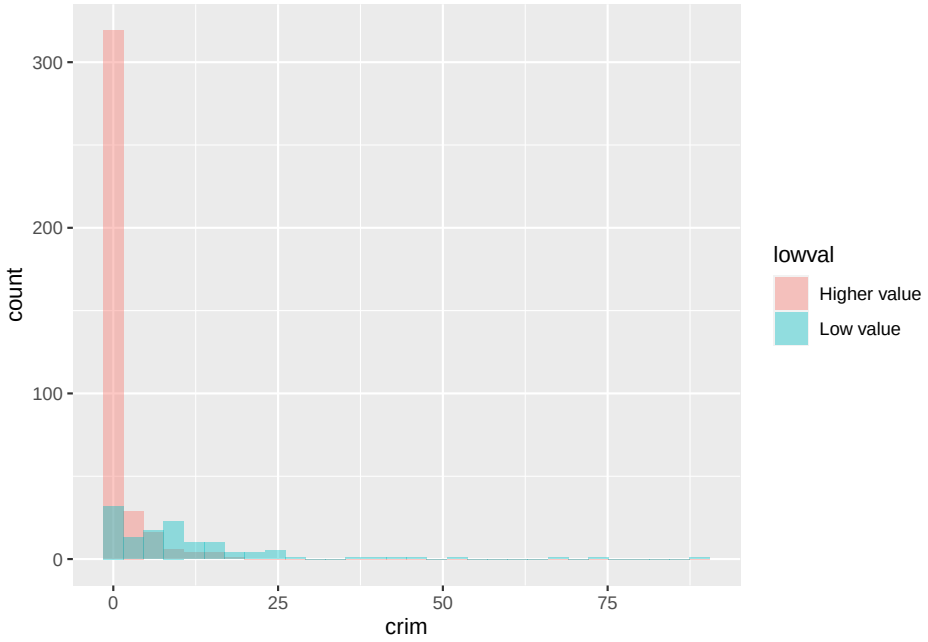
```
ggplot(Boston, aes(x = crim, fill = lowval)) +
  geom_histogram(binwidth = 1) +
  facet_grid(lowval ~ .) +
  theme(legend.position = "none")
```



The `fill` argument within the `aes` is telling R what variable to assign colours. Now each of the levels (groups) defined by the *lowval* variable will have a different colour. The `theme` statement that we add is telling R not to place a legend in the graphic explaining that red is higher value and the greenish colour is low value. We can already see that without a label.

Instead of using facets, we could overlay the histograms with a bit of transparency. Transparencies work better when projecting in screens than in printed document, so keep in mind this when deciding whether to use them instead of facets. The code is as follows:

```
ggplot(Boston, aes(x = crim, fill = lowval)) +
  geom_histogram(position = "identity", alpha = 0.4)
```

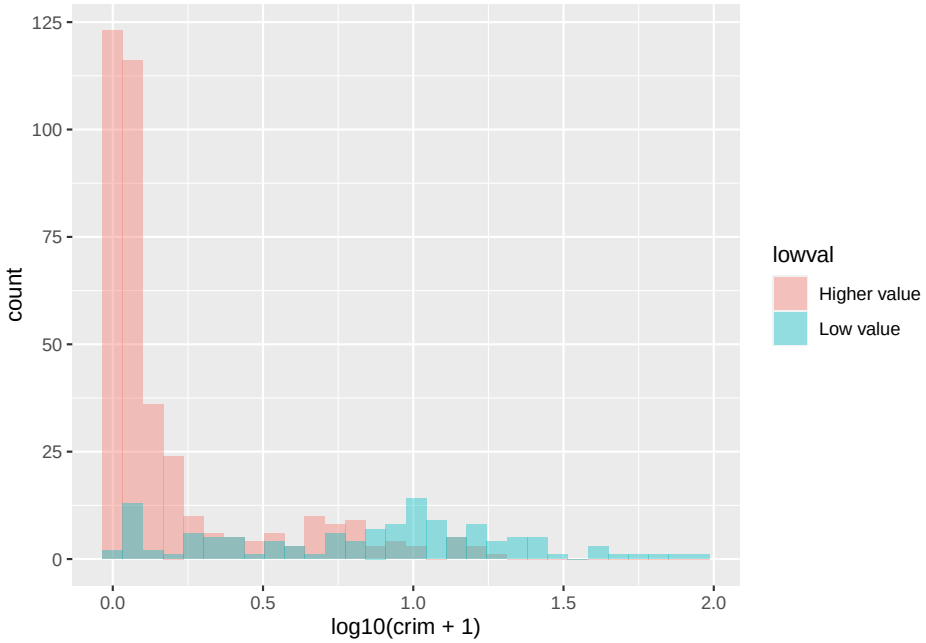


In the code above, the `fill` argument identifies again the factor variable in the dataset grouping the cases. Also, `position = identity` tells R to overlay the distributions and `alpha` asks for the degree of transparency, a lower value (e.g., 0.2) will be more transparent.

In this case, part of the problem we have is that the skew can make it difficult to appreciate the differences. When you are dealing with skewed distributions such as this, it is sometimes convenient to use a transformation [5]. We will come back to this later this semester. For now suffice to say, that taking the logarithm of a skewed variable helps to reduce the skew and to see patterns more clearly. In order to visualise the differences here a bit better we could ask for the logarithm of the crime per capita rate. Notice how I also add a constant of 1 to the variable *crim*, this is to avoid NA values in the newly created variable if the value in *crim* is zero (you cannot take the log of 0).

```
ggplot(Boston, aes(x = log10(crim + 1), fill = lowval)) +
  geom_histogram(position = "identity", alpha = 0.4)
```

[5] This is an interesting blog entry in solutions when you have highly skewed data.

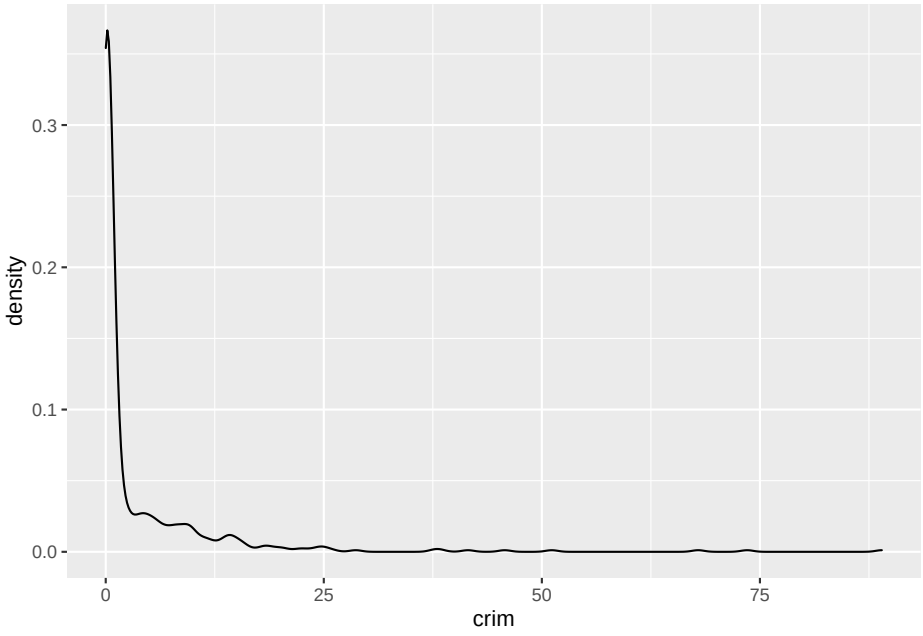


The plot now is a bit clearer. It seems pretty evident that the distribution of crime is quite different between these two types of towns.

3.5 Visualising numerical variables: Density plots

For smoother distributions, you can use density plot. You should have a healthy amount of data to use these or you could end up with a lot of unwanted noise. Let's first look at the single density plot for all cases. Notice all we are doing is invoking a different kind of geom:

```
ggplot(Boston, aes(x = crim)) +
  geom_density()
```

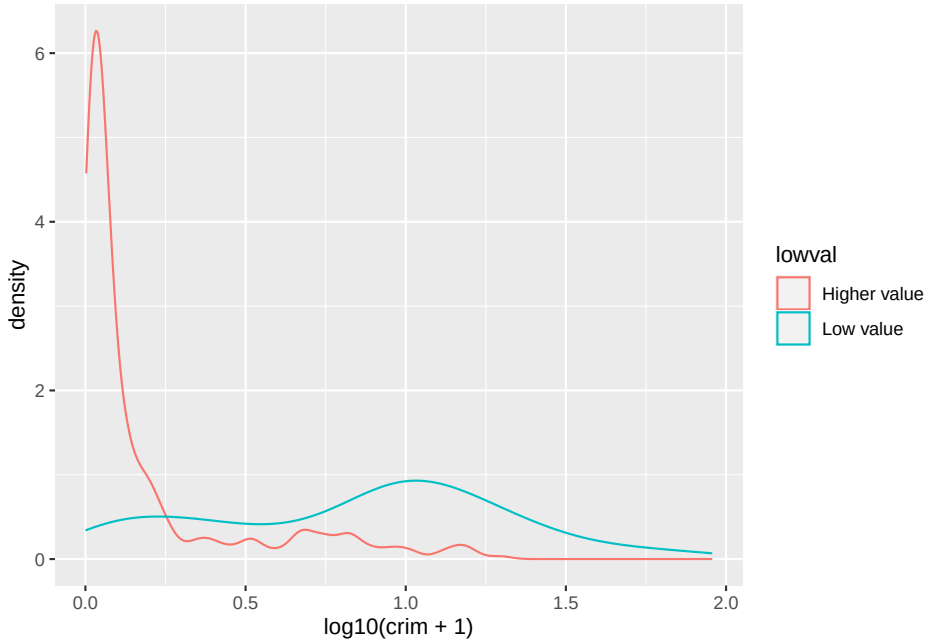


In a density plot, we attempt to visualize the underlying probability distribution of the data by drawing an appropriate continuous curve. So in a density plot then the area under the lines sum to 1 and the Y, vertical, axis now gives you the estimated (guessed) probability for the different values in the X, horizontal, axis. This curve is guessed from the data and the method we use for this guessing or estimation is called kernel density estimation. You can read more about density plots here.

In this plot we can see that there is a high estimated probability of observing a town with near zero per capita crime rate and a low estimated probability of seeing towns with large per capita crime rates. As you can observe it provides a smoother representation of the distribution (as compared to the histograms).

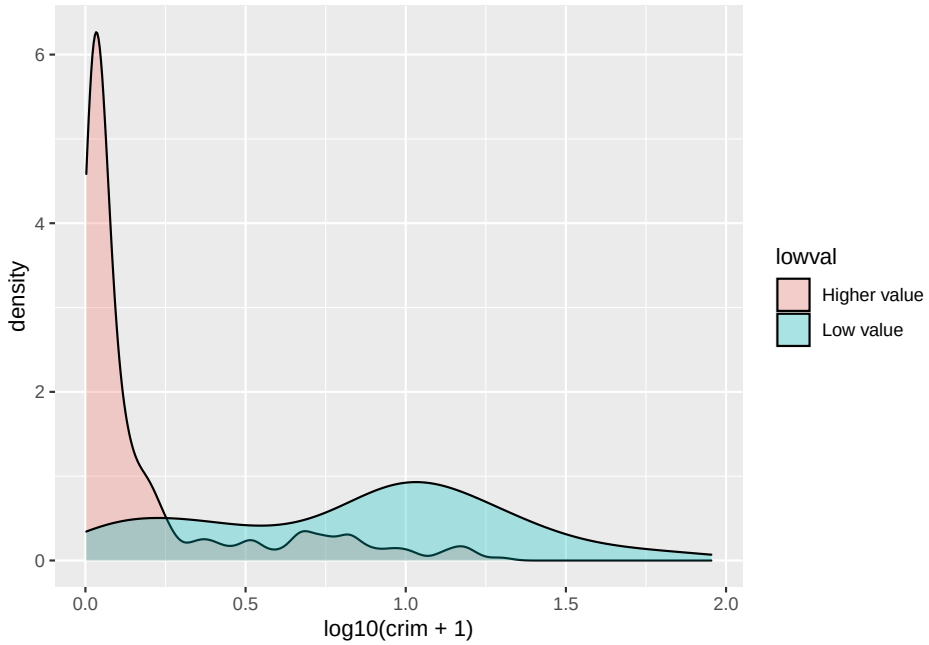
You can also use this to compare the distribution of a quantitative variable across the levels in a categorical variable (factor) and, as before, is possibly better to take the log of skewed variables such as crime:

```
#We are mapping "lowval" as the variable colouring the lines
ggplot(Boston, aes(x = log10(crim + 1), colour = lowval)) +
  geom_density()
```



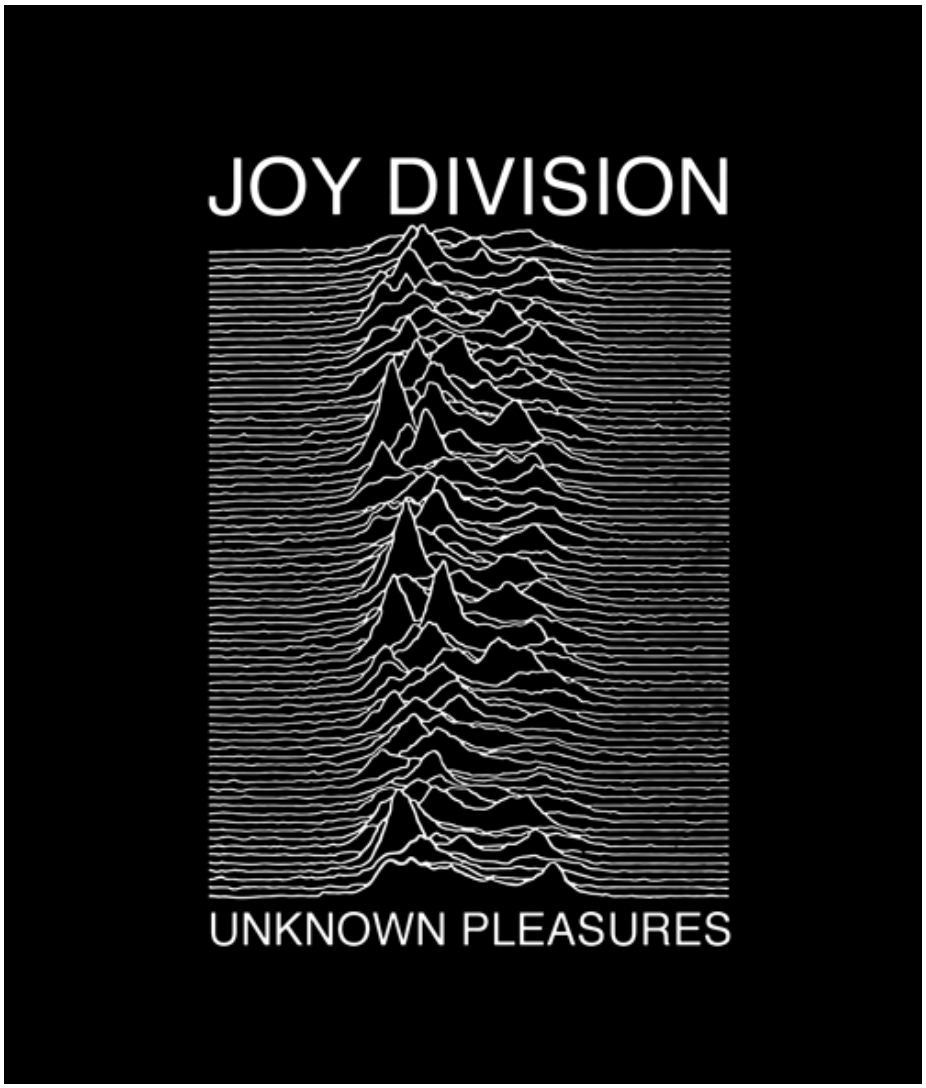
Or you could use transparencies:

```
ggplot(Boston, aes(x = log10(crim + 1), fill = lowval)) +
  geom_density(alpha = .3)
```



Did you notice the difference with the comparative histograms? By using density plots we are rescaling to ensure the same area for each of the levels in our grouping variable. This makes it easier to compare two groups that have different frequencies. The areas under the curve add up to 1 for both of the groups, whereas in the histogram the area within the bars represent the number of cases in each of the groups. If you have many more cases in one group than the other it may be difficult to make comparisons or to clearly see the distribution for the group with fewer cases. So, this is one of the reasons why you may want to use density plots.

Density plots are a good choice when you want to compare up to three groups. If you have many more groups you may want to consider other alternatives. One such alternative is the **ridgeline plot**, also often called the Joy Division plot (since it was immortalised in the cover of one of their albums):



They can be produced with the `ggridges` package. Before we dichotomised the variable *medv* in a manual way, we can use more direct ways of splitting numerical variables into various categories using information already embedded on them. Say we want to split *medv* into deciles. We could use the `mutate` function in `dplyr` for this.

```
library(dplyr)
Boston <- mutate(Boston, dec_medv = ntile(medv, 10))
```

The `mutate` function adds a new variable to our existing data frame object. We are naming this variable *dec_medv* because we are going to split *medv* into ten

groups of equal size (this name is arbitrary, you may call it something else). To do this we will use the `ntile` function as an argument within mutate. We will define the new *dec_medv* variable explaining to R that this variable will be the result of passing the `ntile` function to *medv*. So that `ntile` breaks *medv* into 10 we pass this value as an argument to the function. So that the result of executing `mutate` are stored, we assign this to the *Boston* object.

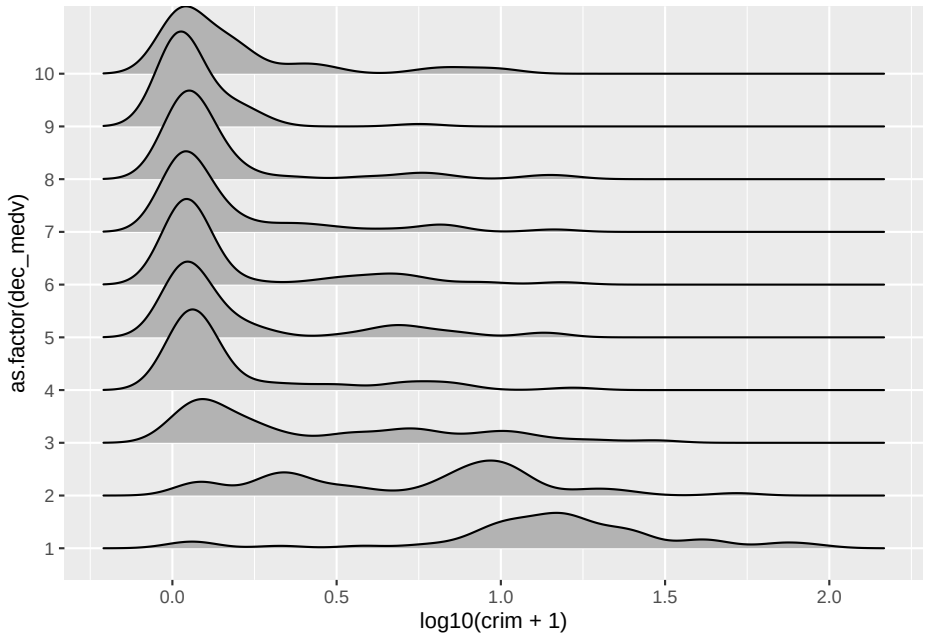
Check the results:

```
table(Boston$dec_medv)
```

```
##
##  1  2  3  4  5  6  7  8  9 10
## 51 51 51 51 51 51 50 50 50 50
```

We can now use this new variable to illustrate the use of the `ggridge` package. First you will need to install this package and then load it. You will see all this package does is to extend the functionality of `ggplot2` by adding a new type of geom. Here the variable defining the groups needs to be a factor, so we will tell `ggplot` to treat *dec_medv* as a factor using `as.factor`. Using `as.factor` in this way save us from having to create yet another variable that we are going to store as a factor. Here we are not creating a new variable, we are just telling R to treat this numeric variable *as if* it were a factor. Make sure you understand this difference.

```
library(ggridges)
ggplot(Boston, aes(x = log10(crim + 1), y = as.factor(dec_medv))) + geom_density_ridges()
```



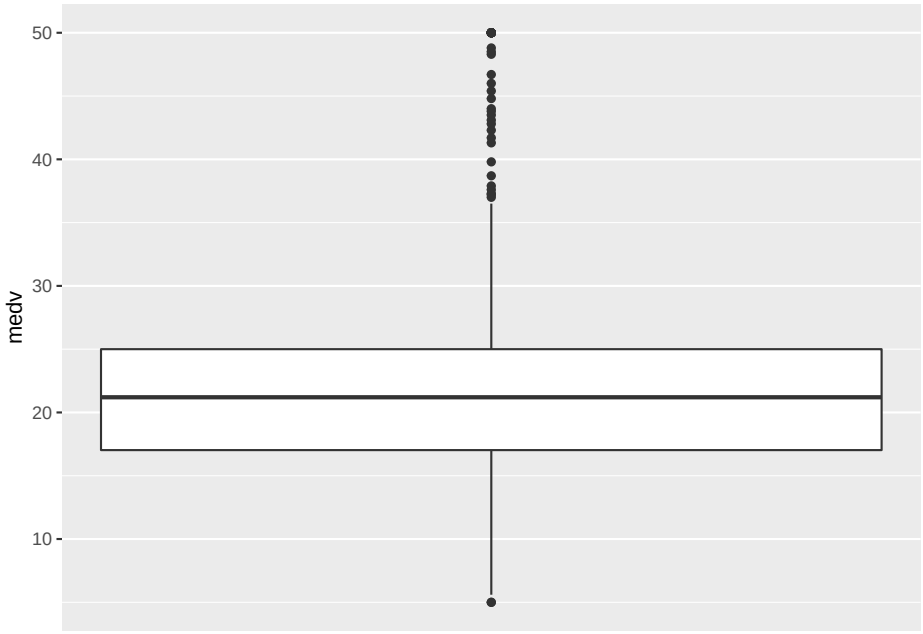
We can see that the distribution of crime is particularly different when we focus in the three deciles with the lowest level of income. For more details on this kind of plot you can read the vignette for this package.

3.6 Visualising numerical variables: Box plots

Box plots are an interesting way of presenting the 5 number summary (the minimum value, the first quartile, the median, the third quartile, and the maximum value of a set of numbers) in a visual way. This video may help you to understand them better. If we want to use `ggplot` for plotting a single numerical variable we need some convoluted code, since `ggplot` assumes you want a boxplot to compare various groups. Therefore we need to set some arbitrary value for the grouping variable and we also may want to remove the x-axis tick markers and label.

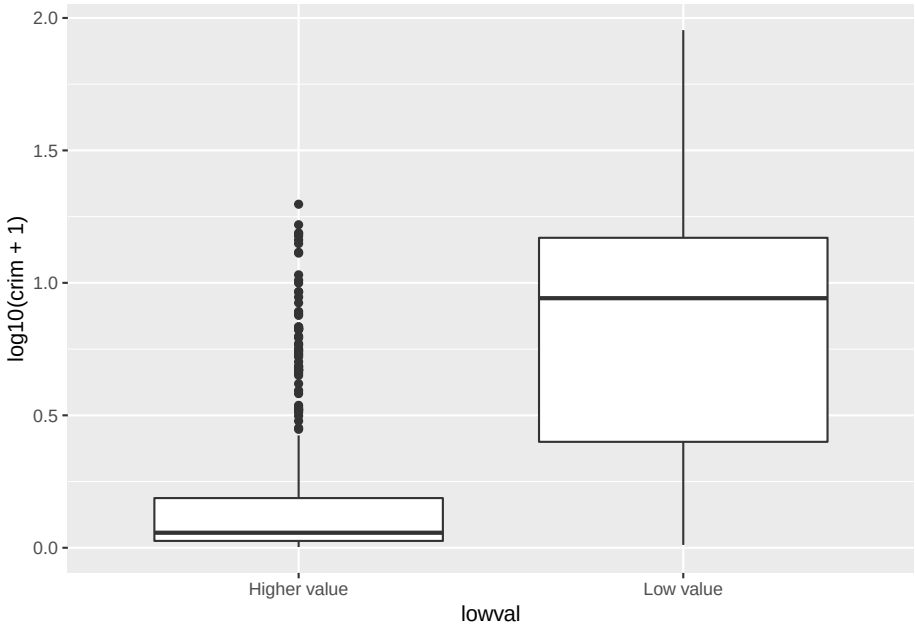
For this illustration, I am going to display the distribution of the median value of property in the various towns instead of crime.

```
ggplot(Boston, aes(x = 1, y = medv)) +
  geom_boxplot() +
  scale_x_continuous(breaks = NULL) + #removes the tick markers from the x axis
  theme(axis.title.x = element_blank())
```



Boxplots, however, really come to life when you do use them to compare the distribution of a quantitative variable across various groups. Let's look at the distribution of *log(crime)* across cheaper and more expensive areas:

```
ggplot(Boston, aes(x = lowval, y=log10(crim + 1))) +
  geom_boxplot()
```



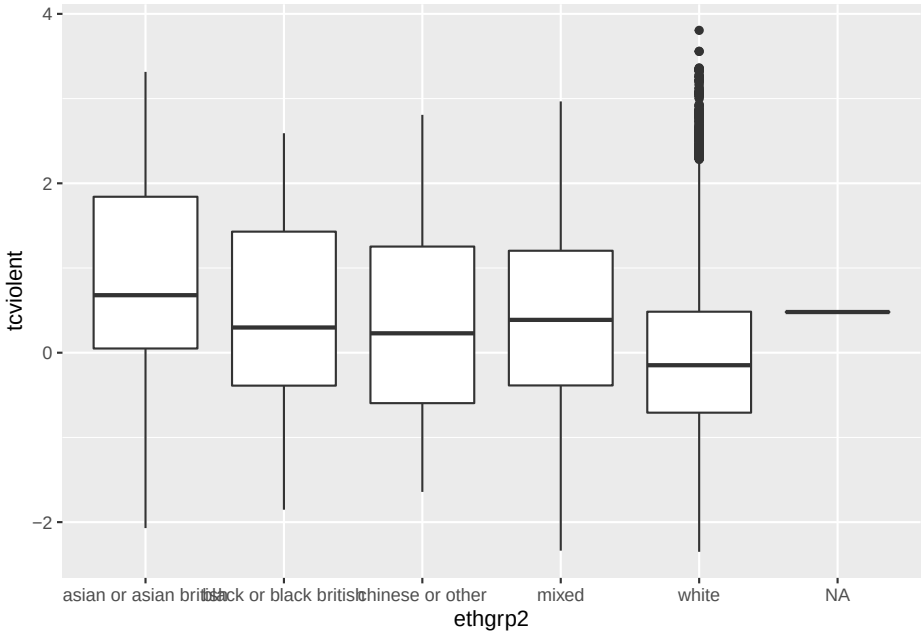
With a boxplot like this you can see straight away that the bulk of cheaper areas are very different from the bulk of more expensive areas. The first quartile of the distribution for low areas about matches the point at which we start to see **outliers** for the more expensive areas.

This can be even more helpful when you have various groups. Let's try an example using the *BCS0708* data frame. This is a dataset from the 2007/08 British Crime Survey.You can use the code below to download it.

```
##R in Windows have some problems with https addresses, that's why we need to do this i
urlfile<-'https://raw.githubusercontent.com/jjmedinaariza/LAWS70821/master/BCS0708.csv
#We create a data frame object reading the data from the remote .csv file
BCS0708<-read.csv(url(urlfile))
```

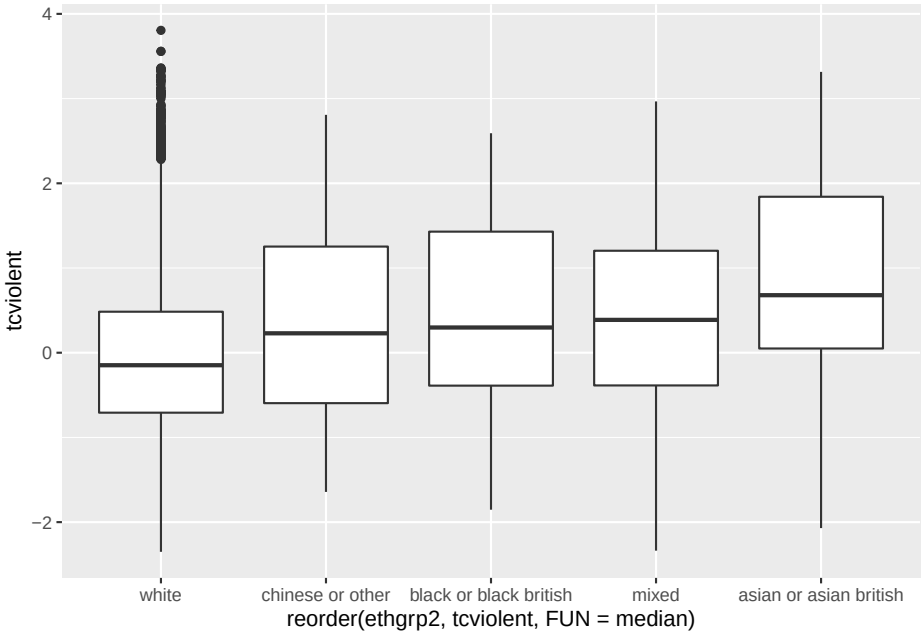
This dataset contains a quantitative variable that measures the level of worry for crime (*tcviolent*): high scores represent high levels of worry. We are going to see how the score in this variable changes according to ethnicity (*ethgrp2*).

```
#A comparative boxplot of ethnicity and worry about violent crime
ggplot(BCS0708, aes(x = ethgrp2, y = tcviolent)) +
  geom_boxplot()
```



Nice. But could be nicer. To start with we could order the groups along the X axis so that the ethnic groups are positioned according to their level of worry. Secondly, we may want to exclude the information for the NA cases on ethnicity (represented by a flat line).

```
#A nicer comparative boxplot (excluding NA and reordering the X variable)
ggplot(filter(BCS0708, !is.na(ethgrp2) & !is.na(tcviolent)),
       aes(x = reorder(ethgrp2, tcviolent, FUN = median), y = tcviolent)) +
        geom_boxplot()
```



The `filter` function from `dplyr` is using a logical argument to tell R to only use the cases that do not have NA values in the two variables that we are using. The exclamation mark followed by `is.na` and then the name of a variable is R way of saying "the contrary of is NA for the specified variable". So in essence we are saying to R just look at data that is not NA in these variables. The `reorder` function on the other hand is asking R to reorder the levels in ethnicity according to the median value of worry of violent crime. Since we are using those functions *within* the `ggplot` function this subsetting and this reordering (as with `as.factor` earlier) are not introducing permanent changes in your original dataset. If you prefer to reorder according to the mean you only need to change that parameter after the `FUN` option (e.g, `FUN = mean`).

3.7 Exploring relationships between two quantitative variables: scatterplots

So far we have seen how you can use histograms, density plots and boxplots to plot numerical variables and to compare groups in relation to numerical variables. Another way of saying that is that you can use comparative histograms, density plots, or boxplots to assess the relationship between a numerical variable and a categorical variable (the variable that defines the groups). How do you explore the relationship between two numerical variables?

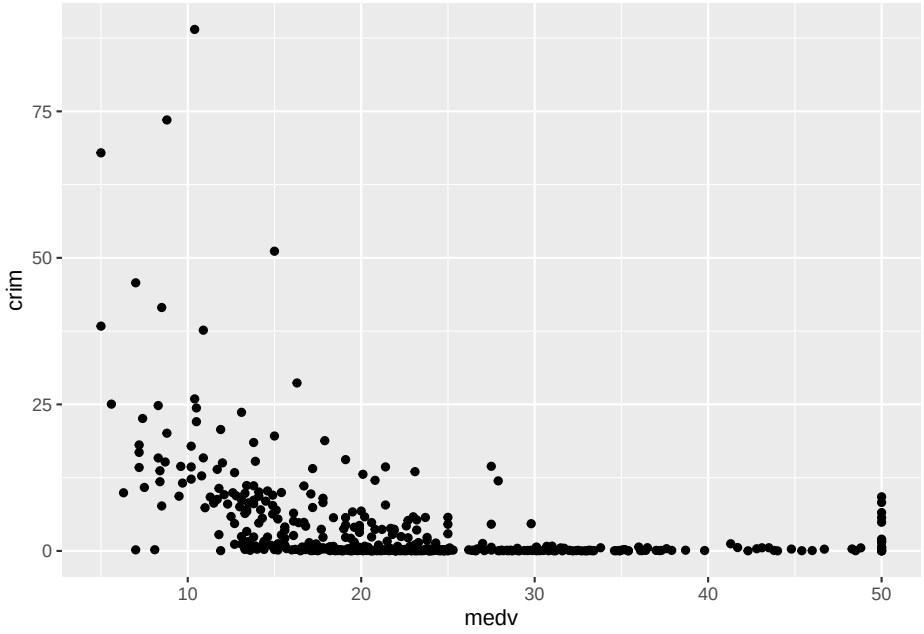
When looking at the relationship between two quantitative variables nothing beats the **scatterplot**. This is a lovely article in the history of the scatterplot

and this video may help you to understand them better.

A scatterplot plots one variable in the Y axis, and another in the X axis. Typically, if you have a clear outcome or response variable in mind, you place it in the Y axis, and you place the explanatory variable in the X axis.

This is how you produce a scatterplot with `ggplot()`:

```
#A scatterplot of crime versus median value of the properties
ggplot(Boston, aes(x = medv, y = crim)) +
  geom_point()
```



Each point represents a case in our dataset and the coordinates attached to it in this two dimensional plane are given by their value in the Y (crime) and X (median value of the properties) variables.

What do you look for in a scatterplot? You want to assess global and local patterns, as well as deviations. We can see clearly that at low levels of *medv* there is a higher probability in this data that the level of crime is higher. Once the median value of the property hits $30,000 the level of crime is nearly zero for all towns. So far so good, and surely predictable. The first reason why we look at scatterplots is to check our hypothesis (e.g., poorer areas, more crime).

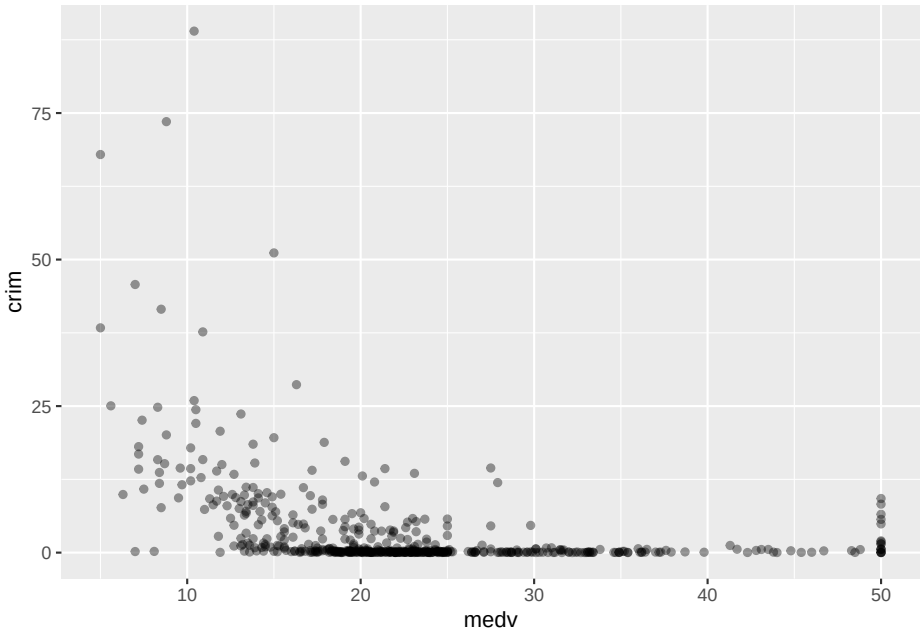
However, something odd seems to be going on when the median property value is around $50,000. All of the sudden the variability for crime goes up. We seem to have some of the more expensive areas also exhibiting some fairly decent level of crime. In fact, there is quite a break in the distribution. What's going on?

To be honest I have no clue. But the pattern at the higher level of property value is indeed odd, it is just too abrupt to be natural.

This is the second reason why you want to plot your data before you do anything else. It helps you to detect apparent anomalies. I say this is an anomaly because the break in pattern is quite noticeable and abrupt. It is hard to think of a natural process that would generate this sudden radical increase in crime once the median property value reaches the 50k dollars mark. If you were analysing this for real, you would want to know what's really driving this pattern (e.g., find out about the original data collection, the codebook, etc.): perhaps the maximum median value was capped at 50K dollars and we are seeing this as a dramatic increase when the picture is more complex? For now we are going to let this rest.

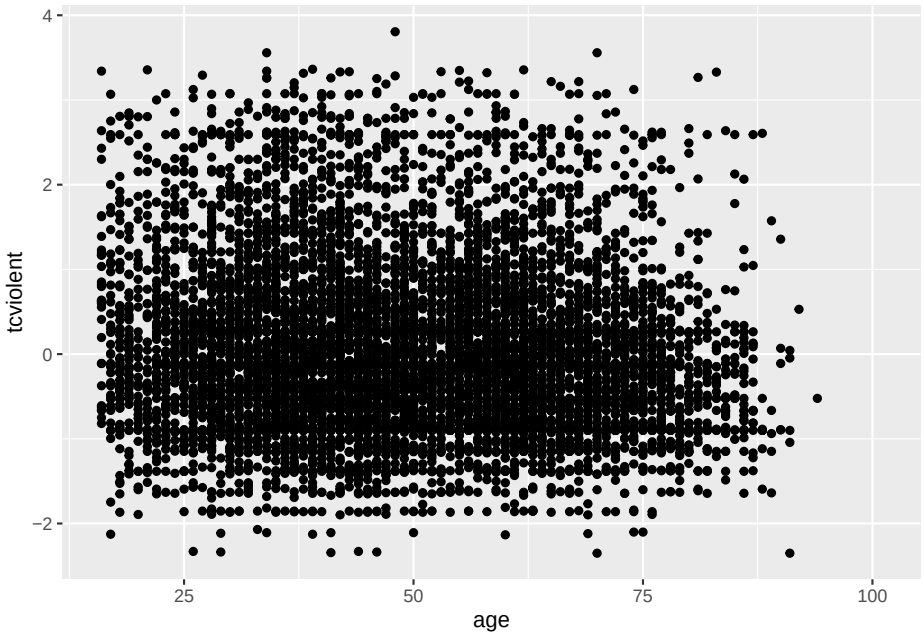
One of the things you may notice with a scatterplot is that even with a smallish dataset such as this, with just about 500 cases, **overplotting** may be a problem. When you have many cases with similar (or even worse the same) value, it is difficult to tell them apart. Imagine there is only 1 case with a particular combination of X and Y values. What you see? A single point. Then imagine you have 500 cases with that same combination of values for X and Y. What do you see? Still a single point. There's a variety of ways for dealing with overplotting. One possibility is to add some **transparency** to the points:

```
ggplot(Boston, aes(x = medv, y = crim)) +
  geom_point(alpha=.4) #you will have to test different values for alpha
```

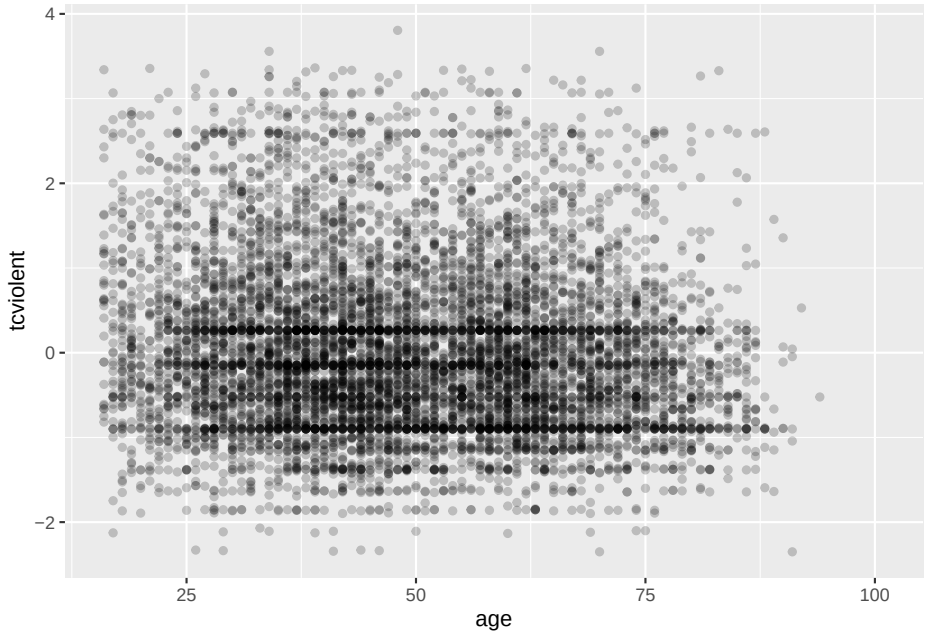


Why this is an issue may be more evident with the `BCS0708 data`. Compare the two plots:

```
ggplot(BCS0708, aes(x = age, y = tcviolent)) +
  geom_point()
```




```
ggplot(BCS0708, aes(x = age, y = tcviolent)) +
  geom_point(alpha=.2)
```

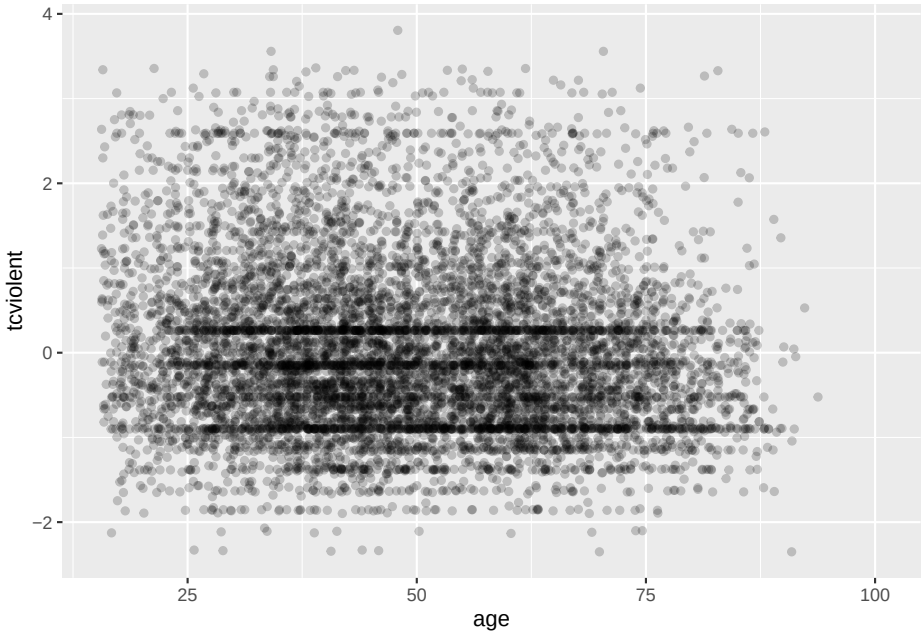


The second plot gives us a better idea of where the observations seem to concentrate in a way that we could not see with the first.

Overplotting can occur when a continuous measurement is rounded to some convenient unit. This has the effect of changing a continuous variable into a discrete ordinal variable. For example, age is measured in years and body weight is measured in pounds or kilograms. Age is a discrete variable, it only takes integer values. That's why you see the points lined up in parallel vertical lines. This also contributes to the overplotting in this case.

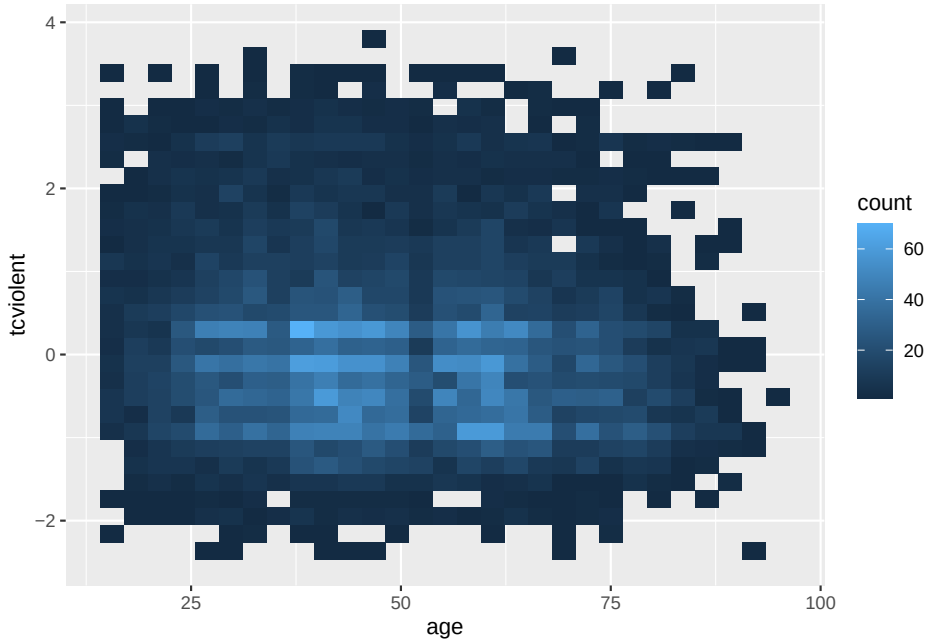
One way of dealing with this particular problem is by **jittering**. Jittering is the act of adding random noise to data in order to prevent overplotting in statistical graphs. In `ggplot` one way of doing this is by passing an argument to `geom_point` specifying you want to jitter the points. This will introduce some random noise so that *age* looks less discrete.

```
ggplot(BCS0708, aes(x = age, y = tcviolent)) +
  geom_point(alpha=.2, position="jitter") #Alternatively you could replace geom_point(
```

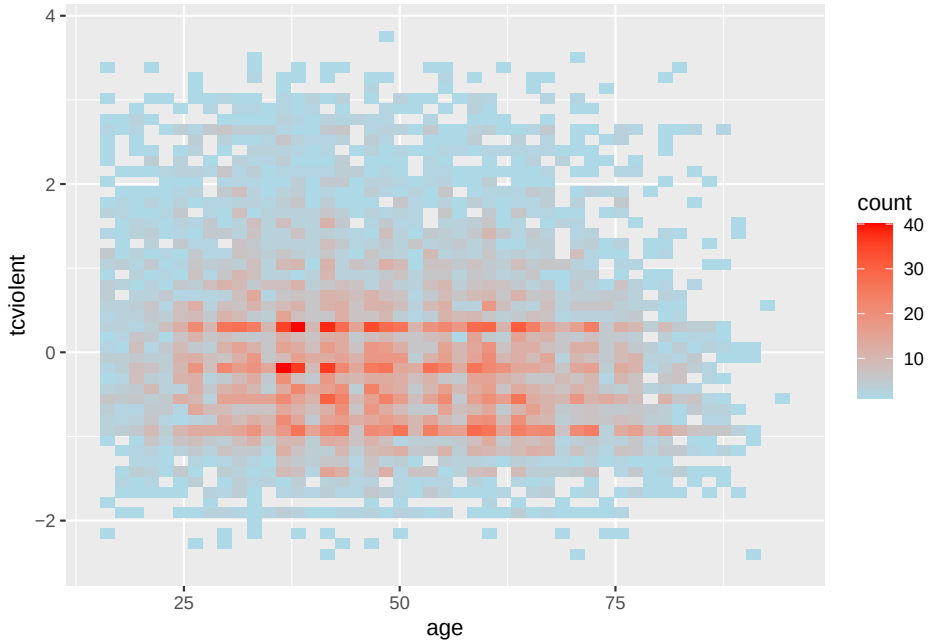


Another alternative for solving overplotting is to **bin the data** into rectangles and map the density of the points to the fill of the colour of the rectangles.

```
ggplot(BCS0708, aes(x = age, y = tcviolent)) +
  stat_bin2d()
```




```
#The same but with nicer graphical parameters
ggplot(BCS0708, aes(x = age, y = tcviolent)) +
  stat_bin2d(bins=50) + #by increasing the number of bins we get more granularity
  scale_fill_gradient(low = "lightblue", high = "red") #change colors
```

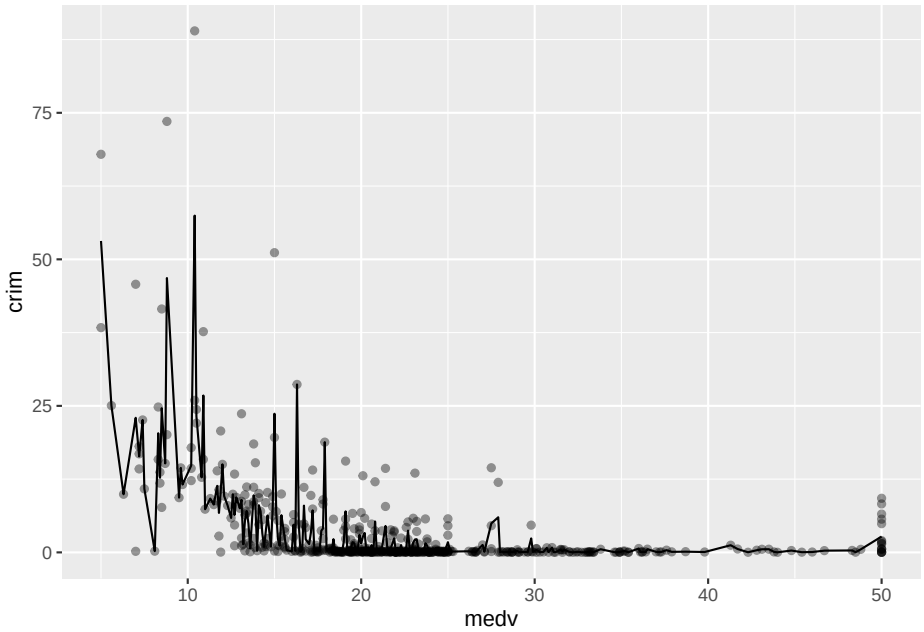


What this is doing is creating boxes within the two dimensional plane; counting the number of points within those boxes; and attaching a colour to the box in function of the density of points within each of the rectangles.

When looking at scatterplots, sometimes it is useful to summarise the relationships by mean of drawing lines. You could for example add a line representing the **conditional mean**. A conditional mean is simply mean of your Y variable for each value of X. Let's go back to the Boston dataset. We can ask R to plot a line connecting these means using `geom_line()` and specifying you want the conditional means.

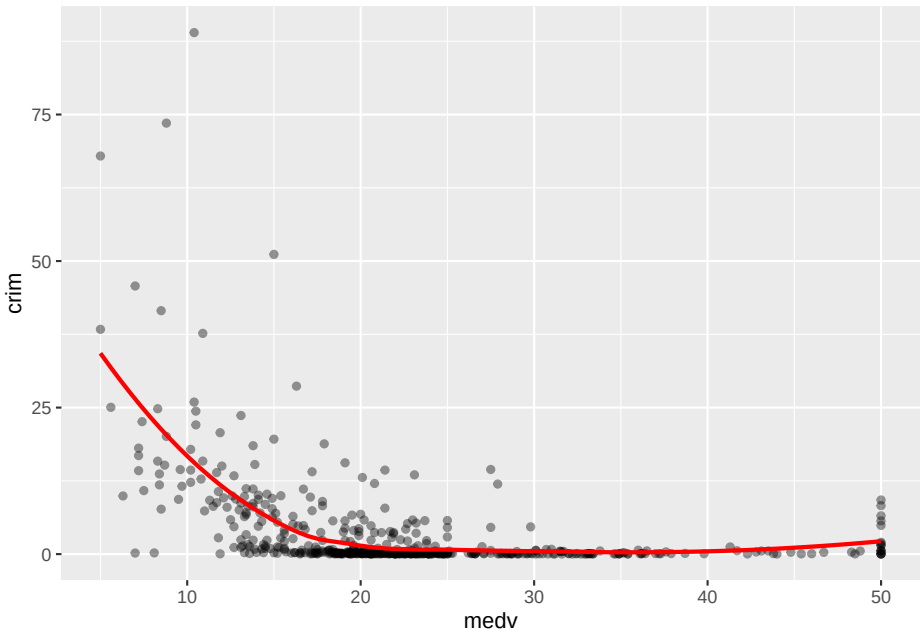
```
ggplot(Boston, aes(x = medv, y = crim)) +
  geom_point(alpha=.4) +
  geom_line(stat='summary', fun.y=mean)
```

```
## Warning: Ignoring unknown parameters: fun.y
```



With only about 500 cases there are loads of ups and downs. If you have many more cases for each level of X the line would look less rough. You can, in any case, produce a smoother line using `geom_smooth` instead. We will discuss later this semester how this line is computed (although you will see the R output tells you, you are using something call the "loess" method). For now just know that is a line that tries to *estimate*, to guess, the typical value for Y for each value of X.

```
ggplot(Boston, aes(x = medv, y = crim)) +
  geom_point(alpha=.4) +
  geom_smooth(colour="red", size=1, se=FALSE) #We'll explain later this semester what
```



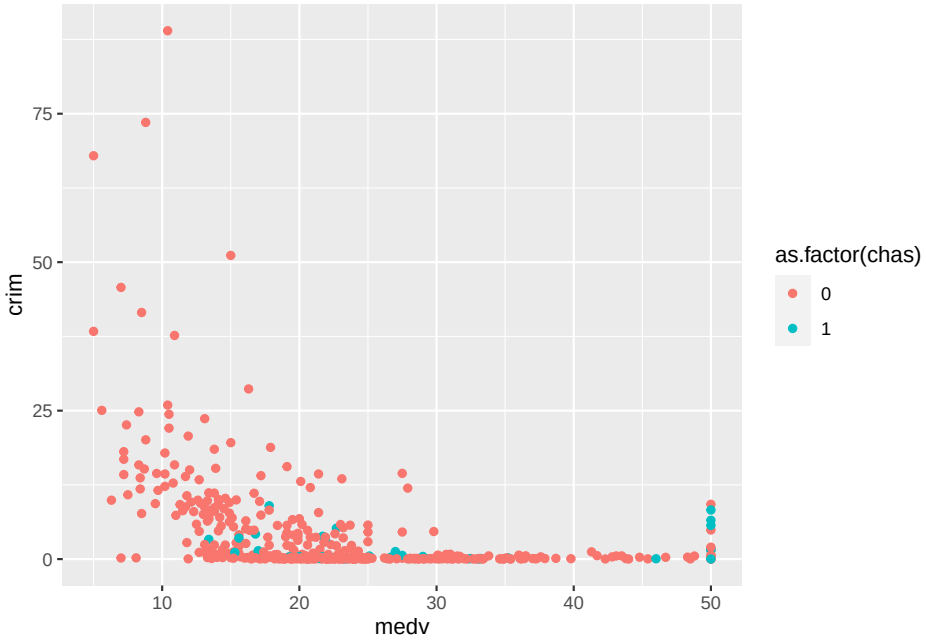
As you can see here you produce a smoother line than with the conditional means. The line, as the scatterplot, seems to be suggesting an overall curvilinear relationship that almost flattens out once property values hit $20k.

3.8 Scatterplots conditioning in a third variable

There are various ways to plot a third variable in a scatterplot. You could go 3D and in some contexts that may be appropriate. But more often than not it is preferable to use only a two dimensional plot.

If you have a grouping variable you could map it to the colour of the points as one of the aesthetics arguments. Here we return to the *Boston* scatterplot but will add a third variable, that indicates whether the town is located by the river or not.

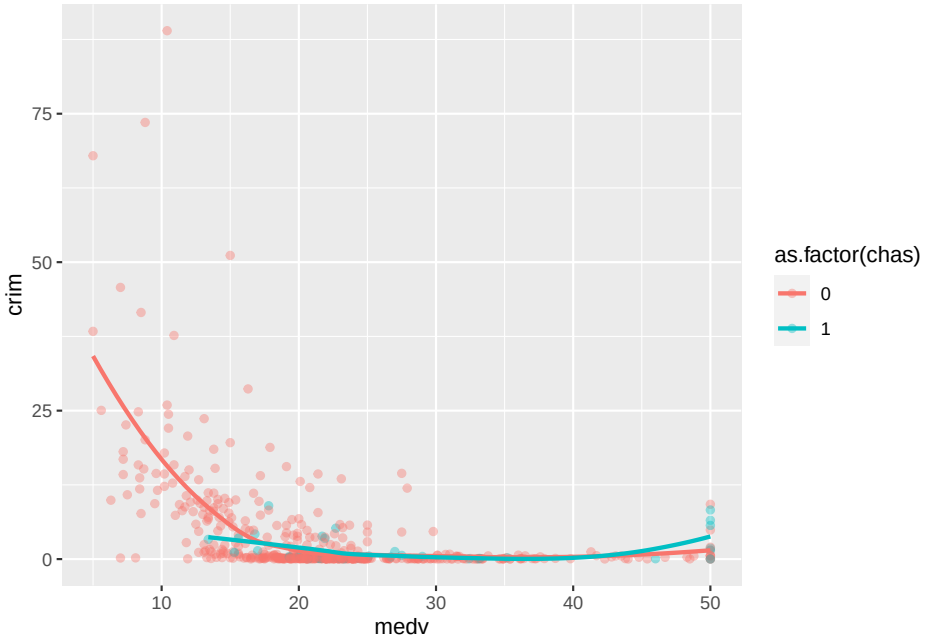
```
#Scatterplot with two quantitative variables and a grouping variable, we are telling R
ggplot(Boston, aes(x = medv, y = crim, colour = as.factor(chas))) +
  geom_point()
```



Curiously, we can see that there's quite a few of those expensive areas with high levels of crime that seem to be located by the river. Maybe that is a particularly attractive area?

As before you can add smooth lines to capture the relationship. What happens now, though, is that `ggplot` will produce a line for each of the levels in the categorical variable grouping the cases:

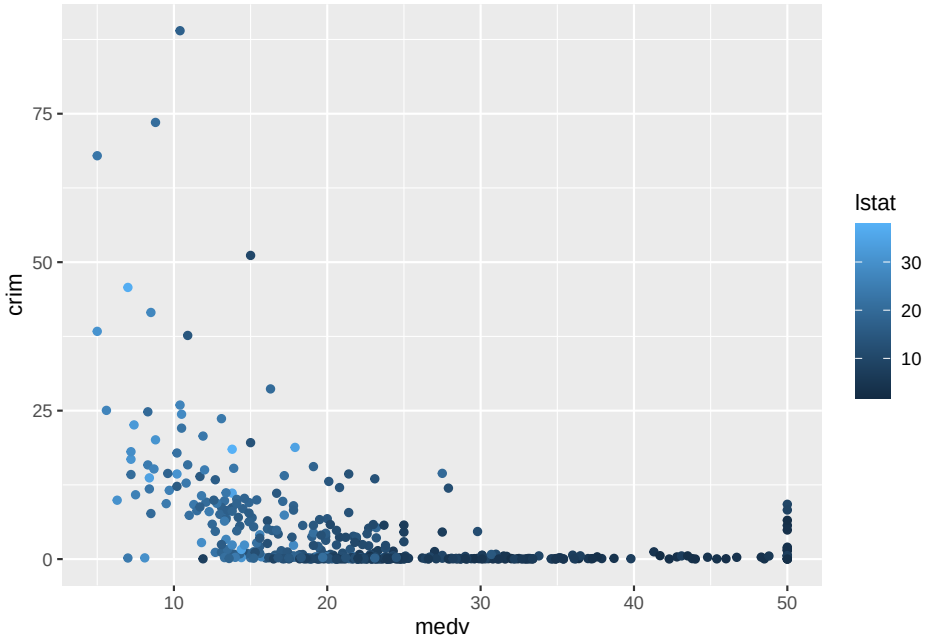
```
ggplot(Boston, aes(x = medv, y = crim, colour = as.factor(chas))) +
  geom_point(alpha=.4) + #I am doing the points semi-transparent to see the lines better
  geom_smooth(se=FALSE, size=1) #I am doing the lines thicker to see them better
```



You can see how the relationship between crime and property values is more marked for areas not bordering the river, mostly because you have considerably fewer cheaper areas bordering the river. Notice as well the upward trend in the green line at high values of *medv*. As we saw there seems to be quite a few of those particularly more expensive areas that have high crime and seem to be by the river.

We can also map a quantitative variable to the colour aesthetic. When we do that, instead of different colours for each category we have a gradation in colour from darker to lighter depending on the value of the quantitative variable. Below we display the relationship between crime and property values conditioning on the status of the area (high values in *lstat* represent lower status).

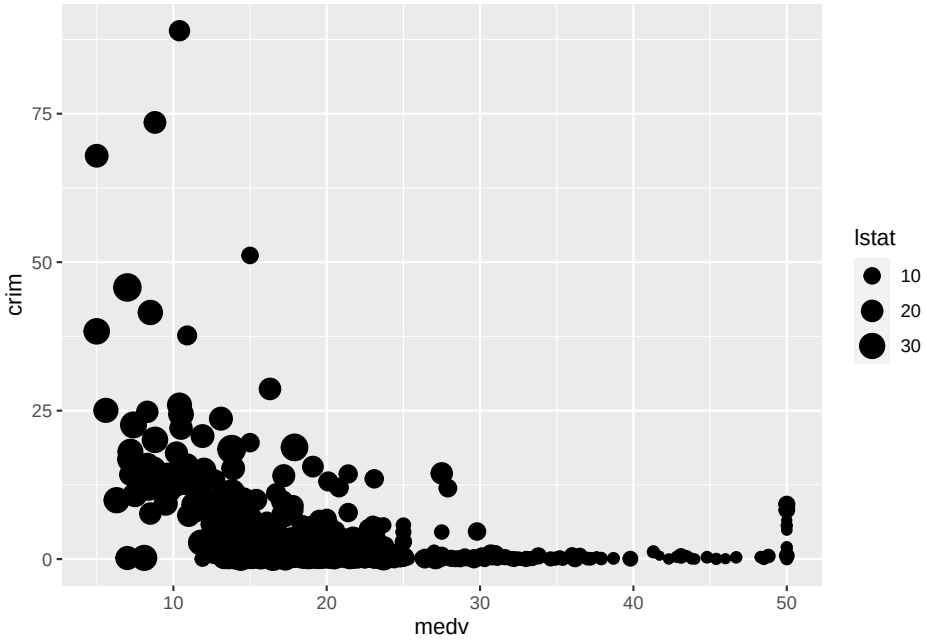
```
ggplot(Boston, aes(x = medv, y = crim, colour = lstat)) +
  geom_point()
```



As one could predict *lstat* and *medv* seem to be correlated. The areas with low status tend to be the areas with cheaper properties (and more crime) and the areas with higher status tend to be the areas with more expensive properties (and less crime).

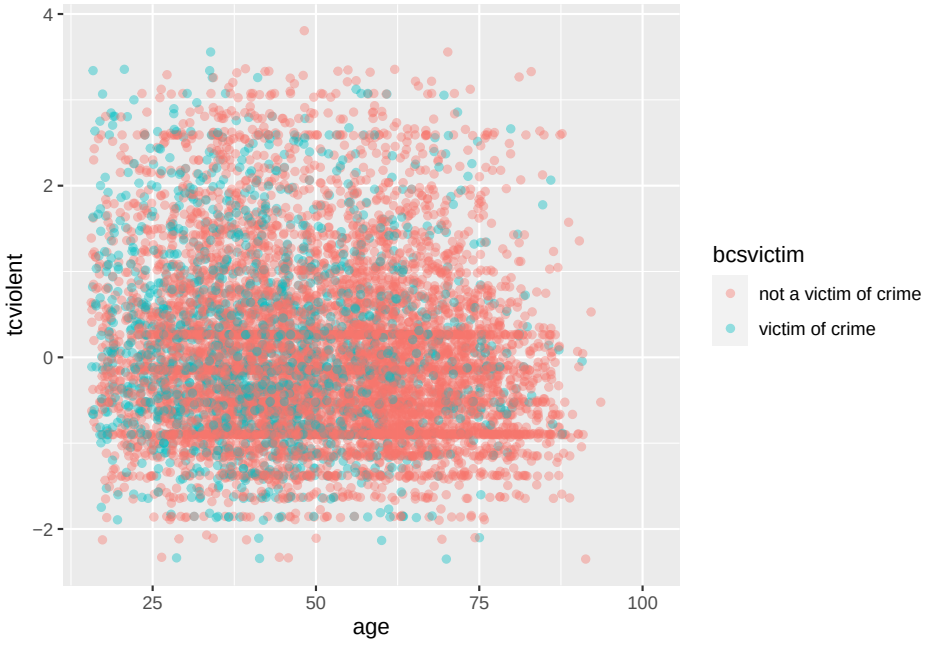
You could map the third variable to a different aesthetic (rather than colour). For example, you could map *lstat* to size of the points. This is called a **bubblechart**. The problem with this, however, is that it can make overplotting more acute sometimes.

```
ggplot(Boston, aes(x = medv, y = crim, size = lstat)) +
  geom_point() #You may want to add alpha for some transparency here.
```



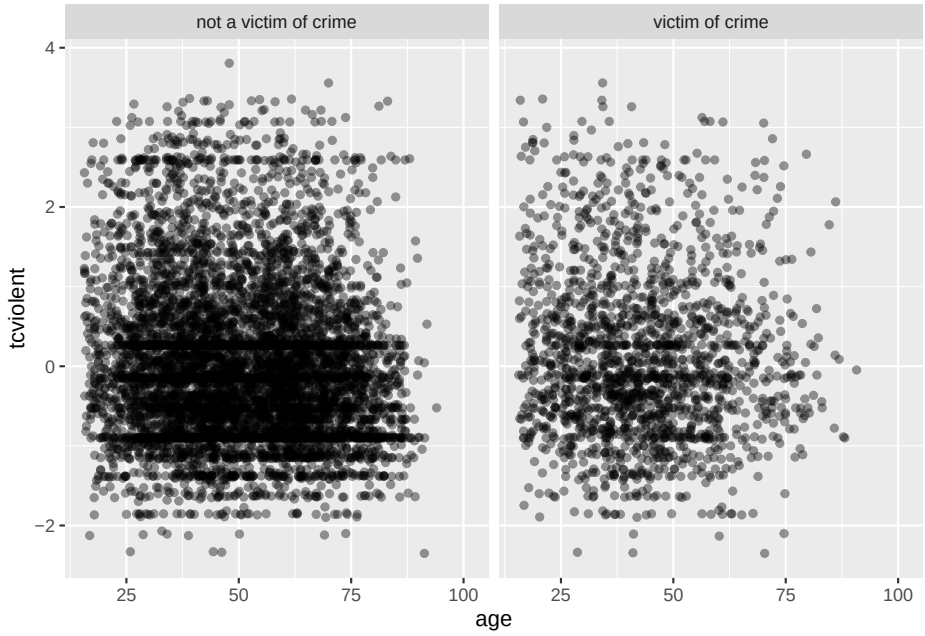
If you have larger samples and the patterns are not clear (as we saw when looking at the relationship between age and worry of violent crime) conditioning in a third variable can produce hard to read scatterplots (even if you use transparencies and jittering). Let's look at the relationship between worry for violent crime and age conditioned on victimisation during the previous year:

```
ggplot(BCS0708, aes(x = age, y = tcviolent, colour = bcsvictim)) +
  geom_point(alpha=.4, position="jitter")
```



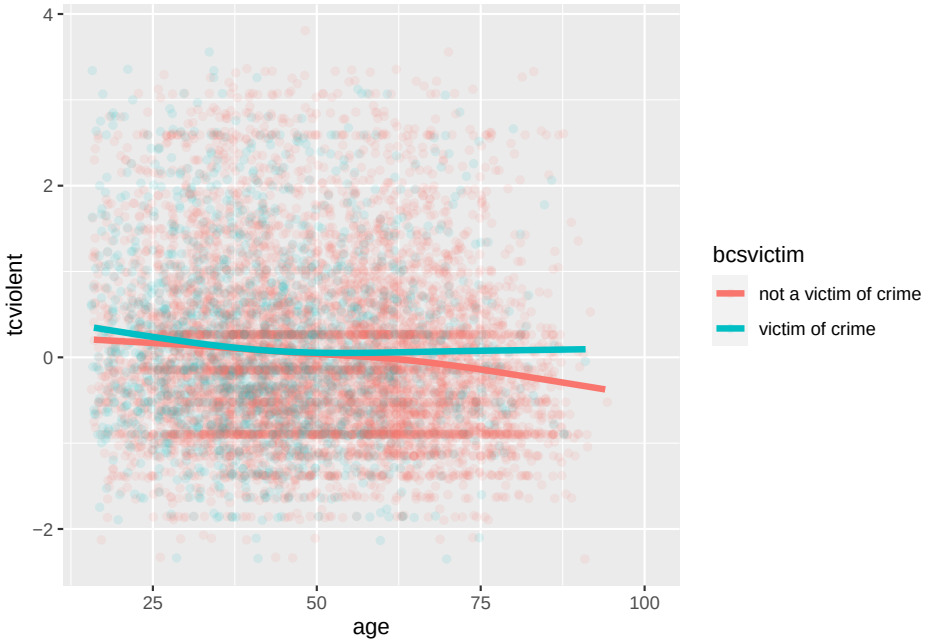
You can possibly notice that there are more green points in the left hand side (since victimisation tend to be more common among youth). But it is hard to read the relationship with age. We could try to use facets instead using `facet_grid`?

```
ggplot(BCS0708, aes(x = age, y = tcviolent)) +
  geom_point(alpha=.4, position="jitter") +
  facet_grid( .~ bcsvictim)
```



It is still hard to see anything, though perhaps you can notice the lower density the points in the bottom right corner in the facet displaying victims of crime. In a case like this may be helpful to draw a smooth line

```
ggplot(BCS0708, aes(x = age, y = tcviolent, colour = bcsvictim)) +
  geom_point(alpha=.1, position="jitter") +
  geom_smooth(size=1.5, se=FALSE)
```



What we see here is that for the most part the relationship of age and worry for violent crime looks quite flat, regardless of whether you have been a victim of crime or not. At least, for most people. However, once we get to the 60s things seem to change a bit. Those over 62 that have not been a victim of crime in the past year start to manifest a lower concern with crime as they age (in comparison with those that have been a victim of crime).

3.9 Scatterplot matrix

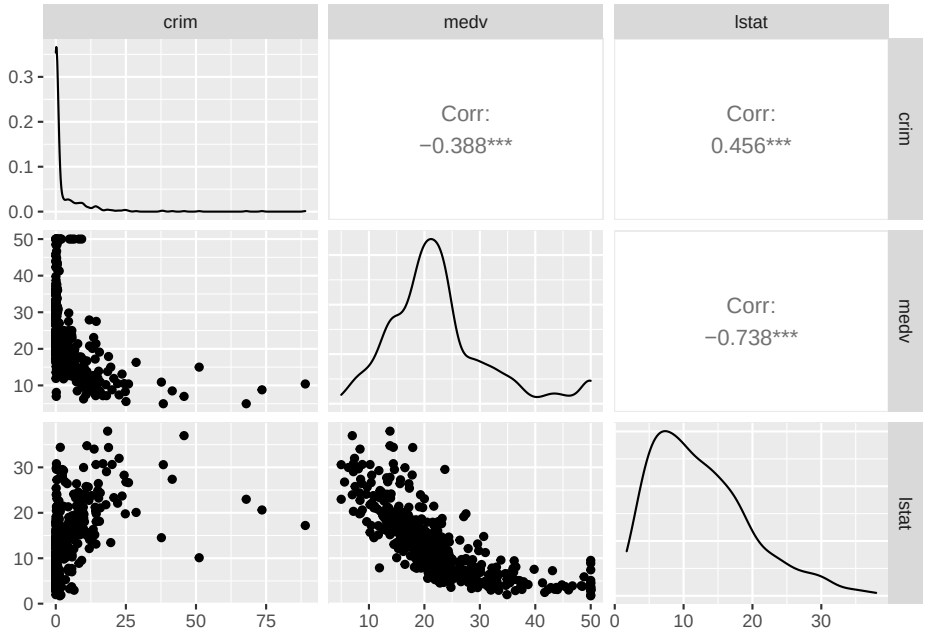
Sometimes you want to produce many scatterplots simultaneously to have a first peak at the relationship between the various variables in your data frame. The way to do this is by using a scatterplot matrix. There are some packages that are particularly good for this. One of them is `GGally`, basically an extension for `ggplot2`.

Not to overcomplicate things we will only use a few variables from the *Boston* dataset:

```
#I create a new data frame that only contains 4 variables included in the Boston dataset and I am
Boston_spm <- dplyr::select(Boston, crim, medv, lstat)
```

Then we load `ggpairs` and use run the scatterplot matrix using the `ggpairs` function:

```
library(GGally)
ggpairs(Boston_spm)
```

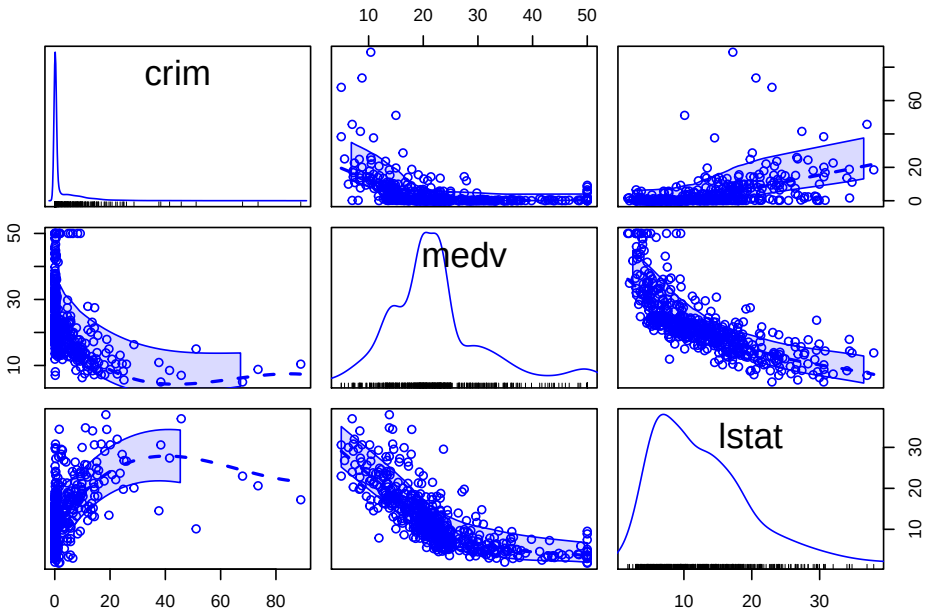


The diagonal set of boxes that go from the top left to the bottom right gives you the univariate density plot for each of the variables. So, for example, at the very top left you have the density plot for the *crim* variable. If you look underneath this one, you see a scatterplot between *crim* and *medv*. In this case *crim* defines the X axis and *medv* the Y axis, which is why it looks a bit different from the one we saw earlier. The labels in the top and the left tell you what variables are plot in each faceted rectangle. I the top right hand side of this matrix you see that the rectangles say "corr" and the give you a number. These numbers are **correlation coefficients**, which are a metric we use to indicate the strength of a relationship between two quantitative or numeric variables. The close to one (whether positive or negative) this value is the stronger the relationship is. The closer to zero the weaker the relationship. The stronger relationship here is between *crime* and *medv*. The fact that is negative indicates that as the values in one increase, the values in the other tend to decrease. So high values of crime correspond to low values of property prices -as we saw earlier. This coefficient is a summary number of this relationship. We will come back to it later on. For now keep in mind this metric only works well is the relationship should see in the scatterplot is well represented by a straight line. If the relationship is curvilinear it will be a very bad metric that you should not trust.

R gives you a lot of flexibility and there are often competing packages that aim to do similar things. So, for example, for a scatterplot matrix you could also

use the `spm` function from the `car` package.

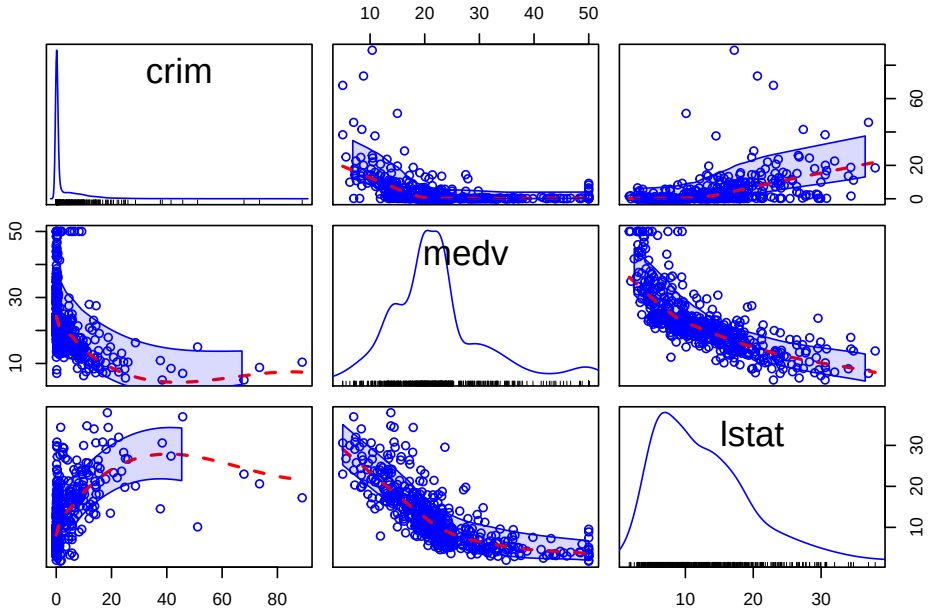
```
library(car)
spm(Boston_spm, regLine=FALSE)
```




```
#The regLine argument is used to avoid displaying something we will cover in week 8.
```

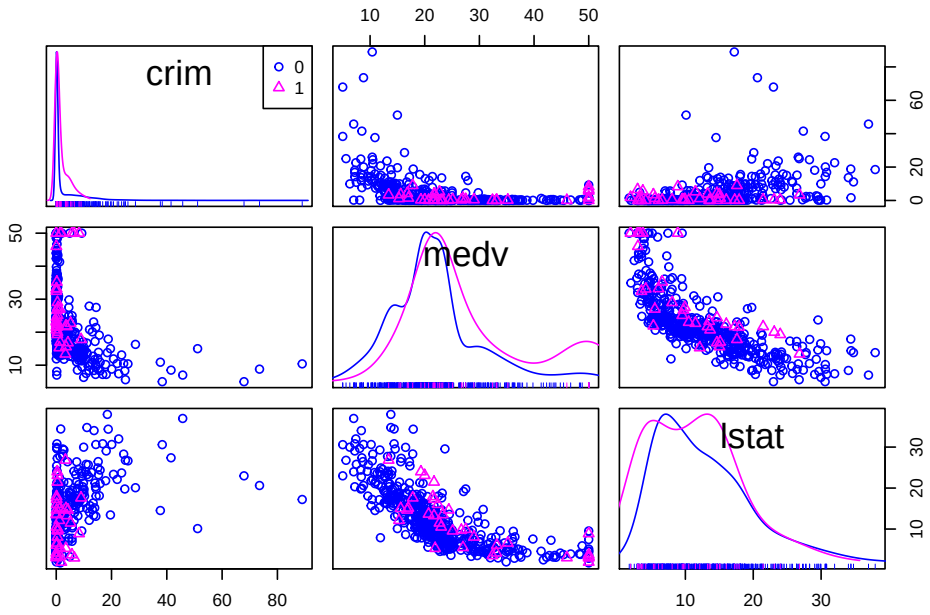
This is a bit different form the one below because rather than displaying the correlation coefficients, what you get is another set of scatterplot but with the Y and X axis rotated. You can see the matrix is symmetrical. So the first scatterplot that you see in the top row (second column from the left) shows the relationship between *medv* (in the X axis) and *crim* (in the Y axis). Which is the same relationship shown in the first scatterplot in the second row (first column), only here *crim* defines the X axis and *medv* the Y axis. In this scatterplot you can see, although not very well, that smoothed lines representing the relationship have been added to the plots.

```
library(car)
spm(Boston_spm, smooth=list(col.smooth="red"), regLine=FALSE)
```



And you can also condition in a third variable. For example, we could condition on whether the areas bound the Charles River (variable *chas*).

```
Boston_spm <- dplyr::select(Boston, crim, medv, lstat, chas)
spm(~crim+medv+lstat, data=Boston_spm, groups=Boston_spm$chas, by.groups=TRUE, smooth=
```



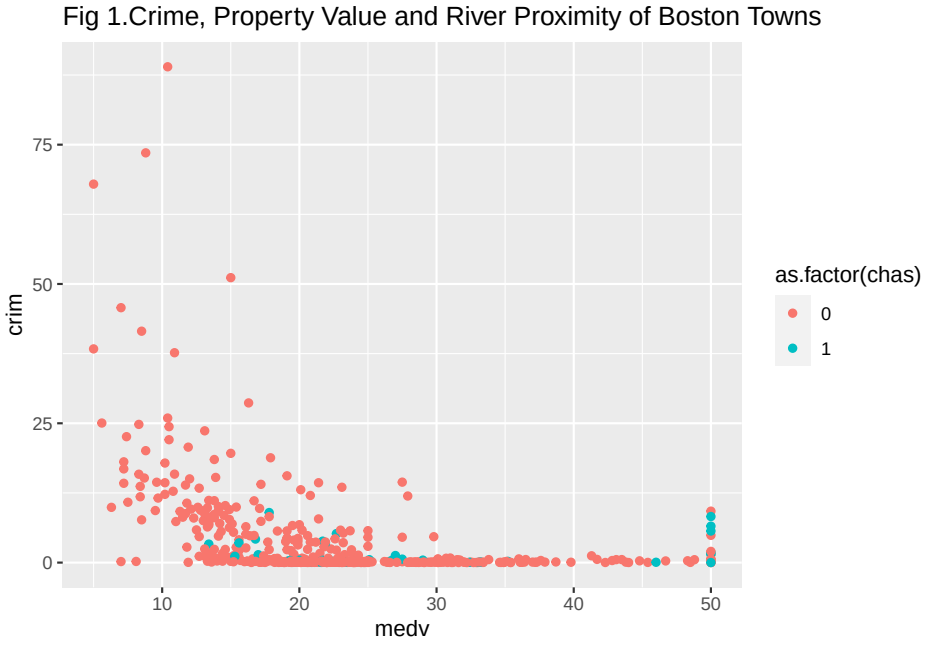
Getting results, once you get the knack of it, is only half of the way. The other,

and more important, half is trying to make sense of the results. What are the stories this data is telling us? R cannot do that for you. For this you need to use a better tool: **your brain** (scepticism, curiosity, creativity, a lifetime of knowledge) and what Kaiser Fung calls "numbersense".

3.10 Titles, legends, and themes in ggplot2

We have introduced a number of various graphical tools, but what if you want to customise the way the produce graphic looks like? Here I am just going to give you some code for how to modify the titles and legends you use. For adding a title for a `ggplot` graph you use `ggtitle()`.

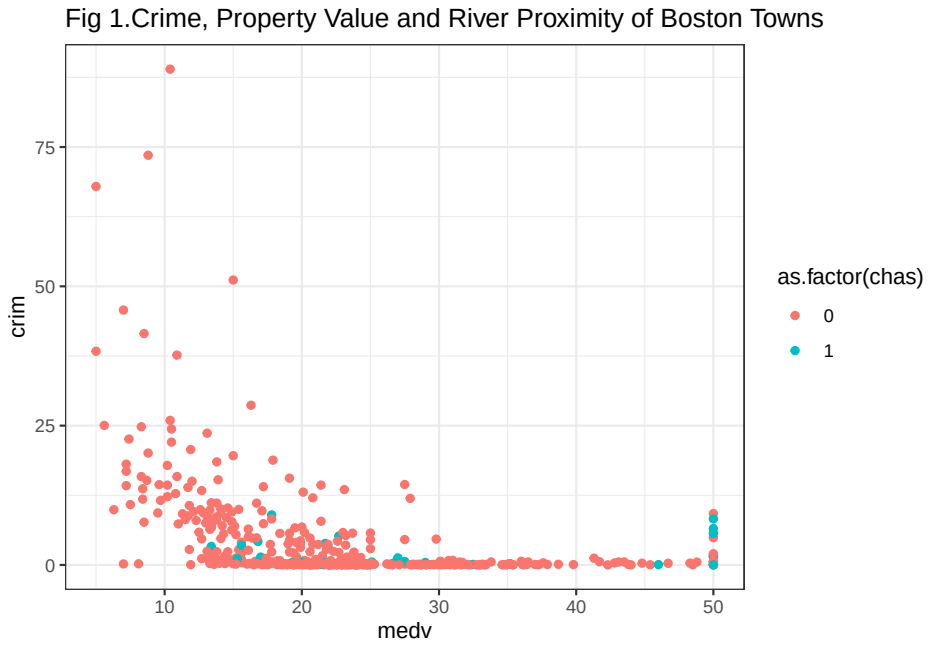
```
#Notice how here we are using an additional function to ask R to treat the variable chas, which
ggplot(Boston, aes(x = medv, y = crim, colour = as.factor(chas))) +
  geom_point() +
  ggtitle("Fig 1.Crime, Property Value and River Proximity of Boston Towns")
```



If you don't like the default background theme for `ggplot` you can use a theme as discussed at the start, for example with creating a black and white background by adding `theme_bw()` as a layer:

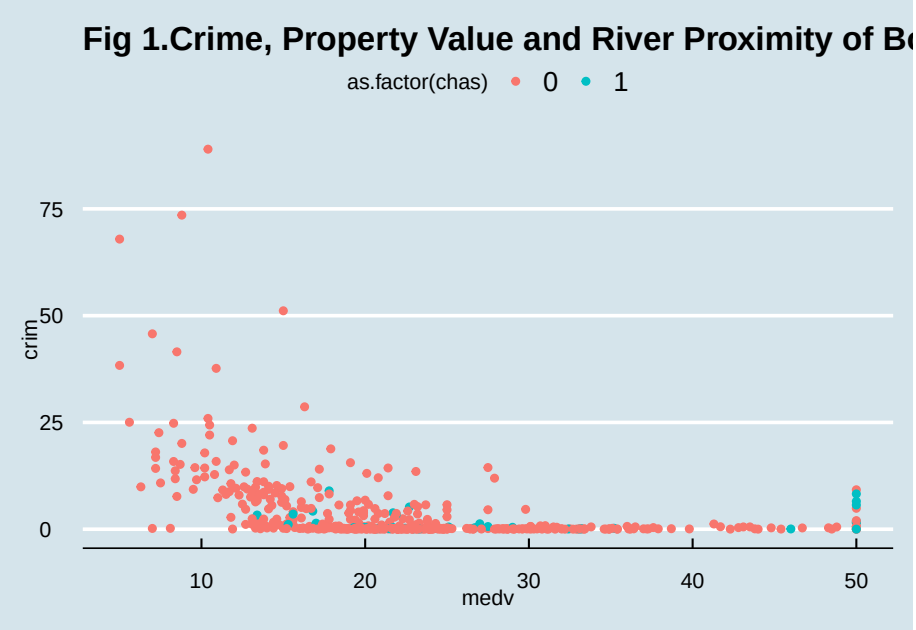
```
ggplot(Boston, aes(x = medv, y = crim, colour = as.factor(chas))) +
  geom_point() +
```

```
  ggtitle("Fig 1.Crime, Property Value and River Proximity of Boston Towns") +
  theme_bw()
```



As we said earlier `ggthemes` gives you additional themes you could use. So, for example, you can use the style inspired by *The Economist* magazine.

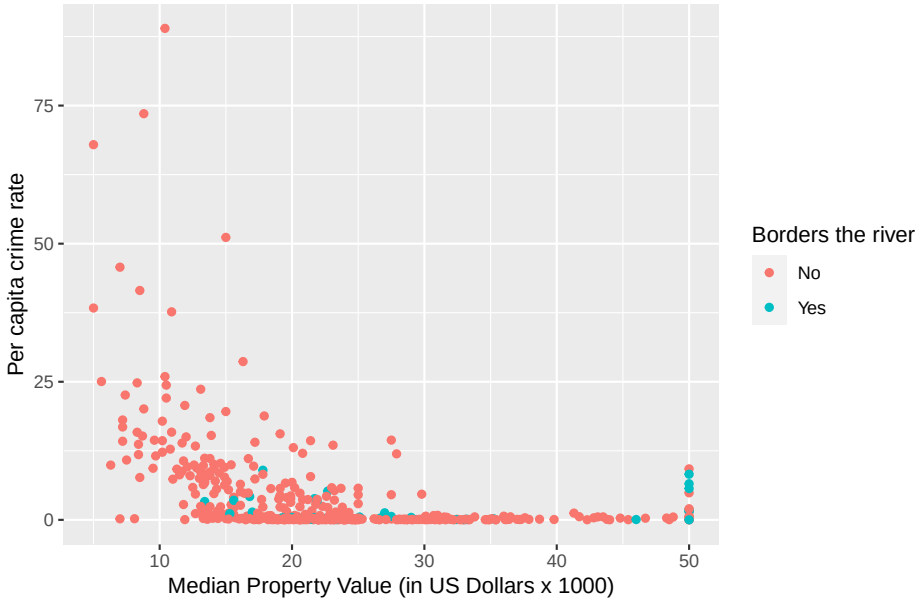
```
library(ggthemes)
ggplot(Boston, aes(x = medv, y = crim, colour = as.factor(chas))) +
  geom_point() +
  ggtitle("Fig 1.Crime, Property Value and River Proximity of Boston Towns") +
  theme_economist()
```



Using `labs()` you can change the text of axis labels (and the legend title), which may be handy if your variables have cryptic names. Equally you can manually name the labels in a legend. The value for *chas* are 0 and 1. This is not informative. We can change that.

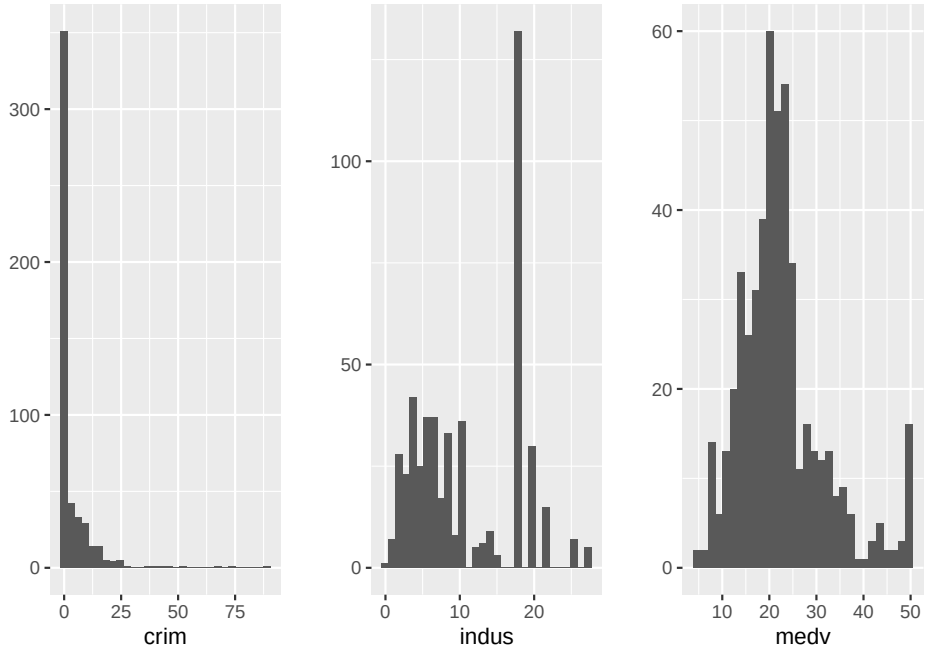
```
ggplot(Boston, aes(x = medv, y = crim, colour = as.factor(chas))) +
  geom_point() +
  ggtitle("Fig 1.Crime, Property Value and River Proximity of Boston Towns") +
  labs(x = "Median Property Value (in US Dollars x 1000)",
       y = "Per capita crime rate",
       colour = "Borders the river") +
  scale_colour_discrete(labels = c("No", "Yes"))
```

Fig 1.Crime, Property Value and River Proximity of Boston Towns



Sometimes you may want to present several plots together. For this the `gridExtra` package is very good. You will first need to install it and then load it. You can then create several plots and put them all in the same image.

```
#You may need to install first with install.packages("gridExtra")
library(gridExtra)
#Store your plots in various objects
p1 <- qplot(x=crim, data=Boston)
p2 <- qplot(x=indus, data=Boston)
p3 <- qplot(x=medv, data=Boston)
#Then put them all together using grid.arrange()
grid.arrange(p1, p2, p3, ncol=3) #ncol tells R we want them side by side, if you want
```

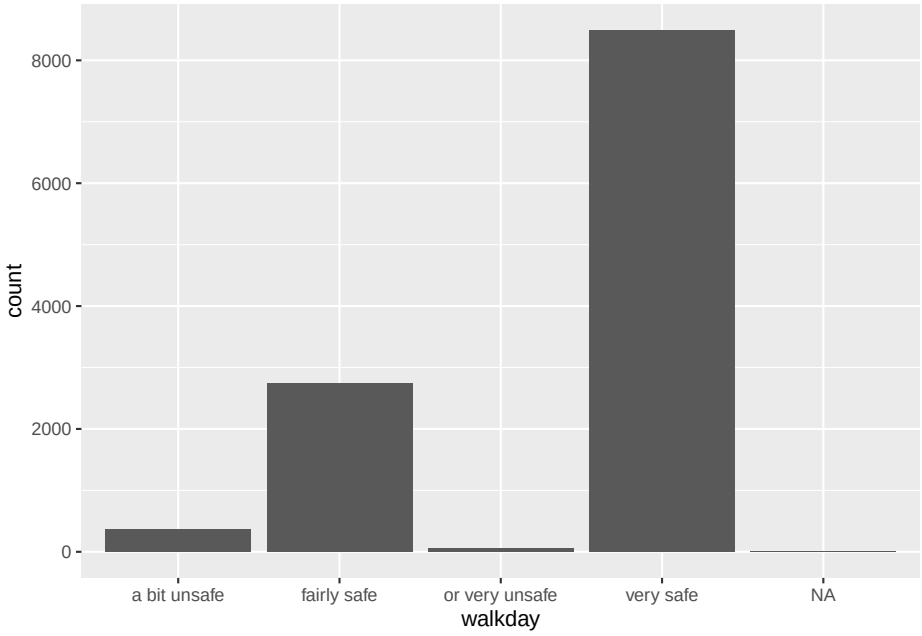


We don't have time to get into the detail of all the customisation features available for `ggplot2`. You can find some additional solutions in the cookbook put together by the data journalists at the BBC or in Kieran Healy free online book.

3.11 Plotting categorical data: bar charts

You may be wondering what about categorical data? So far we have only discussed various visualisations where at least one of your variables is quantitative. When your variable is categorical you can use bar plots (similar to histograms). We map the factor variable in the aesthetics and then use the `geom_bar()` function to ask for a bar chart.

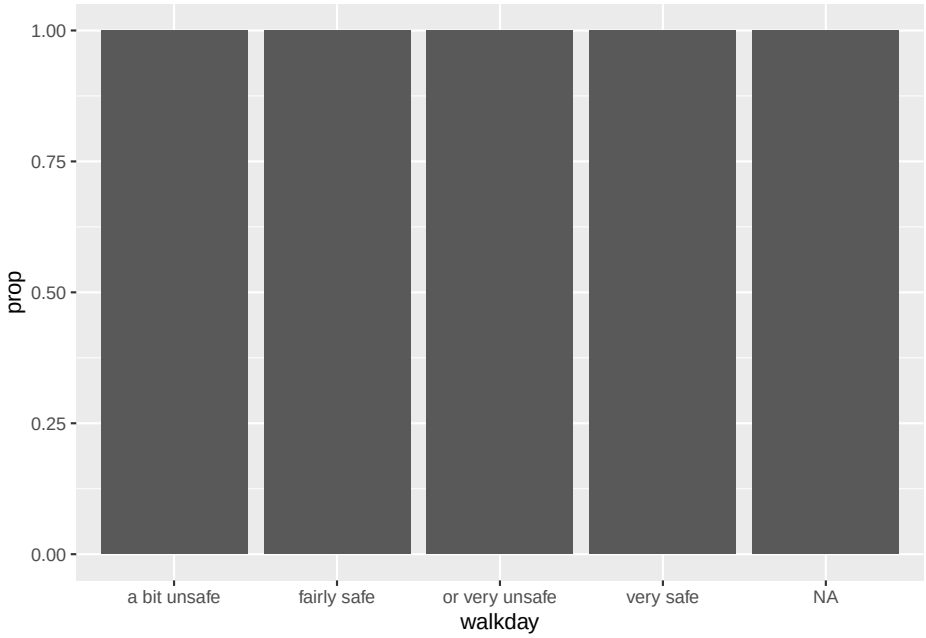
```
ggplot(BCS0708, aes(x=walkday)) +
  geom_bar()
```



You can see in the Y axis the label *count*. This is not a variable in your data. When you run a `geom_bar` like this you are invoking as a hidden default a call to the `stat_` function. In this case what is happening is that this function is *counting* the number of cases in each of the levels of *walkday* and this count is what is used to map the height of each of the bars.

The `stat_` function apart from counting the cases in each levels computes their relative frequency, their proportion, and stores this information in a temporal variable called *..prop...* If we want this variable to be represented in the Y axis we can change the code as shown below:

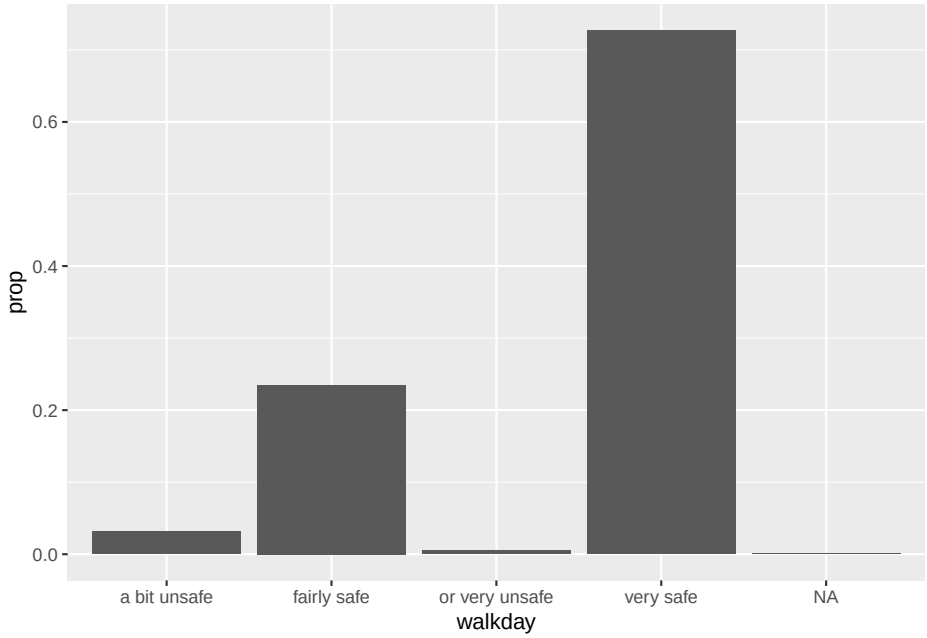
```
ggplot(BCS0708, aes(x=walkday)) +
  geom_bar(mapping = aes(y = ..prop..))
```



As Kieran Helay (2018) indicates:

"The resulting plot is still not right. We no longer have a count on the y-axis, but the proportions of the bars all have a value of 1, so all the bars are the same height. We want them to sum to 1, so that we get the number of observations per" (level) "as a proportion of the total number of observations. This is a grouping issue again... we need to tell ggplot to ignore the x-categories when calculating denominator of the proportion, and use the total number observations instead. To do so we specify group = 1 inside the `aes()` *call. The value of 1 is just a kind of "dummy group" that tells ggplot to use the whole dataset when establishing the denominator for its prop calculations."*

```
ggplot(BCS0708, aes(x=walkday)) +
  geom_bar(mapping = aes(y = ..prop.., group = 1))
```

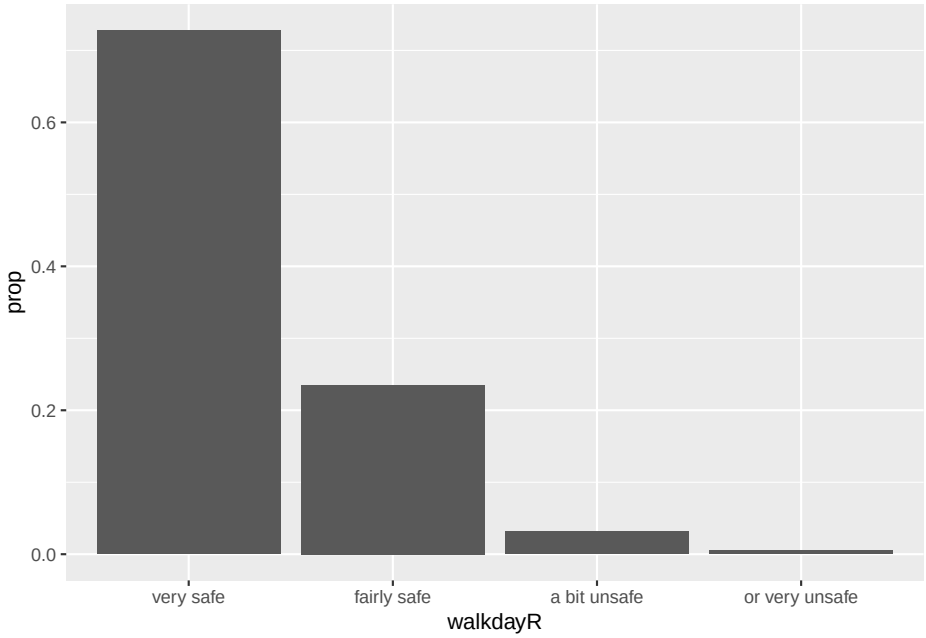


Unfortunately, the levels in this factor are ordered by alphabetical order, which is confusing. We can modify this by reordering the factors levels first -click here for more details. You could do this within the `ggplot` function (just for the visualisation), but in real life you would want to sort out your factor levels in an appropriate manner more permanently. As discussed last week, this is the sort of thing you do as part of pre-processing your data. And then plot.

```
#Print the original order
print(levels(BCS0708$walkday))
```

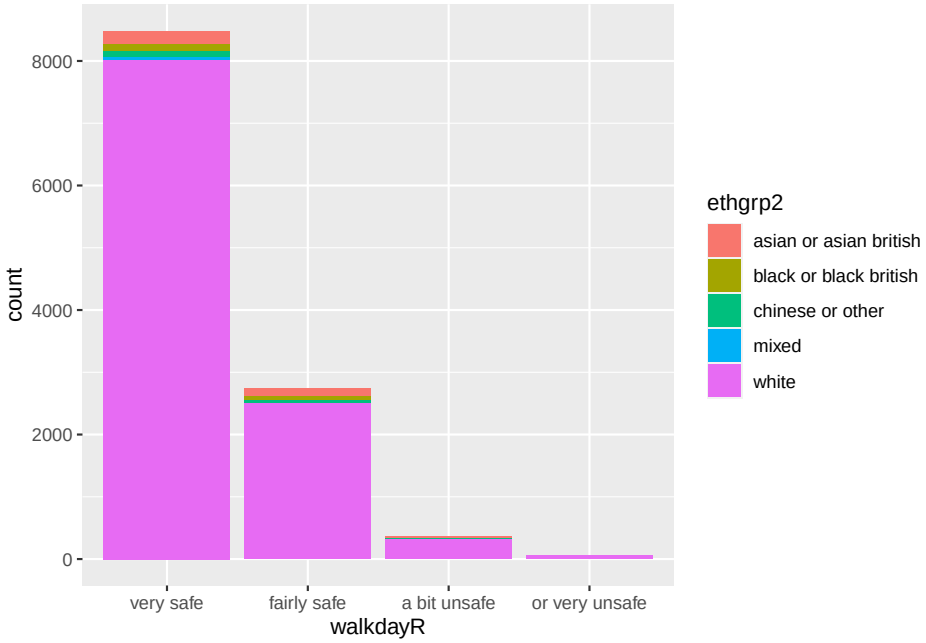
```
## NULL
```

```
#Reordering the factor levels from very safe to very unsafe (rather than by alphabet).
BCS0708$walkdayR <- factor(BCS0708$walkday, levels=c('very safe',
  'fairly safe','a bit unsafe','or very unsafe'))
#Plotting the variable again (and subsetting out the NA data)
ggplot(subset(BCS0708, !is.na(walkdayR)), aes(x=walkdayR)) +
  geom_bar(mapping = aes(y = ..prop.., group = 1))
```



We can also map a second variable to the aesthetics, for example, let's look at ethnicity in relation to feelings of safety. For this we produce a **stacked bar chart**.

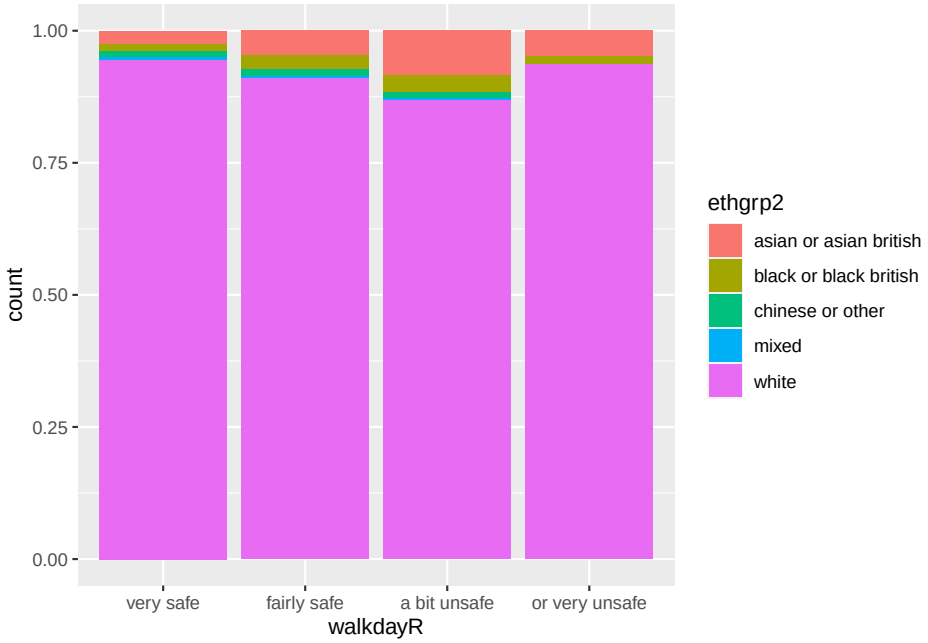
```
bcs_bar <-filter(BCS0708, !is.na(walkdayR), !is.na(ethgrp2))
ggplot(data=bcs_bar, aes(x=walkdayR, fill=ethgrp2)) +
  geom_bar()
```



These sort of stacked bar charts are not terribly helpful if you are interested in understanding the relationship between these two variables. It is hard to see if any group is proportionally more likely to feel safe or not.

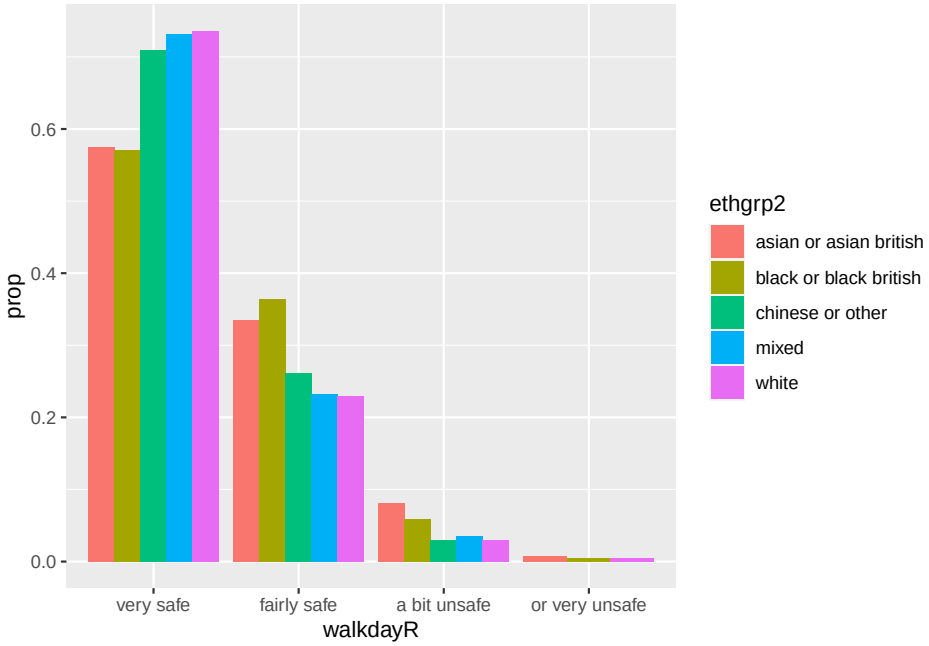
Instead what you want is a different kind of stacked bar chart, that gives you the proportion of your "explanatory variable" (ethnicity) within each of the levels of your "response variable" (here feelings of safety).

```
ggplot(data=bcs_bar, aes(x=walkdayR, fill=ethgrp2)) +
  geom_bar(position = "fill")
```



Now we can more easily compare proportions across groups, since all the bars have the same height. But it is more difficult to see how many people there are within each level of the X variable.

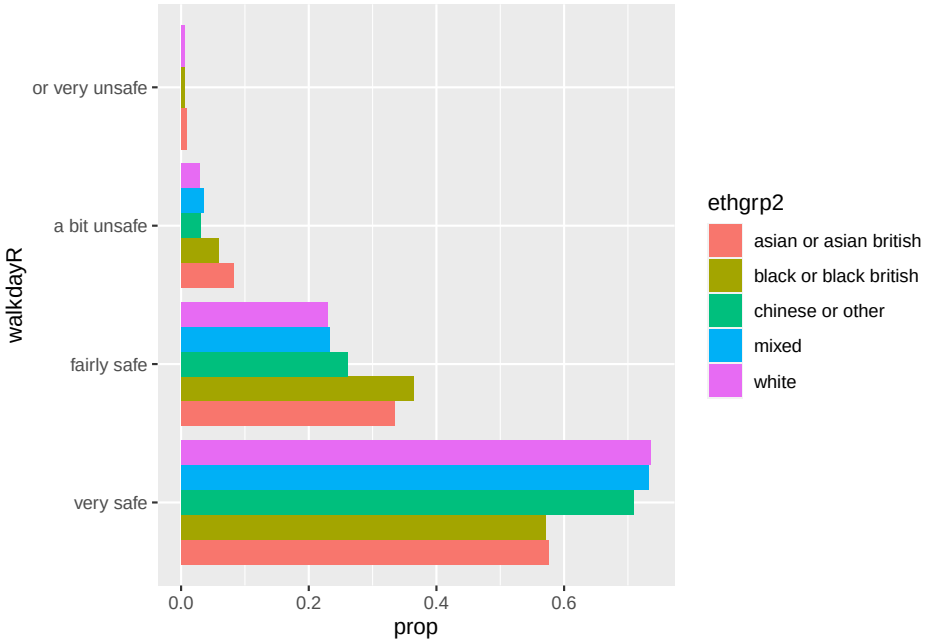
```
p <- ggplot(data=bcs_bar, aes(x=walkdayR, fill=ethgrp2)) + geom_bar(position = "dodge",
             mapping = aes(y = ..prop.., group = ethgrp2))
p
```



Now we have a bar chart where the values of ethnicity are broken down across the levels of fear of crime, with a proportion showing on the y-axis. Looking at the bars you will see that they do not sum to one within each level of fear. Instead, the bars for any particular ethnicity sum to one across all the levels of fear. You can see for example that nearly 75% of the White respondents are in the "Very Safe" level, whereas for example less than 60% of the Black respondents feel "Very Safe".

Sometimes, you may want to flip the axis, so that the bars are displayed horizontally. You can use the `coord_flip()` function for that.

```
#First we invoke the plot we created and stored earlier and then we add an additional
p + coord_flip()
```



You can also use `coord_flip()` with other `ggplot` plots (e.g., boxplots).

A particular type of bar chart is the divergent stacked bar chart, often used to visualise **Likert scales**. You may want to look at some of the options available for it via the HH package, the likert package or sjPlot. But we won't cover them here in detail.

Keep in mind that knowing *how* to get R to produce a particular visualisation is only half the job. The other half is knowing *when* to produce a particular kind of visualisation. This blog, for example, discusses some of the problems with stacked bar charts and the exceptional circumstances in which you may want to use them.

There are other tools sometimes used for visualising categorical data. Pie charts is one of them. However, as mentioned at the beginning, many people advocate strongly against using pie charts and therefore this is the only pie chart you will see in this course:

What I would use instead are **waffle charts**. They're super easy to make with the "waffle" package but I don't think there's time for them at this point, but look into them here.

3.12 Further resources

By now you should know the path to data analysis wisdom will take time. The good news is that we live in a time where there are multiple (very often free)

Figure 3.1: cute pie chart

resources to help you along the way. The time where this knowledge was just the preserve of a few is long distant. If you are a motivated and disciplined person there is a lot that you can do to further consolidate and expand your data visualisation skills without spending money. I already recommended you the excellent `ggplot2` online documentation. Here we just want to point you to a a few useful resources that you can pursue in the future.

First **MOOCS (Massive Online Open Courses)**. There are loads of useful MOOCS that provide training in data visualisation for free (well, you can pay if you want a certificate, but you can also use the resources and learn for free).

- Information visualisation. This course offers an introduction to some more advanced concepts to those presented in our tutorial. It also covers more advanced software for visualisations in the web.

- You may also want to keep your eyes open for any new edition of Alberto Cairo's MOOC on "Data Visualisation for Storytelling and Discovery" which covers not the software but the principles and theory of good static and interactive visualisation. But you can find the free materials for the course posted in the linked site (including the videolectures).

Second, **online tutorials**. One of the things you will also soon discover is that R users are keen to write "how to" guides in some of the 750 R devoted blogs or as part of Rpubs. Using Google or similar you will often find solutions to almost any problem you will encounter using R. For example, in this blog there are a few tutorials that you may want to look to complement my own:

- Cheatsheet for visualising scatterplots
- Cheatsheet for visualizing distributions
- Cheatsheet for barplots

Third, *blogs on data visualisation.* If you use Feedly or a similar blog aggregator, you should add the following blogs to your reading list. They are written by leading people in the field and are full of useful advice: + Flowing Data + The Functional Art + Visual Business Intelligence + chartsnthings + Data Revelations

Fourth, **resources for visualisations we don't have the time to cover**. R is way ahead some of the more popular data analysis programs you may be more familiar with (e.g SPSS or Excel). There's many things you can do.

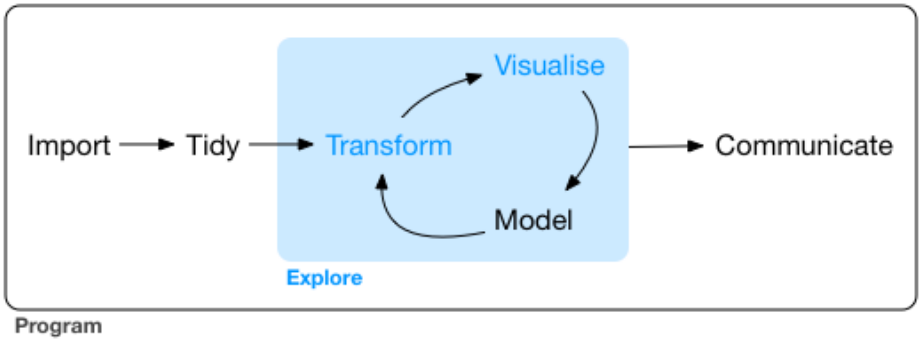
For example, if you like maps, **R can also be used to produce visualisations of spatial data**. There are various resources to learn how to do this and we teach this in our *Crime Mapping* module in the third year.

Chapter 4

Refresher on descriptive statistics & data carpentry

4.1 Introduction

We have already introduce the typical workflow of data analysis:



In this figure produced by Hadley Wickham you can see the first stage of data analysis involve importing your data, getting it into a tidy format, and then doing some transformations so that you get the data in good shape for analysis. There is a famous and possibly false statistic that says that 80% of an analyst time is often devoted to this kind of operations. Although the statistic is likely made up, the truth is that the experience of many analysts resonates with it. So, you should not underestimate data carpentry or data wrangling (as these processes are often called) as a part of your analysis.

For various decades, social scientists of a quantitative persuasion worked primarily with survey data (for an excelent history of how this come to be you can read this book), which came in rather tidy formats, but often required some transformations. Today we are more likely to rely on "big" and other new forms of data

(from the web, administrative sources, or a variety of sensors) that may require more significant processing before we can do any analysis with it. Think, for example, of data from online vendors of drugs available in the Dark Net. Some people talk of the advent of a new computational social science around these new methods. This kind of data, indeed, opens avenues for research we could only dream of in the past as argued by some of our colleagues. But getting this kind of data requires the development of new skills (e.g, web scrapping) and generally requires more processing before they are tidy and ready for analysis.

R is particularly well suited for this new world. In this module we only work with survey data, which tends to be tidier and easier to work with in the context of an introductory course unit. But even when working with this kind of data you often have to think hard about required tidying and transformations before you can start your analysis.

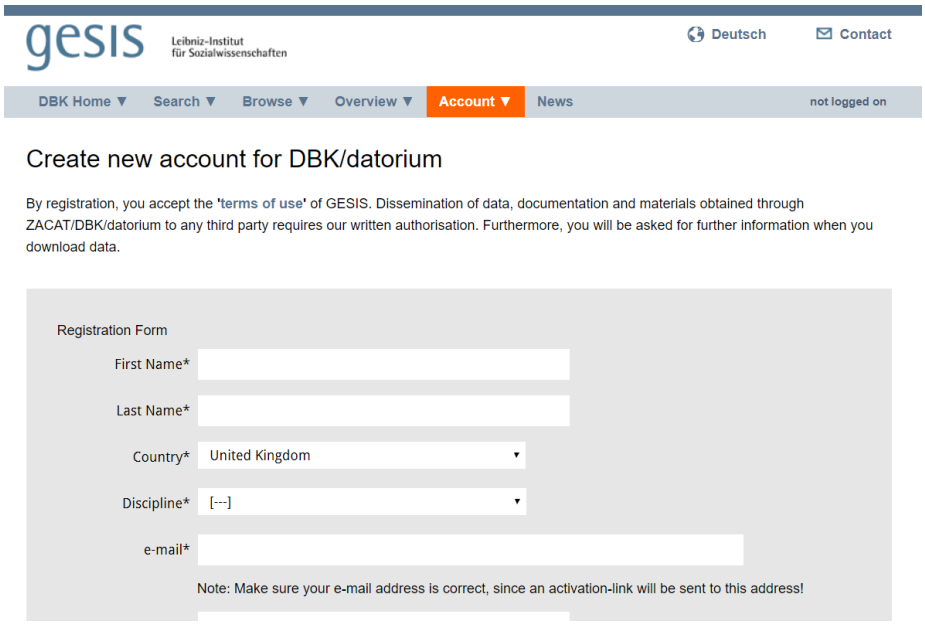
In this module we expect you to download a survey dataset for analysis. These datasets are already rather tidy and have been professionally cleaned and prepared for analytical consumption. But you may still have to select cases and variables that are appropriate for your own analysis. Also, you likely will need to generate new variables or change existing ones for various reasons. It's a rare data set in which every variable you need is measured directly. Examples of things you may need to do include:

- Combine various variables into a new one (eg., computing a rate)
- Reduce the number of levels in a categorical variable
- Format the variable to assign it to a more appropriate class if this is called for (e.g., character into factor). You will need to format date variables as dates, numerical variables as numbers, etc.).
- Recode values identifying missing cases as NA.
- Labelling all variables and categorical values so you don't have to keep looking them up.
- Change the labels associated with the levels of a categorical variable.

4.2 A template for your assignment

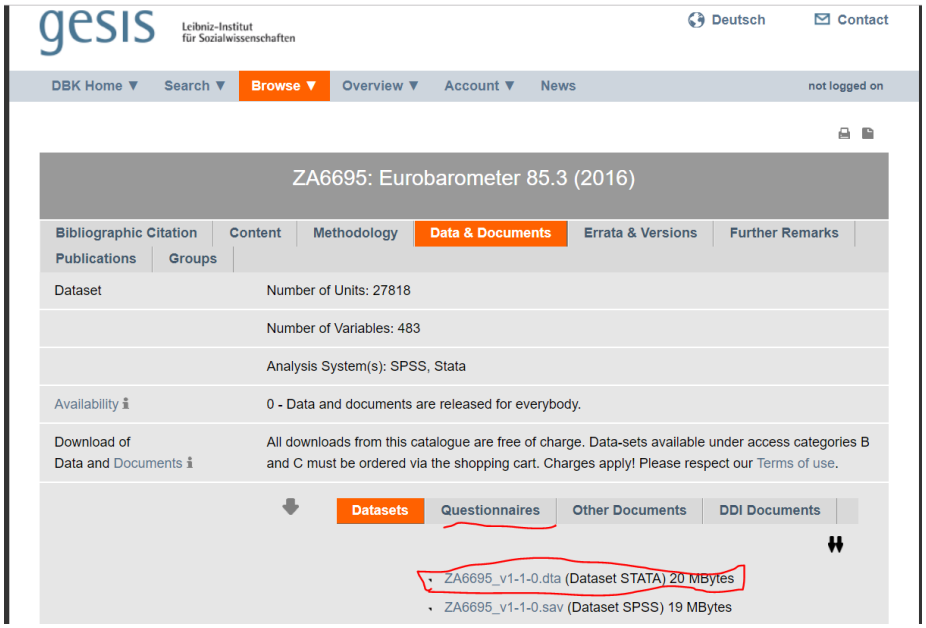
For this practical we are going to proceed as if we had an assignment similar to your final assignment for this module. Only that we will use different data, specifically data from a *Eurobarometer*. Eurobarometers are opinion polls conducted regularly on behalf of the European Commission since 1973. Some of them ask the same questions over time to evaluate changes in European's views in a variety of subjects (standard Eurobarometers). Others are focused on special topics and are conducted less regularly (special Eurobarometers). They are a useful source of datasets that you could use, for exampple, for your undergraduate dissertation.

The data from these surveys is accessible through the data catalogue of GESIS, a data warehouse at the *Leibniz Institute for the Social Sciences* in Germany. For downloading data from GESIS you have to register with them (following the registration link) here. Once you activate your registration you should be able to access the data at GESIS.



GESIS has a section of their website devoted to the Eurobarometer survey. You can access that section here. You could navigate this webpage to find the data we will be using for this tutorial, but to save time I will provide you a direct link to it. We will be using the special Eurobarometer 85.3 form 2016. This survey among other things asked Europeans about their views on gender violence.

You can access the data for this eurobarometer here.



You will see here that there are links to the files with the data in SPSS and STATA format. You can also see a tab where you could obtain the questionnaire for the survey. Once you are registered download the STATA version of the file and also an English version of the questionnaire. Make sure you place the file

in your working directory. Once you do this you should be able to use the code we are using in this session.

First, we will load the data into our session. Since the data is in STATA format we will need to read the data into R using the `haven` package. Specifically, we will use the `read_dta` function for importing STATA data into R. As an argument we need to write the name of the file with the data (and if it is not in your working directory, the appropriate path file).

```
library(haven)
eb85_3 <- read_dta("ZA6695_v1-1-0.dta")
dim(eb85_3)
```

```
## [1] 27818   483
```

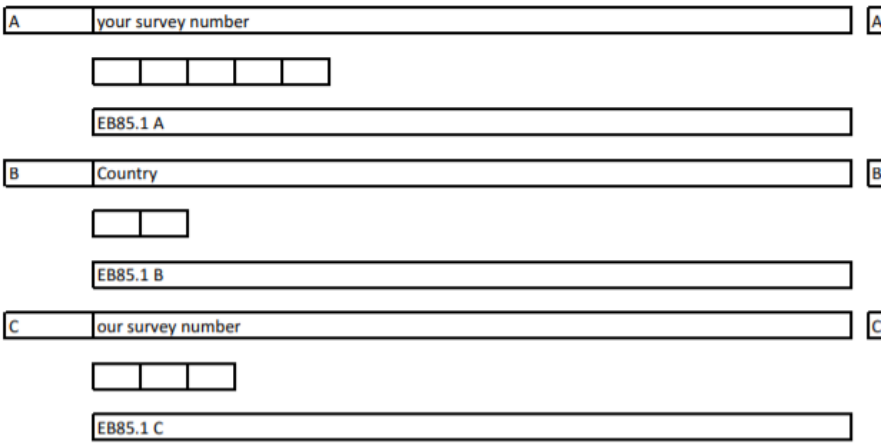
We can see there are over 27000 cases (survey participants) and 483 variables.

4.3 Thinking about your data: filtering cases

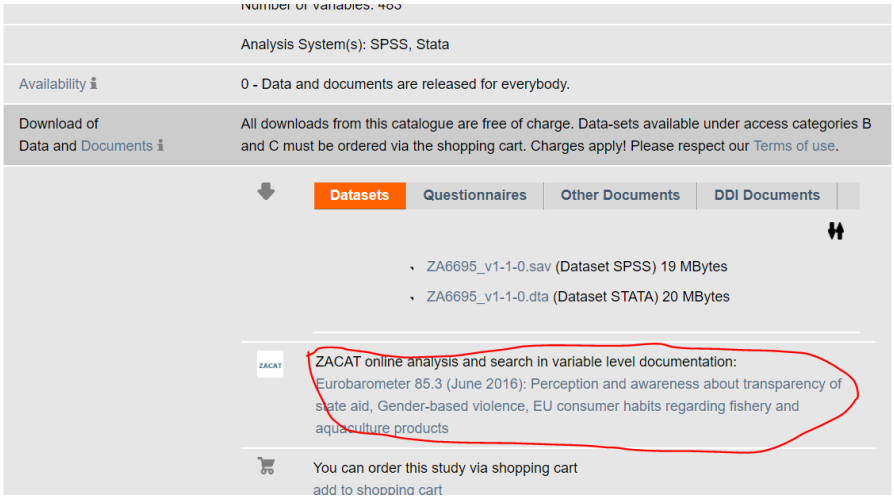
For the final coursework assignment, you will need to download other datasets but the process will be similar in that once you have the datasets you will need to think about what cases and what variables you want to work with. Imagine that we had to write our final assignment using this dataset and that we had to write a report about attitudes to sexual violence.

First, we would need to think if we wanted to use all cases in the data or only a subset of the cases. For example, when using something like the Eurobarometer we would need to consider if we are interested in exploring the substantive topic across Europe or only for some countries. Or alternatively you may want to focus your analysis only on men attitudes to sexual violence. In a situation like this you would need to filter cases. This decision needs to be guided by your theoretical interests and your driving research question.

So, for example, if we only wanted to work with the UK sample we would need to figure out if there is a variable that identifies the country in the dataset. In the questionnaire you can see indeed that there is such a variable:



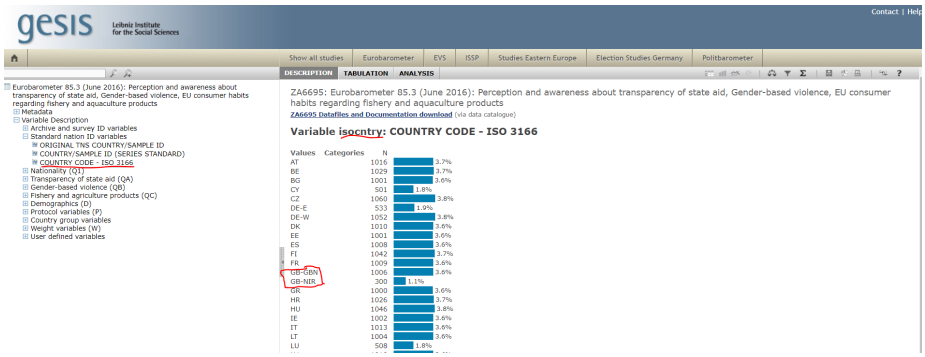
What we do not know is how this variable is named in the dataset. For this we need to look at the *codebook*. In this case, we can look at the interactive facility provided for GESIS for online data analysis, which provides an interactive online codebook for this dataset. You can access this facility in the link highlighted in the image below:



You can access similar applications for the two datasets that you need to use for the final coursework assignment. If you will use the ESS you can use an online facility here and if you are using the CSEW you can access a similar tool here.

Let's explore the online facility for the Eurobarometer. If we expand the menus in the left hand side by clicking in variable description, and then in standard nation id variables, you will see there is a variable that provides a "country code". This must be it. click on it and then you will see in the right hand side some information about this variable. You see the name of this variable

(as it will appear in the dataset) *isocntry* and we can see this variable uses ISO 3166 codes to designate the country. This is an international standard set of abbreviations for country names. For the UK these codes are "GB-GBN" and "GB-NIR" (for Northern Ireland).



Now that we have this information we can run the code to select only the cases that have these values in the dataset. For doing something like that we would use the `dplyr::filter` function. We used the `filter` function in week 2. You can read more about it in the `dplyr` vignette.

```
library(dplyr)
#First, let's see what type of vector isocntry is
class(eb85_3$isocntry)
```

```
## [1] "character"
```

```
uk_eb85_3 <- filter(eb85_3, isocntry %in% c("GB-GBN", "GB-NIR"))
```

The variable *isocntry* is a character vector with codes for the different participating countries. Here we want to select all the cases in the survey that have either of two values in this vector (GB-GBN or GB-NIR). Since these values are text we need to use quotes to wrap them up. Because we are selecting more than one value we cannot simply say `isocntry == "GB-BGN"`. We also need the cases from Northern Ireland. So, we use a particular operator introduced by `dplyr` called the piping operator (`%in%`). The piping operator is essentially saying to R *"and then"*. Basically, here we are creating a vector with the values we want *and then* asking R to look at those values within tne list (containing the right labels) in the the *isocntry* vector so that we can filter everything else out.

If you run this code you will end up with a new object called `uk_eb85_3` that only has 1306 observations. We have now a dataset that only has the British participants.

4.4 Selecting variables: using `dplyr::select`

Perhaps for your coursework you define your essay question in such a way that you do not need to do any a priory filtering. Perhaps, for the sake of this example, we decide to do an analysis that focuses on looking at attitudes toward sexual violence for all of Europe and for all participants. Yet, you won't be using 483 variables for certain. Among other reasons because our guidelines for the essay suggest you use fewer variables. But more generally because typically your theoretical model will tell you that some things matter more than others.

The first thing you need to do is to think about what variables you are going to use. This involves first thinking about what variables are available in the dataset that measure your outcome of interest but then also consider what your theory of attitudes to gender violence say (this generally will include things that are not measured in the survey, such is life!).

For the sake of this exercise we are assuming the thing you are interested in explaining or better understanding is attitudes regarding sexual violence. So, before anything else you would need to spend some time thinking about how this survey measures these attitudes. You would need to screen the questionnaire and the codebook to identify these variables and their names in the dataset. Have a look at the questionnaire. The questions about gender violence start at the bottom of page 7. Which of these questions are questions about attitudes towards sexual violence?

Once you have all of this you would need to think about which of these survey questions and items make more sense for your research question. This is something where you will need to use your common sense but also your understanding of the literature in the topic. Criminologists and survey researchers spend a lot of time thinking about what is the best way of asking questions about topics or concepts of interest. They often debate and write about this. In data analysis measurement is key and is the process of systematically assigning numbers to objects and their properties, to facilitate the use of mathematics in studying and describing objects and their relationships. So, as part of your essay, you will need to consider what do researchers consider are good questions to measure, to tap into, the abstract concepts you are studying.

There are many items in this survey that relate to this topic, but for purposes of continuing our illustration we are going to focus on the answers to question *QB10*. This question asks respondents to identify in what circumstances may be justified to have sexual intercourse without consent. The participants are read a list of items (e.g., "flirting before hand") and they can select various of them if so they wish.

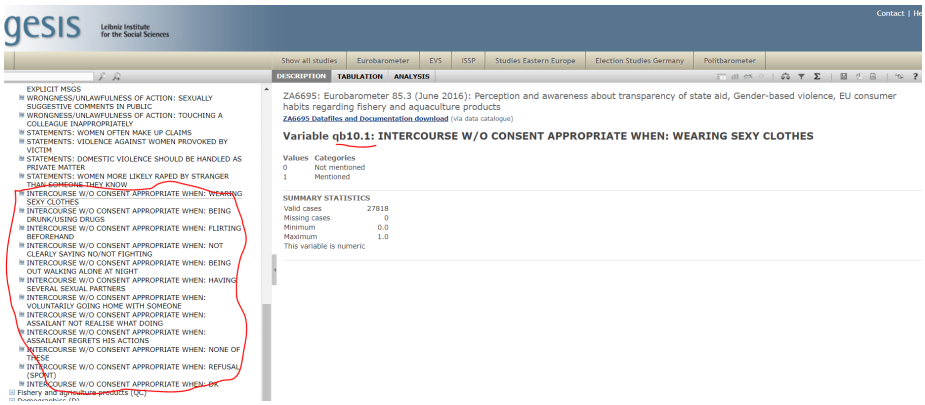
QB10	Some people believe that having sexual intercourse without consent may be justified in certain situations. Do you think this applies to the following circumstances?

(SHOW SCREEN - READ OUT - MULTIPLE ANSWERS POSSIBLE)

EB85.3 Bilingu

Wearing revealing, provocative or sexy clothing	1,
Being drunk or using drugs	2,
Flirting beforehand	3,
Not clearly saying no or physically fighting back	4,
Being out walking alone at night	5,
Having several sexual partners	6,
Voluntarily going home with someone, for example after a party or date	7,
If the assailant does not realise what they were doing	8,
If the assailant regrets his actions	9,
None of these	10,
Refusal (SPONTANEOUS)	11,
Don't know	12,

Ok, so we have now selected our variable. Say that we have done so on the basis of our understanding of the literature. Next we need to identify these variables in the dataset. What name is associated with this variable? Let's look at the online interactive facility.



Damn! We have one question but several variables! This is common when the question in the survey allows for multiple responses. Typically when this is read into a dataset, survey researchers create a variable for each of the possible multiple responses. If the respondent identified one of those potential responses they will be assigned a "yes" or a "1" for that column. If they did not they will be assigned a "no" or a "0". Let's see how this was done in this case:

```
class(eb85_3$qb10_1)
```

```
## [1] "haven_labelled" "vctrs_vctr"     "double"
```

This is a vector labelled by haven. We could see what labels were used using the `attributes` function.

```
attributes(eb85_3$qb10_1)
```

```
## $label
## [1] "INTERCOURSE W/O CONSENT APPROPRIATE WHEN: WEARING SEXY CLOTHES"
##
## $format.stata
## [1] "%8.0g"
##
## $class
## [1] "haven_labelled" "vctrs_vctr"     "double"
##
## $labels
## Not mentioned     Mentioned
##             0             1
```

We can see here that the value 1 corresponds to cases where this circumstance was mentioned. Let's see how many people considered this a valid circumstance to have sex without consent.

```
table(eb85_3$qb10_1)
```

```
##
##     0     1
## 24787  3031
```

Fortunately, only a minority of respondents.

Apart from thinking about the variables we will use to measure our outcome of interest for the coursework assignment you will need to select some variables

that you think may be associated with this outcome. In the essay you will need to select a wider set. Here we will only do a few. Again, this is something that needs to be informed by the literature (what variables does the literature considers important) and your own interest.

For the sake of illustration, let's say we are going to look at gender, political background of the respondent, country of origin, age, occupation of the respondents, and type of community they live in. Most of these are demographic variables (not always the more fun or theoretically interesting), but that's all we have in this eurobarometer and so they will have to do.

The same way you try to identify the names of the variables for your outcome variable, you would need to do this for the variables you use to "explain" your outcome. Once you have done your variable selection you can subset your data to only include these.

```
df <- select(eb85_3, qb10_1, qb10_2, qb10_3, qb10_4,
             qb10_5, qb10_6, qb10_7, qb10_8, qb10_9,
             qb10_10, qb10_11, qb10_12, d10, d11,
             isocntry, d1, d25, d15a, uniqid)
```

Ta-da! We now have a new object called *df* with only the variables we will be working with. In this format, it is easier to visualise the data. Notice we have also added a variable called `uniqid`. With many datasets like this you will have a unique id value that allows you to identify individuals. This id may be handy later on, so we will preserve it in our object.

If you `View` this object *df* you will notice that the selected variables appear in the order you selected them. If you wanted a different arrangement, for example you may have preferred to have *uniqid* as your first column, you could have modified the code like so:

```
df <- select(eb85_3, uniqid, qb10_1, qb10_2, qb10_3, qb10_4,
             qb10_5, qb10_6, qb10_7, qb10_8, qb10_9,
             qb10_10, qb10_11, qb10_12, d10, d11,
             isocntry, d1, d25, d15a)
```

Or if you just want to reorder one or few columns (for example you want to also move d1, d25 and d15 a to the front), you could use `everything()` afterwards to save some typing, as below:

```
df <- select(df, uniqid, d1, d25, d15a, everything())
```

If you want to add a lot of columns, it can save you some typing to have a good look at your data and see whether you can't get to your selection by using chunks. Since *qb10_1* and the others are one after the other in the dataset we can use the `start.col:end.col` syntax like below:

```
df <- select(eb85_3, uniqid, qb10_1:qb10_12, d10, d11,
             isocntry, d1, d25, d15a)
```

If you have a lot of columns with a similar structure you can use partial matching by adding `starts_with()`, `ends_with()` or `contains()` in your select statement. So, for example, an alternative to the syntax above we could use the following:

```
df <- select(eb85_3, uniqid, starts_with("qb10"), d10, d11,
             isocntry, d1, d25, d15a)
```

Notice how the text we pass as an argument goes between double quotes.

An alternative is to deselect columns by adding a minus sign in front of the column name. You can also deselect chunks of columns. **Don't execute the code below for this practical**, but if you wanted, for example to get rid of *uniqid* you could do the following:

```
df <- select(df, -uniqid)
```

Yes, a lot of these tips are about saving you some typing. Being lazy (productive, efficient) is fine. You can find more tips like this and useful ideas if you want to rename your columns or variables in this tutorial.

4.5 Creating summated scales

Now comes the next part. What are we going to do with these variables? How are we going to use them? Here you need to do some thinking using your common sense and also considering how other researchers may have used this question about attitudes to sexual violence. There's always many possibilities.

We could, for example, consider that we are going to split the sample in two: those that consider any of these circumstances valid and those that didn't. We would then end up with a binary indicator that we could use as our outcome variable in our analysis.

The thing is that doing that implies loosing information. We may think that someone that consider many circumstances as valid is not the same than the person that only considers one as valid. Yet, creating a global binary indicator would treat these two individuals in the same way.

Another alternative could be to see how many of these circumstances are considered valid excuses for each individual and to produce a sum then for every respondent. Since there are 9 "excuses" we could have a sum from 0 to 9.

This is a very rough **summated scale**. You can read more about the proper development of summated scales here.

Let's do this. We are going to create a new variable that add up the responses to *qb10_1* all the way to *qb10_9*. For this we use the `mutate` function from the `dplyr` package.

```
df <- mutate(df,
       at_sexviol = qb10_1 + qb10_2 + qb10_3 + qb10_4 +
         qb10_5 + qb10_6 + qb10_7 + qb10_8 + qb10_9)
table(df$at_sexviol)
```

```
##
##     0     1     2     3     4     5     6     7     8     9
## 19681  2529  2117  1643   841   416   255   155    57   124
```

We have a skewed distribution. Most people in the survey consider that none of the identified circumstances are valid excuses for having sexual intercourse without consent. On the other hand, only a minority of individuals (124) consider that all of these 9 circumstances are valid excuses for having sex without consent. A high score in this count variable is telling you the participant is more likely to accept a number of circumstances in which sexual intercourse without consent is acceptable. You may read more about count data of this kind here.

Hold on for a second, though. There is a variable `qb10_10` that specifies people that answered "none of these" circumstances. In theory the number of people with a "1" in that variable (that is, selected this item) should equal 19681 (the number of zeros in our new variable). Let's check this out:

```
table(df$qb10_10)
```

```
##
##     0     1
##  9400 18418
```

Oops! Something doesn't add up! Only 18418 people said that none of these circumstances were valid. So why when we add all the other items we end up with 19681 rather than 18418? Notice that there is also a qb10_11 and a qb10_12. These two items identify the people that refused to answer this question and those which did not know how to answer it.

```
table(df$qb10_11)
```

```
##
##     0     1
## 27563   255
```

```
table(df$qb10_12)
```

```
##
##     0     1
## 26810  1008
```

There are 255 people that refused to answer and 1008 that did not know how to answer. If you add 1008, 255, and 18418 you get 19681. So our new variable is actually computing as zeroes people that did not know how to answer this question or refused to answer it. We do not want that. We do not know what these people think, so it would be wrong to assume that they consider that none of these circumstances are valid excuses for sexual intercourse without consent.

There are many ways to deal with this. He could simply filter out cases where we have values of 1 in these two variables (since we don't know their answers we could as well get rid of them). But We could also recode the variable to define this values as what they are NA (missing data, cases for which we have no valid information).

```
df$at_sexviol[df$qb10_11 == 1 | df$qb10_12 == 1] <- NA
table(df$at_sexviol)
```

```
##
##     0     1     2     3     4     5     6     7     8     9
## 18418  2529  2117  1643   841   416   255   155    57   124
```

Pay attention to the code above. When we want to recode based in a condition we use something like that. What we are saying with that code is that we are going to assign as "`NA`" (missing data) the cases for which the condition between the square brackets are met. Those conditions are as defined, when the participants answered don't know *or* (the logical operator for "or" is "|") refused to answer the question (that is when *qb10_11* or *qb10_12* equals 1). You can see other scenarios for recoding using this kind of syntax in the resources we link below. You can see other logical operators used in R here.

Notice that once we do the recoding and rerun the frequency distribution you will see we achieved what we wanted. Now all those missing cases are not counted as zero.

Summated scales like the one we created here are quick and not the most kosher way of combining several questions into a single measure. In research methods you probably learnt that we often work with theoretical constructs that we do not observe directly (e.g., self control, anomie, collective efficacy, etc.). We often call these **latent variables**. In order to study these constructs we create measures or scales that are based in **observed variables** (those we actually

include and ask in the survey). This is a famous measure of *depression* often used in research. As you can see we have different questions (our oberved variables) that we think are related to our latent or not observed (directly) variable (*depression*). You can also see there are several items or questions all of which we researchers think are linked to depression in this scale. If you provide positive answers to many of these questions we may think that you have this unobserved thing we call *depression*.

When measuring latent variables it is a good idea to have multiple items that all capture aspects of the unobserved variable we are really interested in measuring. There is a whole field of statistics that focuses in how to analise if your observed variables are good indicators of your unobserved variable (**psychometry** is how we call this field in psychology) and also that focuses on how to best combine the answers to our observed variables in a single score (**latent variable modelling**). Some of the scores in the Crime Survey for England and Wales that you may use for your essay have been created with this more advanced methods (some of the measures on confidence in the police, for example). Summated scales are not really the best way to do this. But these are more advanced topics that are covered in upper level undergraduate or postgraduate courses. So for now, we will use summated scales as a convenient if imperfect way of aggregating observed variables.

4.6 Collapsing categories in character variables

One of the variables we selected is the country in which the participant lives. Let's have a look at this variable.

```
table(df$isocntry)
```

```
##
##     AT     BE     BG     CY     CZ   DE-E   DE-W     DK     EE     ES     FI
##   1016   1029   1001    501   1060    533   1052   1010   1001   1008   1042
##     FR GB-GBN GB-NIR     GR     HR     HU     IE     IT     LT     LU     LV
##   1009   1006    300   1000   1026   1046   1002   1013   1004    508   1010
##     MT     NL     PL     PT     RO     SE     SI     SK
##    500   1003   1002   1000   1007   1109   1012   1008
```

There are 30 countries in the sample. You may consider that for the purposes of your analysis maybe that is too much. For the sake of this tutorial, let's say that maybe you are not really interested in national differences but in regional differences across different parts of Europe. Say you may want to explore whether these attitudes are different across Western/Central Europe, Scandinavian countries, Mediterranean countries, and Eastern Europe. You do not have

a variable with these categories but since you have a variable that gives you the nations you could create such a variable. How would you do that?

First, you need to consider what the new categories are going to be and how you are going to distribute the countries in your sample across those categories. You may do that in a piece of paper. Then you would need to write the code to have a new variable with this information.

We are going to group the countries in 4 regions: Western (AT, BE, CZ, DE-E, DE-W, FR, GB-GBN, GB-NIR, IE, LU, NL), Eastern (BG, EE, HU, LT, LV, PL, RO, SK), Southern (CY, ES, GR, HR, IT, MT, PT, SI), and Northern Europe (DK, FI, SE).

Then you need to figure out what kind of variable we are dealing with here:

```
class(df$isocntry)
```

```
## [1] "character"
```

Ok, this is a categorical unordered variable, we know that. But this kind of variables could be encoded into R as either *character* vectors, *factor* variables, or as we have seen as well as *haven_labelled*. How you recode a variable is contingent in how it is encoded. Here we are going to show you how you would do the recoding with a *character* variable such as *isocntry* into another character variable we will call *region*. We will see later examples for how to recode *factors*.

We will have a variable with four new categories (Western, Southern, Eastern, and Northern) whenever the right conditions are met. See the code below:

```
df$region[df$isocntry == "AT" | df$isocntry == "BE" |
            df$isocntry == "CZ" | df$isocntry == "DE-E" |
            df$isocntry == "DE-W" | df$isocntry == "FR" |
            df$isocntry == "GB-GBN" | df$isocntry == "GB-NIR" |
            df$isocntry == "IE" | df$isocntry == "LU" |
            df$isocntry == "NL"] <- "Western"
df$region[df$isocntry == "BG" | df$isocntry == "EE" |
            df$isocntry == "HU" | df$isocntry == "LT" |
            df$isocntry == "LV" | df$isocntry == "PL" |
            df$isocntry == "RO" |
            df$isocntry == "SK"] <- "Eastern"
df$region[df$isocntry == "DK" | df$isocntry =="FI" |
            df$isocntry == "SE"] <- "Northern"
df$region[df$isocntry == "CY" | df$isocntry == "ES" |
            df$isocntry == "GR" | df$isocntry =="HR" |
            df$isocntry == "IT" | df$isocntry =="MT" |
            df$isocntry == "PT" |
            df$isocntry =="SI"] <- "Southern"
```

```
table(df$region)
```

```
##
##  Eastern Northern Southern  Western
##     8079     3161     7060     9518
```

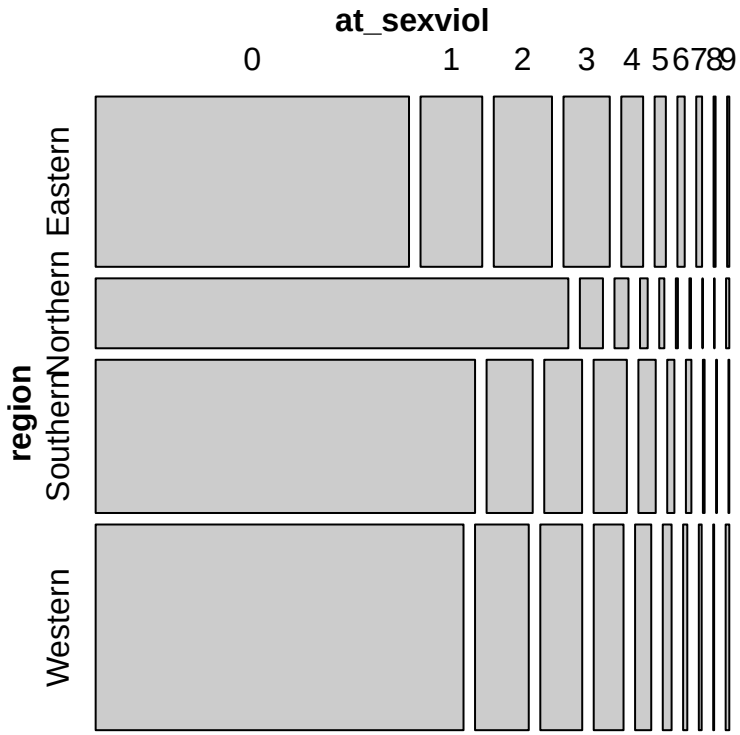
What we are doing above is initialising a new variable in the *df* object that we are calling *region*. We are then assigning to each of the four categories in this character vector those participants in the survey that have the corresponding values in the *isocntry* variable as defined for each category within the square brackets. So for example if Austria is the value in *isocntry* we are telling R we want this person to be assigned the value of "Western" in the *region* variable. And so on. You can see the list of ISO country codes here.

Once you have created the variable you could start exploring if there is any association with our outcome variable. For example, using **mosaic plots** from the `vcd` package (if you don't have it installed the code won't run).

```
library(vcd)
```

```
## Loading required package: grid
```

```
mosaic(~region + at_sexviol, data = df)
```



In a mosaic plot like this the height of the region levels indicate how big that group is. You can see there are many more observations in our sample that come from Western countries than from Northern countries. Here what we are interested is the length. We see that Northern countries have proportionally more people in the zero category than any other group. On the other hand, Eastern countries have the fewer zeros (so looking as if attitudes more permissive towards sexual violence are more common there, even if still a minority). We will come back to this kind of plots later on this semester.

4.7 Working with apparently cryptic variable names and levels

Let's look at the variable *d10* in our dataset:

```
table(df$d10)
```

```
##
##     1     2
## 12230 15588
```

What is this? Unclear, isn't it? If you look at the questionnaire you will see that this variable measures gender, that values of 1 correspond to men and that values of 2 corresponds to woman. But this is far from straightforward just by looking a the printed table or the name of the variable.

To start with we may want to change the name of the variable. One way to do this is the following:

```
colnames(data)[colnames(data)=="old_name"] <- "new_name"
```

Of course, we need to change the names for valid ones in our case. So adapting that code we would write as follows:

```
colnames(df)[colnames(df)=="d10"] <- "gender"
```

If you prefer the *tydiverse* dialect (that aims to save you typing among other things), then you would use the `rename` function as below (beware if you already renamed *d10* using `colnames` there will be no longer a *d10* variable and therefore R will return an error).

```
df <- rename(df, gender=d10)
```

If you now check the names of the variables (with the `names` function) in `df` you will see what has happened:

```
names(df)
```

```
##  [1] "uniqid"     "qb10_1"     "qb10_2"     "qb10_3"     "qb10_4"
##  [6] "qb10_5"     "qb10_6"     "qb10_7"     "qb10_8"     "qb10_9"
## [11] "qb10_10"    "qb10_11"    "qb10_12"    "gender"     "d11"
## [16] "isocntry"   "d1"         "d25"        "d15a"       "at_sexviol"
## [21] "region"
```

If you want to change many variable names it may be more efficient doing it all at once. First we are going to select fewer variables and retain only the ones will continue using:

```
df <- select(df, uniqid, at_sexviol, gender, d11, d1, d25, d15a, region)
```

Now we can change a bunch of names all at once with the following code:

```
names(df)[4:7] <- c("age", "politics", "urban", "occupation")
names(df)
```

```
## [1] "uniqid"     "at_sexviol" "gender"     "age"        "politics"
## [6] "urban"      "occupation" "region"
```

You may have seen we wrote `[4:7]` above. This square bracket notation is identifying the columns within that particular object. It is indicating to R that we only wanted to change the name of the variables defining column 4 to 7 in the dataframe, those corresponded to d11, d1, d25, and d15a, which respectively (we can see in the questionnaire) correspond to age, politics, whether the respondent live in a urban setting, and occupation. We assign to those columns the names we specify (in the appropriate order) in the list that follows. Now we will be less likely to confuse ourselves, since we have columns with meaningful names (which will appear in any output and plots we produce).

Let's look at occupation:

```
table(df$occupation)
```

```
##
##    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16
## 1588 1864 1845 8916  129   14  384  768  517  878  313 1921 2202  856 2062  271
##   17   18
## 2471  819
```

There are 18 categories here. And it is not clear what they mean.

```
class(df$occupation)
```

```
## [1] "haven_labelled" "vctrs_vctr"     "double"
```

Remember that this is `haven_labelled`. If we look at the attributes we can see the correspondence between the numbers above and the labels.

```
attributes(df$occupation)
```

```
## $label
## [1] "OCCUPATION OF RESPONDENT"
##
## $format.stata
## [1] "%8.0g"
##
```

```
## $class
## [1] "haven_labelled" "vctrs_vctr"     "double"
##
## $labels
##       Responsible for ordinary shopping, etc.
##                                             1
##                                       Student
##                                             2
##           Unemployed, temporarily not working
##                                             3
##                         Retired, unable to work
##                                             4
##                                        Farmer
##                                             5
##                                    Fisherman
##                                             6
##                   Professional (lawyer, etc.)
##                                             7
##              Owner of a shop, craftsmen, etc.
##                                             8
##                     Business proprietors, etc.
##                                             9
## Employed professional (employed doctor, etc.)
##                                            10
##                       General management, etc.
##                                            11
##                        Middle management, etc.
##                                            12
##                     Employed position, at desk
##                                            13
##                  Employed position, travelling
##                                            14
##                Employed position, service job
##                                            15
##                                    Supervisor
##                                            16
##                           Skilled manual worker
##                                            17
##                 Unskilled manual worker, etc.
##                                            18
```

Having to look at this every time is not very convenient. You may prefer to simply use the labels directly. For this we can use the `to_factor` function from the `labelled` package.

```
library(labelled)
df$f_occup <- to_factor(df$occupation)
class(df$f_occup)
```

```
## [1] "factor"
```

```
table(df$f_occup)
```

```
##
##       Responsible for ordinary shopping, etc.
##                                          1588
##                                       Student
##                                          1864
##           Unemployed, temporarily not working
##                                          1845
##                         Retired, unable to work
##                                          8916
##                                        Farmer
##                                           129
##                                     Fisherman
##                                            14
##                   Professional (lawyer, etc.)
##                                           384
##              Owner of a shop, craftsmen, etc.
##                                           768
##                     Business proprietors, etc.
##                                           517
## Employed professional (employed doctor, etc.)
##                                           878
##                       General management, etc.
##                                           313
##                        Middle management, etc.
##                                          1921
##                     Employed position, at desk
##                                          2202
##                  Employed position, travelling
##                                           856
##                Employed position, service job
##                                          2062
##                                     Supervisor
##                                           271
##                          Skilled manual worker
##                                          2471
##                 Unskilled manual worker, etc.
##                                           819
```

Now you have easier to interpret output.

4.8 Recoding factors

As we said there are many different types of objects in R and depending on their nature the recoding procedures may vary. You may remember that many times categorical variables will be encoded as factors. Let's go back our newly created *f_occup*. We have 18 categories here. These are possibly too many. Some of them have to few cases, like *fisherman*. Although this is a fairly large dataset we only have 14 fisherman. It would be a bit brave to guess what fishermen across Europe think about sexual violence based in a sample of just 14 of them.

This is a very common scenario when you analyse data. In these cases is helpful to think of ways of adding the groups with small counts (like fishermen in this case) to a broader but still meaningful category. People often refer to this as *collapsing categories*. You also have to think theoretically, do you have good enough reasons to think that there's ought to be meaningful differences in attitudes to sexual violence among these 18 groups? If you don't you may want to collapse into fewer categories that may make more sense.

As when we created the *region* variable the first thing to do is to come out with a new coding scheme for this variable. We are going to use the following. I'm not making this scheme up, I am just using the scheme that the Eubarometer uses to simplify these categories.

```
1   Self-employed (5 to 9 in d15a)
2   Managers (10 to 12 in d15a)
3   Other white collars (13 or 14 in d15a)
4   Manual workers (15 to 18 in d15a)
5   House persons (1 in d15a)
6   Unemployed (3 in d15a)
7   Retired (4 in d15a)
8   Students (2 in d15a)
```

It would be quicker to recode from *d15a* into a new variable (there would be less typing when dealing with numbers rather than labels). But here we are going to use this example to show you how to recode from a factor variable, so instead we will recode form *f_occup*.

As often in R, there are multiple ways of doing this. We could use `dplyr::recode`. But here we will show you how to do this using the `recode` function from the `car` library.

```
df$occup2 <- df$f_occup
levels(df$occup2) <- list("Self-employed" =
```

```
                          c("Farmer" , "Fisherman" ,
                            "Professional (lawyer, etc.)" ,
                            "Owner of a shop, craftsmen, etc.",
                            "Business proprietors, etc."),
                        "Managers" =
                          c("Employed professional (employed doctor, etc.)",
                            "General management, etc.",
                            "Middle management, etc."),
                        "Other white collar" =
                          c("Employed position, at desk",
                            "Employed position, travelling"),
                        "Manual workers" =
                          c("Employed position, service job",
                            "Supervisor",
                            "Skilled manual worker",
                            "Unskilled manual worker, etc."),
                        "House persons" = "Responsible for ordinary shopping, etc.",
                        "Unemployed" = "Unemployed, temporarily not working",
                        "Retired" =  "Retired, unable to work",
                        "Student" = "Student")
```

One of the things you need to be very careful with when recoding factors or character variables is that you need to input the chunk of texts in the existing variables exactly as they appear in the variable. Otherwise you will get into trouble. So, for example, if in the code above you wrote `fishermen` instead of `Fisherman` you would have 14 less cases in the `Self-Employed` level than you should have. When doing this kind of operations is always convenient to check that the categories add up correctly.

For more details in how to recode factors and other potential scenarios you may want to read this paper by Amelia McNamara and Nicholas Horton.

4.9 Understanding missing data

As we have already seen, you will have participants that do not provide you with valid answers for the questions in the survey. In general in any kind of data set you work with, whether it comes from surveys or not, you will have cases for which you won't have valid information for particular variables in your dataframe. Missing data is common.

Let's look at the *politics* variable.

```
class(df$politics)
```

```
## [1] "haven_labelled" "vctrs_vctr"     "double"
```

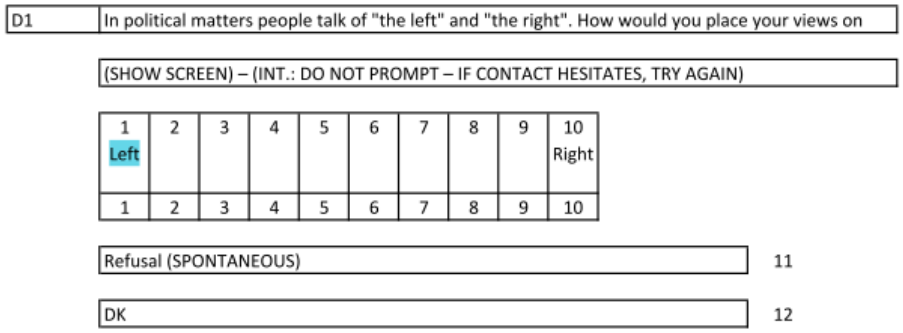
```
table(df$politics)
```

```
##
##    1    2    3    4    5    6    7    8    9   10   97   98
## 1433  830 2161 2217 6690 2238 2056 1658  474 1126 2766 4169
```

```
sum(is.na(df$politics))
```

```
## [1] 0
```

It looks as if we have no missing data. Right? Well, appearances can be deceiving sometimes.



This is the question as it was asked from the survey respondents. Notice the difference in the response options and the categories in *politics*. We know that those that see themselves further to the left will have answer 1 and those that see themselves further to the right would have answer 10. What does 97 and 98 then refers to? If you look at the questionnaire you will see that those that refuse or don't know *in the questionnaire* are coded as 11 and 12.

Let's look closer at the `attributes`:

```
attributes(df$politics)
```

```
## $label
## [1] "LEFT-RIGHT PLACEMENT"
##
## $format.stata
## [1] "%8.0g"
##
## $class
## [1] "haven_labelled" "vctrs_vctr"     "double"
```

```
##
## $labels
##   Box 1 - left          Box 2          Box 3          Box 4          Box 5
##              1              2              3              4              5
##          Box 6          Box 7          Box 8          Box 9 Box 10 - right
##              6              7              8              9             10
##        Refusal             DK
##             97             98
```

A tip if you don't want to see as much output. If you want to access directly only some of the attributes you can call them directly modifying the code as below:

```
attributes(df$politics)$labels
```

```
##   Box 1 - left          Box 2          Box 3          Box 4          Box 5
##              1              2              3              4              5
##          Box 6          Box 7          Box 8          Box 9 Box 10 - right
##              6              7              8              9             10
##        Refusal             DK
##             97             98
```

Or we could use the `val_labels` function from the `labelled` package for the same result:

```
val_labels(df$politics)
```

```
##   Box 1 - left          Box 2          Box 3          Box 4          Box 5
##              1              2              3              4              5
##          Box 6          Box 7          Box 8          Box 9 Box 10 - right
##              6              7              8              9             10
##        Refusal             DK
##             97             98
```

Notice that though the questionnaire assigned a value of 11 to refusal and 12 to "don't know" answers, in the dataset those values are instead 97 and 98. You will see this often - 97, 98, 99, and similar codes are often used to denote missing data.

You may have good theoretical reasons to preserve this in your analysis. Perhaps you think that people that did not answer this question have particular reasons not to do so and those reasons may be associated with their attitudes to violence. In that case you may want to somehow preserve them in your analysis. Absent that rationale you may just want to treat them as they are: missing data. You just have no way of knowing if these people are more or less lefty or conservative. So let's declare them as such for sake of explaining how you would do that:

```
df$politics[df$politics>=97] <- NA
table(df$politics)
```

```
##
##    1    2    3    4    5    6    7    8    9   10
## 1433  830 2161 2217 6690 2238 2056 1658  474 1126
```

The equal or greater operator in R is written as `>=`. What we are saying in the first statement to R is that whenever *politics* has a value equal or greater than 97 we are going to recode those cases as `NA`, as missing data. Since this is a `haven_labelled` vector we can use numbers here rather than characters or factor levels as part of the recoding. That makes things a bit easier in that it saves typing. Let's look a the results:

```
summary(df$politics)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   1.000   4.000   5.000   5.196   7.000  10.000    6935
```

Now you see you have 6935 cases with missing data. As always it is good to check the original variable. We had 2766 and 4169 between refusals and "don't know" respondents. These add up to 6935.

Let's check something else:

```
library(skimr)
skim(df$politics)
```

Mmmmm. The `summary` function worked, but other functions do not know to treat the values in the `haven_labelled` vector as a numeric. So, to avoid problems we may want to define as such (if we truly believe this is a quantitative rather than a categorical variable or at the very least are willing to treat it as quantitative).

```
df$politics_n <-as.numeric(df$politics)
```

If you try this now, you will see it works:

```
skim(df$politics_n)
```

If we want to see what percentage of cases is NA across our dataset we could use `colMeans` combined with `is.na`. This will compute the mean (the proportion) of cases that are NA in each of the columns of the dataframe:

```
colMeans(is.na(df))
```

```
##     uniqid at_sexviol     gender        age   politics      urban occupation
## 0.00000000 0.04540226 0.00000000 0.00000000 0.24929902 0.00000000 0.00000000
##     region    f_occup     occup2 politics_n
## 0.00000000 0.00000000 0.00000000 0.24929902
```

This is suspicious. Only the variables we have created and already sorted seem to have NA. This may be a function of the `haven_labelled` vectors behaving like with the original *politics* variable. Let's explore it. We can use the `val_labels` function from the `labelled` package to extract labels from the whole dataframe like this:

```
val_labels(df)
```

```
## $uniqid
## NULL
##
## $at_sexviol
## NULL
##
## $gender
##   Man Woman
##     1     2
##
## $age
##                       15 years                       96 years
##                             15                             96
## 99 and older [not documented]
##                             99
##
## $politics
##   Box 1 - left          Box 2          Box 3          Box 4          Box 5
##              1              2              3              4              5
##          Box 6          Box 7          Box 8          Box 9 Box 10 - right
##              6              7              8              9             10
##        Refusal             DK
##             97             98
##
## $urban
## Rural area or village     Small/middle town            Large town
##                     1                     2                     3
##                    DK
##                     8
```

```
##
## $occupation
##       Responsible for ordinary shopping, etc.
##                                             1
##                                       Student
##                                             2
##           Unemployed, temporarily not working
##                                             3
##                         Retired, unable to work
##                                             4
##                                        Farmer
##                                             5
##                                    Fisherman
##                                             6
##                   Professional (lawyer, etc.)
##                                             7
##              Owner of a shop, craftsmen, etc.
##                                             8
##                     Business proprietors, etc.
##                                             9
## Employed professional (employed doctor, etc.)
##                                            10
##                       General management, etc.
##                                            11
##                        Middle management, etc.
##                                            12
##                      Employed position, at desk
##                                            13
##                  Employed position, travelling
##                                            14
##                Employed position, service job
##                                            15
##                                    Supervisor
##                                            16
##                           Skilled manual worker
##                                            17
##                 Unskilled manual worker, etc.
##                                            18
##
## $region
## NULL
##
## $f_occup
## NULL
##
## $occup2
```

```
## NULL
##
## $politics_n
## NULL
```

Notice that in *urban* there are explicit codes for missing data but that these values won't be treated as such, at least we do something. Let's see how many cases we have in this scenario:

```
table(df$urban)
```

```
##
##     1     2     3     8
##  8563 11881  7356    18
```

Not too bad. Let's sort this variable:

```
df$urban[df$urban>=8] <- NA
df$f_urban <- to_factor(df$urban)
table(df$f_urban)
```

```
##
## Rural area or village     Small/middle town            Large town
##                  8563                 11881                  7356
##                    DK
##                     0
```

```
mean(is.na(df$f_urban))
```

```
## [1] 0.0006470631
```

You can see that even though have now no cases with a value of 8 or DK, the label appears printed in the output. This is still a valid level:

```
levels(df$f_urban)
```

```
## [1] "Rural area or village" "Small/middle town"     "Large town"
## [4] "DK"
```

So, we may want to remove it explicitly if we don't want output reminding us what we already know, that "there are not" DK cases (since now we call them and treat them as NA). For removing unused levels in a factor we use the `droplevels` function,

```
df$f_urban <- droplevels(df$f_urban)
table(df$f_urban)
```

```
##
## Rural area or village     Small/middle town            Large town
##                  8563                 11881                  7356
```

Above also saw the codes for *gender* let's use `to_factor` again for this variable:

```
df$f_gender <- to_factor(df$gender)
```

Let's just keep the variables we will retain for our final analysis:

```
df_f <- select(df, uniqid, at_sexviol, f_gender, age, politics_n,
               f_urban, f_occup, region)
```

We are going to do some further exploration of NA. Let's create a new variable that identifies the cases that have missing data in at least one of the columns in the data with the final variables for our analysis. For this we can use the `complete.cases` function that creates a logical vector indicating whether any of the cases has missing values in at least one column:

```
df_f$complete <- complete.cases((df_f))
table(df_f$complete)
```

```
##
## FALSE  TRUE
##  7634 20184
```

```
mean(df_f$complete)
```

```
## [1] 0.7255734
```

So, shocking as this may sound you only have full data in 72% of the participants. Notice how the percentage of missing cases in the variables range:

```
colMeans(is.na(df_f))
```

```
##       uniqid   at_sexviol     f_gender          age   politics_n      f_urban
## 0.0000000000 0.0454022575 0.0000000000 0.0000000000 0.2492990150 0.0006470631
##      f_occup       region     complete
## 0.0000000000 0.0000000000 0.0000000000
```

The function `complete.cases` is returning what cases have missing data **in any of** the variables not in a singular one. It is not unusual for this percentage to be high. You may end up with a massive loss of cases even though the individual variables themselves do not look as bad as the end scenario.

Later this term we will cover forms of analysis (e.g., multiple regression) in which you have to work with multiple variables at the same time. This form of analysis require you to have information for all the cases and all the variables. Every case for which there is not information in one of the variables in your analysis is automatically excluded from such analysis. So the actual sample size you use when using these techniques is the sample size for which you have full information. In our example your sample size (when you do multiple regression analysis or any multivariate analysis) will be 20184 rather than 27818.

This being the case it makes sense to exclude from your data all cases that have some missing information in the variables you will be using in your analisis. For this you can use the `na.omit` function. We create a new object (I name it "full_df", you can call it whatever) and put inside it the outcome of running `na.omit` in our original sample ("df_f").

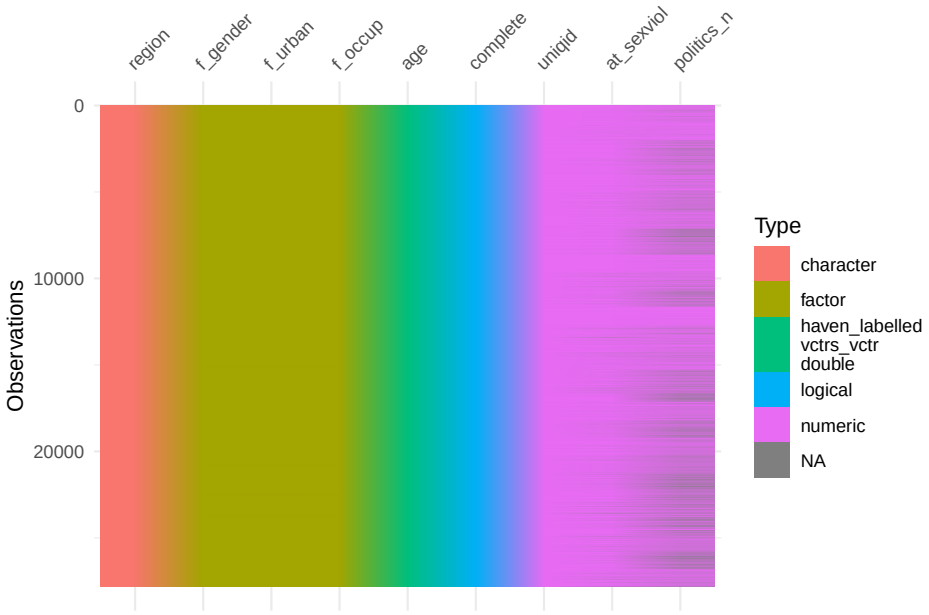
```
full_df <- na.omit(df_f)
```

Now we have a dataset ready for starting our analysis. This is the point, when you are doing your assignment, that you can start running frequency distributions that you will report, and all kinds of statistical tests that we cover in the weeks to come.

4.10 Exploring dataframes visually

We have covered now a number of functions you can use to explore your data, such as `skimr::skim()`, `str()`, `summary()`, `table()`, or `dplyr::glimpse()`. But sometimes is useful to get a more panoramic way. For this we can use the `visdat` package for visualising whole dataframes amd its main function `vis_dat()`.

```
library(visdat)
vis_dat(df_f)
```



Nice one! You get a visual representations of how your variables are encoded in this dataframe. You have several categorical variables such as region, f_gender, f_urban, and f_occup. We see that region is encoded as a `character` vector, whereas the others are `factors`. For the purposes of this course, it is generally better to have your categorical variables encoded as factors. So one of the next steps in our data prep may be to recode region as a factor.

```
df_f$f_region <- as.factor(df_f$region)
```

We can also see we have age, a quantitative variable, *age*, encoded as `haven_labelled`. We could as well encoded as `numeric`.

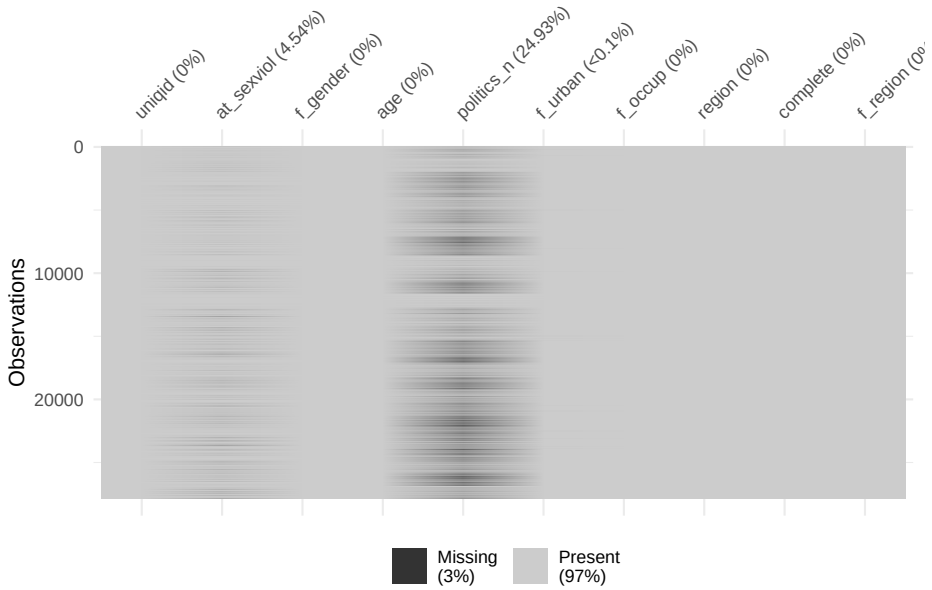
```
df_f$age <- as.numeric(df_f$age)
```

What we do with *at_sexviol* depends on how we decide to treat it. But for arguments sake let's say we are going to treat it as numerical.

```
df_f$at_sexviol <- as.numeric(df_f$at_sexviol)
```

The othe piece of info you get with `vis_dat` is the prevalence of missing data (NA) for each variable (the dark horizontal cases represent a missing case in the variable). We can summarise missing data visually more clearly with `vis_miss`.

```
vis_miss(df_f)
```



You can find more details about how to explore missing data in the vignette of the `naniar` package here.

4.11 A quick recap on descriptive statistics

4.12 Further resources

There are many other ways to recode variables and create new variables based in existing ones. Here we only provided some examples. We cannot exhaust all possibilities in this tutorial. For the purposes of the assignment you may have specific ideas of how you may want to combine variables or recode existing ones. You can find additional examples and code for how to do the recoding of variables in the following links. Please make sure that you spend some time looking at these additional resources. They are not optional.

Wrangling categorical data in R

http://rprogramming.net/recode-data-in-r/

http://www.cookbook-r.com/Manipulating_data/Recoding_data/

https://mgimond.github.io/ES218/Week03a.html

https://www.r-bloggers.com/from-continuous-to-categorical/

You should also look for the `dplyr` documentation for the functions mutate() and recode().

If you would like to combine several variables into one for your analysis in more complex ways but do not know how to do that, please do get in touch. Also do not hesitate to approach us if you have any other specific queries.

Chapter 5

Foundations of statistical inference: confidence intervals

5.1 Introduction

Up to now we have introduced a series of concepts and tools that are helpful to describe sample data. But in data analysis we often do not observe full populations. We often only have sample data.

Think of the following two problems:

You want to know the extent of intimate partner violence in a given country. You could look at police data. But not every instance of intimate partner violence gets reported to, or recorded by, the police. We know there is a large proportion of those instances that are not reflected in police statistics. You could do a survey. But it would not be practical to ask everybody in the country about this. So you select a sample and try to develop an estimate of what the extent of this problem is in the population based on what you observe in the sample. But, how can you be sure your sample guess, your estimate, is any good? Would you get a different estimate if you select a different sample?

You conduct a study to evaluate the impact of a particular crime prevention program. You select a a number of areas as part of the study. Half of it you randomly allocate to your intervention and the other half to your control or comparison group. Imagine that you observe these areas after the intervention is implemented and you notice there is a difference. There is less crime in your intervention areas. How can you reach conclusions about the effectiveness of the intervention based on observations of differences on crime in these areas?

What would have happened if your randomisation would have split your sample in different ways, would you still be able to observe an effect?

For this and similar problems we need to apply statistical inference: a set of tools that allows us to draw inferences from sample data. In this session we will cover a set of important concepts that constitute the basis for statistical inference. In particular, we will approach this topic from the **frequentist** tradition.

It is important you understand this is not the only way of doing data analysis. There is an alternative approach, **bayesian statistics**, which is very important and increasingly popular. Unfortunately, we do not have the time this semester to also cover Bayesian statistics. Typically, you would learn about this approach in more advanced courses.

Unlike in previous and future sessions, the focus today will be less applied and a bit more theoretical. However, it is important you pay attention since understanding the foundations of statistical inference is essential for a proper understanding of everything else we will discuss in this course. The code we cover *in the first few sections* this week is much trickier but won't be instrumental for your assignment, so don't worry too much if you don't fully understand it.

5.2 Generating random data

For the purpose of today's session we are going to generate some fictitious data. We use real data in all other sessions but it is convenient for this session to have some randomly generated fake data (actually technically speaking pseudo-random data)[1].

So that all of us gets the same results (otherwise there would be random differences!), we need to use the `set.seed()` function. Basically your numbers are pseudo random because they're calculated by a number generating algorithm, and setting the *seed* gives it a number to "grow"" these pseudo random numbers out of. If you start with the same seed, you get the same set of random numbers.

So to guarantee that all of us get the same randomly generated numbers, set your seed to 100:

```
set.seed(100)
```

We are going to generate a large object (100,000 cases) with skewed data. We often work with severely skewed data in criminology. For generating this type

[1] Although we would like to think of our samples as random, it is in fact very difficult to generate random numbers in a computer. Most of the time someone is telling you they are using random numbers they are most likely using pseudo-random numbers. If this is the kind of thing that gets you excited you may want to read the wiki entry. If you want to know how R generates these numbers you should ask for the help pages for the Random.Seed function.

of data I am going to use the `rnbinom()` function for something called negative binomial distributions, which is a discrete probability distribution often use as a model for counts.

```
skewed <- rnbinom(100000, mu = 1, size = 0.3)
```

The code above creates data that follow a negative binomial distribution, essentially a highly skewed distribution. Don't worry too much about the other parameters we are using as arguments at this stage, but if curious look at `?rnbinom`.

We can also get the mean and standard deviation for this object using `mean` and `sd` respectively:

```
mean(skewed)
```

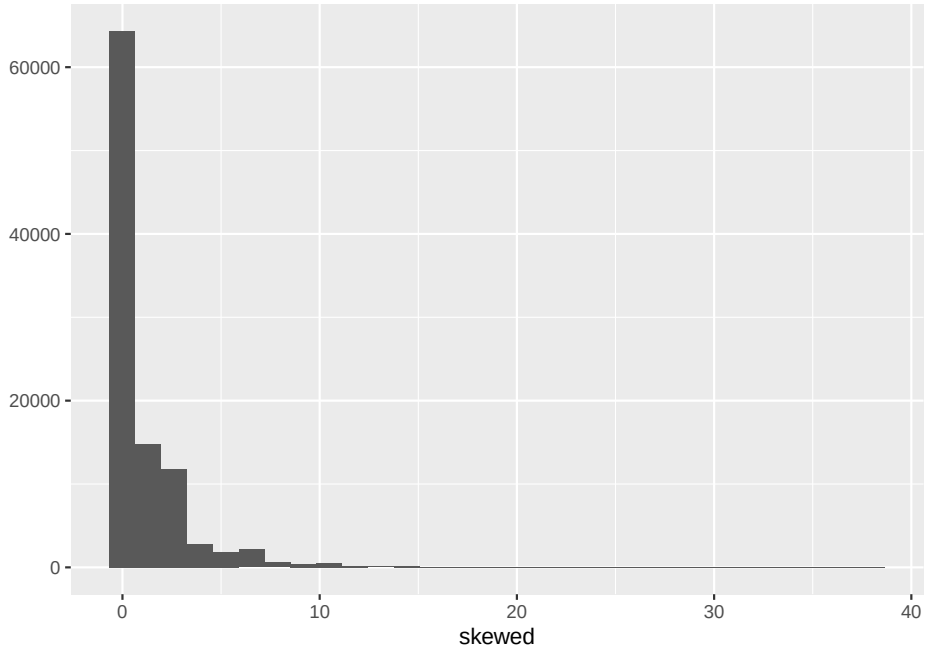
```
## [1] 1.00143
```

```
sd(skewed)
```

```
## [1] 2.083404
```

And we can also see what it looks like:

```
library(ggplot2)
qplot(skewed)
```



We are going to pretend this variable measures numbers of crime perpetrated by an individual in the previous year. Let's see how many offenders we have in this fake population.

```
sum(skewed > 0)
```

```
## [1] 35623
```

We are now going to put this variable in a dataframe and we are also going to create a new categorical variable -identifying whether someone offended over the past year (e.g., anybody with a count of crime higher than 0). Let's start by creating a new dataframe (*"fakepopulation"*) with the skewed variable we created rebaptised as *crimecount*.

```
fake_population <- data.frame(crimecount = skewed)
```

Then let's define all values above 0 as "Yes" in a variable identifying offenders and everybody else as "No". We use the `ifelse()` function for this.

```
fake_population$offender <- ifelse(fake_population$crimecount > 0, c("Yes"), c("No"))
#Let's check the results
table(fake_population$offender)
```

```
##
##    No   Yes
## 64377 35623
```

We are now going to generate a normally distributed variable (watch this short video if you are unclear what a normal distribution is).

We are going to pretend that this variable measures IQ. We are going to assume that this variable has a mean of 100 in the non-criminal population (pretending there is such a thing) with a standard deviation of 15 and a mean of 92 with a standard deviation of 20 in the criminal population. I am pretty much making up these figures.

The first expression is asking R to generate random values from a normal distribution with mean 100 and standard deviation for every of the 64394 "non-offenders" in our fake population data frame. WARNING: If you run the `table()` function and the number of non-offenders you get is different, you will need to amend the code below accordingly (this will happen if you did not use the seed or generated the original skewed variable more than once).If you get the following message it means you are using the wrong number of people in each category: *number of items to replace is not a multiple of replacement lengthnumber of items to replace is not a multiple of replacement length.*

```
fake_population$IQ[fake_population$offender == "No"] <- rnorm(64377, mean = 100, sd = 15)
```

And now we are going to artificially create somehow dumber offenders.

```
fake_population$IQ[fake_population$offender == "Yes"] <- rnorm(35623, mean = 92, sd = 20)
```

We can now have a look at the data. Let's plot the density of IQ for each of the two groups and have a look at the summary statistics.

```
##This will give us the mean IQ for the whole population
mean(fake_population$IQ)
```

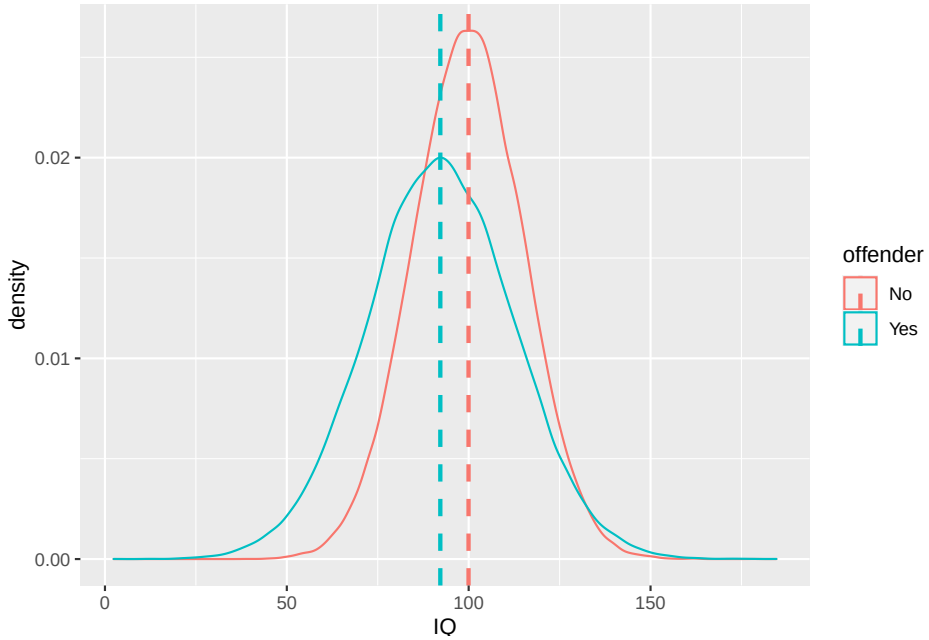
```
## [1] 97.19921
```

```
#We will load the plyr package to get the means for IQ for each of the two offending groups
library(plyr)
#We will store this mean in a data frame (IQ_means) after getting them with the ddply function
IQ_means <- ddply(fake_population, "offender", summarise, IQ = mean(IQ))
#You can see the mean value of IQ for each of the two groups, unsurprisingly they are as we defi
IQ_means
```

```
##   offender       IQ
## 1       No 99.96347
## 2      Yes 92.20370
```

We are going to create a plot with the density estimation for each of the plots (first two lines of code) and then I will add a vertical line at the point of the means (that we saved) for each of the groups

```
ggplot(fake_population, aes(x = IQ, colour = offender)) +
  geom_density() +
  geom_vline(aes(xintercept = IQ, colour = offender), data = IQ_means,
             linetype = "dashed", size = 1)
```



So, now we have our fake population data. In this case, because we generated the data ourselves, we know what the "population"" data looks like and we know what the summary statistics for the various attributes (IQ, crime) of the population are. But in real life we don't normally have access to full population data. It is not practical or economic. It is for this reason we rely on samples.

5.3 Sampling data and sampling variability

It is fairly straightforward to sample data with R. The following code shows you how to obtain a random sample of size 10 from our population data above:

```
#We will use the sample function within the mosaic package.
library(mosaic)
sample(fake_population$IQ, 10)
```

```
##  [1] 123.02991 108.10296  94.90335 101.76552 102.96829 100.88919 105.88429
##  [8] 107.06557  90.07606 108.01750
```

First of all notice that `mosaic` masks quite a few functions from various packages. If you want to use them remember to use the `package_I_need::for_function_I_want` formula we covered in previous sessions.

You may be getting sample elements that are different from mine, depending on the seed you used and how many times before you tried to obtain a *random* sample. You can compute the mean for a sample generated this way:

```
mean(sample(fake_population$IQ, 10))
```

```
## [1] 102.9824
```

And every time you do this, you will be getting a slightly different mean. Try to rerun the code several times. This is one of the problems with sample data. Not two samples are going to be exactly the same and as a result, every time you compute the mean you will be getting a slightly different value. Run the function three or four times and notice the different means you get as the elements that make up your sample vary.

We can also use code to automatise the process. The following code will ask R to draw 15 samples of size 10 and obtain the means for each of them.

```
do(15) * with(sample(fake_population, 10), mean(IQ))
```

```
##         with
## 1   91.83841
## 2   92.54385
## 3   99.13335
## 4  102.72856
## 5  102.78062
## 6   94.71675
## 7   83.25669
## 8   95.69518
## 9   97.58754
## 10 103.38229
## 11  93.91211
## 12 105.36669
## 13  93.90070
## 14  95.85066
## 15  96.46357
```

So here we have the means that we obtain from 15 different samples from this population. Notice how they vary. We can store the results from an exercise such as this as a variable and plot it:

```
#The following code will create a dataframe with the results
samp_IQ <- do(15) * with(sample(fake_population, 10), mean(IQ))
#You can see the name of the variable designating the means in the create data frame
names(samp_IQ)
```

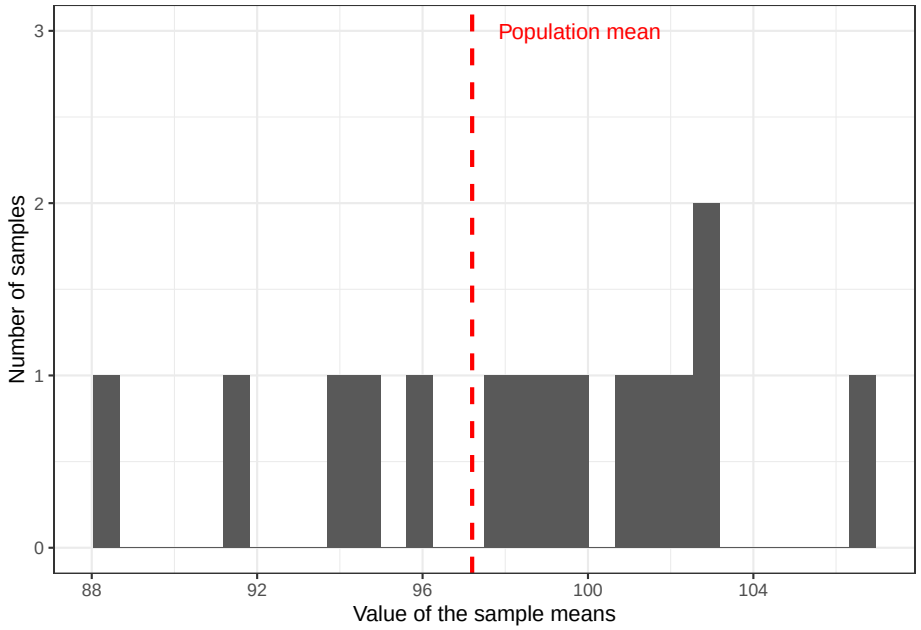
```
## [1] "with"
```

```
#We are going to create a data frame with the population mean
IQ_mean <- data.frame(mean(fake_population$IQ))
#Have a look inside the object to see what the variable containing the mean is called
names(IQ_mean)
```

```
## [1] "mean.fake_population.IQ."
```

```
#And we can plot it then
ggplot(samp_IQ, aes(x = with)) +
  geom_histogram() +
  geom_vline(aes(xintercept = mean.fake_population.IQ.), data = IQ_mean,
             linetype = "dashed", size = 1, colour = "red") + #This code will add a re
  labs(x = "Value of the sample means", y = "Number of samples") +
  annotate("text", x = 99.8, y = 3, label = "Population mean", colour = "red") + #Anno
  theme_bw()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Your exact results may differ from those shown here, but you can surely see the point. We have a problem with using sample means as a guess for population means. Your guesses will vary. How much of a problem is this? This excellent piece and demonstration by New York Times reporters illustrate the problem well. We are going to learn that something called the **central limit theorem** is of great assistance here.

5.4 Sampling distributions and sampling experiments

We are going to introduce an important new concept here: **sampling distribution**. A sampling distribution is the probability distribution of a given statistic based on a random sample. It may be considered as *the distribution of the statistic for all possible samples from the same population of a given size.*

We can have a sense for what the sampling distribution of the means of IQ for samples of size 10 in our example by taking a large number of samples from the population. This is called a **sampling experiment**. Let's do that. We will take 50000 samples (rather than 15 as before) of size 10 (that is, with ten elements each) and compute the means. This may take a bit. Wait until you see the object appear in your environment.

```
sampd_IQ_10 <- do(50000) * with(sample(fake_population, 10), mean(IQ))
```

So now we have 50000 sample means from samples of size 10 taken from our fake population. We are now going to take the mean of this 50000 sample means and then we are going to compare it to the population mean.

```
mean(sampd_IQ_10$with) #mean of the sample means
```

```
## [1] 97.17187
```

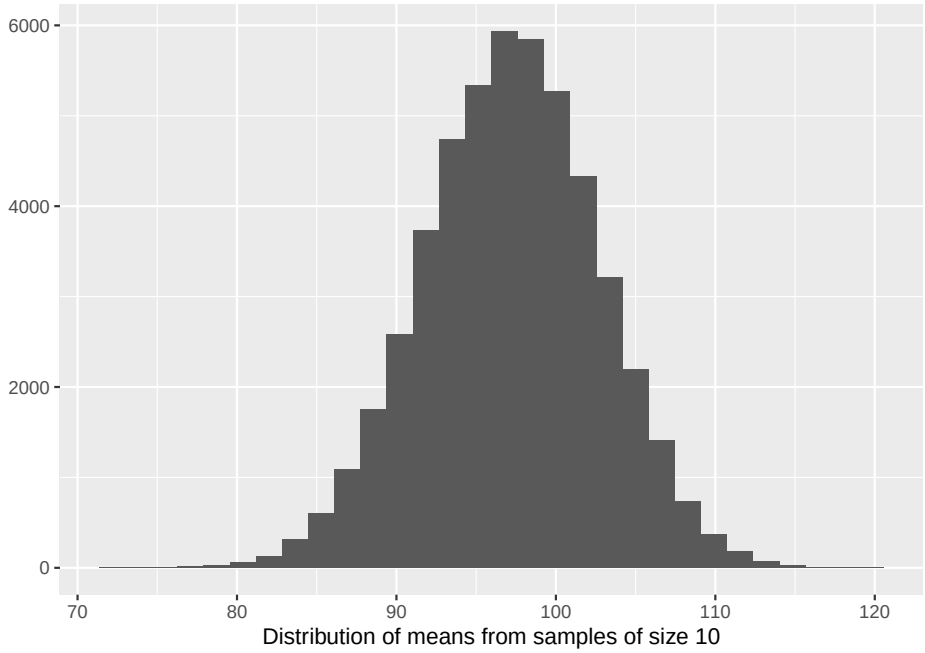
```
mean(fake_population$IQ) #mean of the population
```

```
## [1] 97.19921
```

Wow! They are pretty much the same. What we have observed is part of something called the **central limit theorem**, a concept from probability theory. One of the first things that the central limit theorem tells us is that **the mean of the sampling distribution of the means (also called the expected value) should equal the mean of the population**. It won't be quite the same in this case (to all the decimals) because we only took 50000 samples, but in the very long run (if you take many more samples) they would be the same.

Let's now visually explore the distribution of the sample means.

```
#Now we plot the means
qplot(sampd_IQ_10$with, xlab = "Distribution of means from samples of size 10")
```



Amazing too, isn't it? When you (1) take many random samples from a normally distributed variable; (2) compute the means for each of these samples; and (3) plot the means of each of these samples, you end up with something that is also normally distributed. **The sampling distribution of the means of normally distributed variables in the population is normally distributed**. I want you to think for a few seconds as to what this means and then keep reading.

Did you think about it? What this type of distribution for the sample means is telling us is that most of the samples will give us guesses that are clustered around their own mean, as long as the variable is normally distributed in the population (which is something, however, that we may not know). Most of the sample means will cluster around the value of 97.11 (in the long run), which is the population mean in this case. There will be some samples that will give us much larger and much smaller means (look at the right and left tail of the distribution), but most of the samples won't gives us such extreme values.

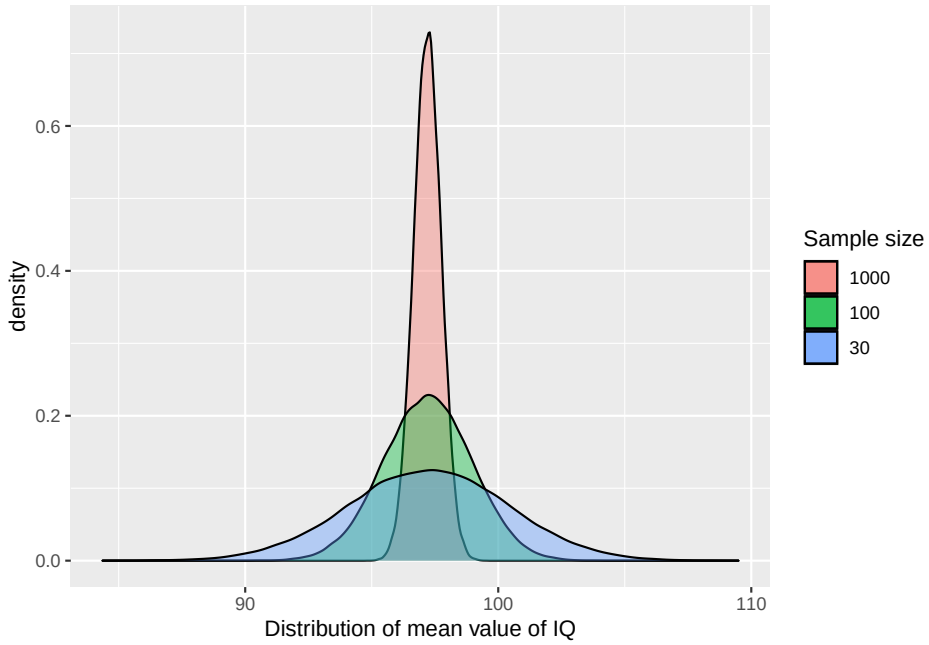
So although every sample will give us different values to the mean, in the long run, they will tend to be similar to the mean of the population. If you were to take repeated samples from the same population more often than not, the sample mean will be closer rather than far from the population mean. When you take a sample you have no way of knowing if your sample is one of those that got close to the population mean or far from it. But it is somehow reassuring to know the procedure in the long run tends to get it right more often than not.

Another way of saying this is that the means obtained from random samples

behave in a predictable way. When we take just one sample and compute the mean we won't we able to tell whether the mean for that sample is close to the centre of its sampling distribution (and thus to the population mean). But we will know the probability of getting an extreme value for the mean is lower than the probability of getting a value closer to the mean. That is, if we can assume that the variable in question is normally distributed in the population.

But it gets better. Let's repeat the exercise with a sample size of 30, 100 and 1000. This code will take some time to run. But compare it with how long it would take you to take 10 pieces of paper from a bag with 100000 pieces of paper 50000 times…

```
sampd_IQ_30 <- do(50000) * with(sample(fake_population, 30), mean(IQ))
sampd_IQ_100 <- do(50000) * with(sample(fake_population, 100), mean(IQ))
sampd_IQ_1000 <- do(50000) * with(sample(fake_population, 1000), mean(IQ))
#Plot the results, notice how we have changed the aesthetics. We are definig them within each geo
ggplot() +
    geom_density(data = sampd_IQ_1000, aes(x = with, fill = "1000"), position = "identity", alpha
   geom_density(data = sampd_IQ_100, aes(x = with, fill = "100"), position = "identity", alpha =
   geom_density(data = sampd_IQ_30, aes(x = with, fill = "30"), position = "identity", alpha = 0.
  labs(fill = "Sample size") + #This will change the title of the legend
  scale_fill_discrete(limits = c("1000", "100", "30")) + #This will ensure the order of the items
  xlab("Distribution of mean value of IQ")
```



Pay attention to the aesthetics here. Because essentially we are looking at three different datasets, the variables we plot are identified not in an `aes` statement

within the general `ggplot()` function but rather the `aes` are included and specified within each of the geoms we are plotting.

But back to the substantive point, can you notice the differences between these **sampling distributions**? *As the sample size increases, more and more of the samples tend to cluster closely around the mean of the sampling distribution.* In other words with larger samples the means you get will tend to differ less from the population mean than with smaller samples. You will be more unlikely to get means that are dramatically different from the population mean.

Let's look closer to the summary statistics using `favstats()` from the loaded `mosaic` package:

```
favstats(~with, data = sampd_IQ_30)
```

```
##       min       Q1   median       Q3      max     mean       sd     n missing
##  84.36391 95.10328 97.23989 99.36223 109.4893 97.22025 3.159672 50000       0
```

```
favstats(~with, data = sampd_IQ_100)
```

```
##       min       Q1   median       Q3      max     mean       sd     n missing
##  90.40428 96.03429 97.21813 98.38842 103.9214 97.20767 1.743929 50000       0
```

```
favstats(~with, data = sampd_IQ_1000)
```

```
##       min       Q1   median       Q3     max     mean        sd     n missing
##  94.84451 96.83374 97.20142 97.56903 99.5329 97.19854 0.5469711 50000       0
```

As you can see the mean of the sampling distributions is pretty much the same regardless of sample size, though since we only did 50000 samples there's still some variability. But notice how the **range** (the difference between the smaller and larger value) is much larger when we use smaller samples. When I run this code I get one sample of size 30 with a sample mean as low as 83 and another as high as 110. But when I use a sample size of a 1000 the smallest sample mean I get is 95 and the largest sample size I get is 99. *When the sample size is smaller the range of possible means is wider and you are more likely to get sample means that are wide off from the expected value.*

This variability is also captured by the standard deviation of the sampling distributions, which is smaller the larger the sample size is. The standard deviation of a sampling distribution receives a special name you need to remember: the **standard error**. In our example, with samples of size 30 the standard error is 3.17, whereas with samples of size 1000 the standard error is 0.55.

We can see that the precision of our guess or estimate (that is the sample mean we use to infer the population mean) improves as we increase the sample size.

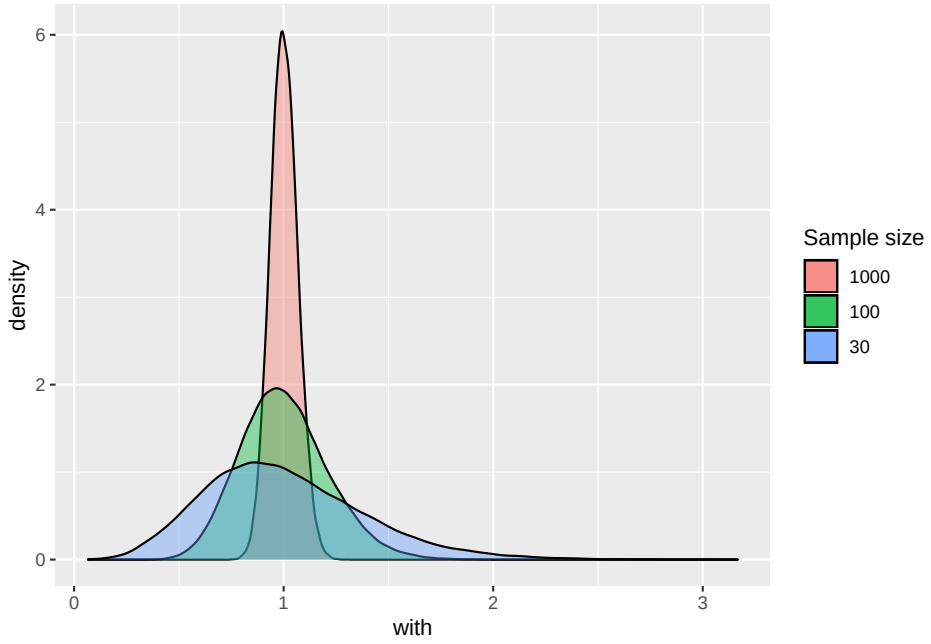
So we can conclude that using the sample mean as a estimate (we call it a **point estimate** because it is a single value, a single guess) of the population mean is not a bad thing to do *if your sample size is large enough and the variable can be assumed to be normally distributed in the population.* As we have illustrated here, more often than not this guess won't be too far off in those circumstances.

But what about variables that are not normally distributed. What about *"crimecount"*? We saw this variable was quite skewed. Let's take numerous samples, compute the mean of *"crimecount"*, and plot its distribution. Let's generate the sampling distributions for sample sizes of 30, 100, and 1000 elements. This will take a minute or so depending in how good your machine is.

```
sampd_CR_30 <- do(50000) * with(sample(fake_population, 30), mean(crimecount))
sampd_CR_100 <- do(50000) * with(sample(fake_population, 100), mean(crimecount))
sampd_CR_1000 <- do(50000) * with(sample(fake_population, 1000), mean(crimecount))
```

And now let's plot the means from these samples (the sampling distributions).

```
ggplot() +
    geom_density(data=sampd_CR_1000, aes(x = with, fill = "1000"), position = "identity", alpha =
   geom_density(data=sampd_CR_100, aes(x = with, fill = "100"), position = "identity", alpha = 0.
   geom_density(data=sampd_CR_30, aes(x = with, fill = "30"), position = "identity", alpha = 0.4)
  labs(fill = "Sample size") + #This will change the title of the legend
  scale_fill_discrete(limits = c("1000", "100", "30")) #This will ensure the order of the items
```



```
favstats(~with, data = sampd_CR_30)
```

```
##         min        Q1    median       Q3      max     mean        sd     n
##  0.06666667 0.7333333 0.9666667 1.233333 3.166667 1.003719 0.3804143 50000
##  missing
##        0
```

```
favstats(~with, data = sampd_CR_100)
```

```
##   min   Q1 median   Q3  max     mean        sd     n missing
##  0.36 0.86   0.99 1.13 2.05 1.003114 0.2082509 50000       0
```

```
favstats(~with, data = sampd_CR_1000)
```

```
##    min    Q1 median    Q3   max     mean         sd     n missing
##  0.751 0.957      1 1.045 1.307 1.001474 0.06548459 50000       0
```

```
mean(fake_population$crimecount)
```

```
## [1] 1.00143
```

You can see something similar happens. Even though *"crimecount"* itself is not normally distributed. The sampling distribution of the means of *"crimecount"* becomes more normally distributed the larger the sample size gets. Although we are not going to repeat the exercise again, the same would happen even for the variable *"offender"*. With a binary categorical variable such as offender (remember it could take two values: yes or no) the "mean" represents the proportion with one of the outcomes. But essentially the same process applies.

What we have seen in this section is an illustration of various amazing facts associated with the central limit theorem. Most sample means are close to the population mean, very few are far away from the population mean, and on average, we get the right answer (i.e., the mean of the sample means is equal to the population mean). This is why statisticians say that the sample mean is an **unbiased** estimate of the population mean.

How is this helpful? Well, it tells us we need large samples if we want to use samples to guess population parameters without being too far off. It also shows that although sampling introduces error (**sampling error**: the difference between the sample mean and the population mean), this error behaves in predictable ways (in most of the samples the error will be small, but it will be larger in some: following a normal distribution). In the next section, we will see how we can use this to produce something called confidence intervals.

If you want to further consolidate some of these concepts you may find these videos on sampling distributions from Khan Academy useful.