Modelling Criminological Data LAWS20452

Juanjo Medina and Reka Solymosi

2019-01-29

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.

#A first lesson about R (Week 1)

##Install R & RStudio

We recommend that you use your own laptops for this course. This way you get used to working in an environment which you will continue to use after this semester. However, our lab sessions will be held in computer clusters in case you do not have access to a laptop (or something goes wrong…).

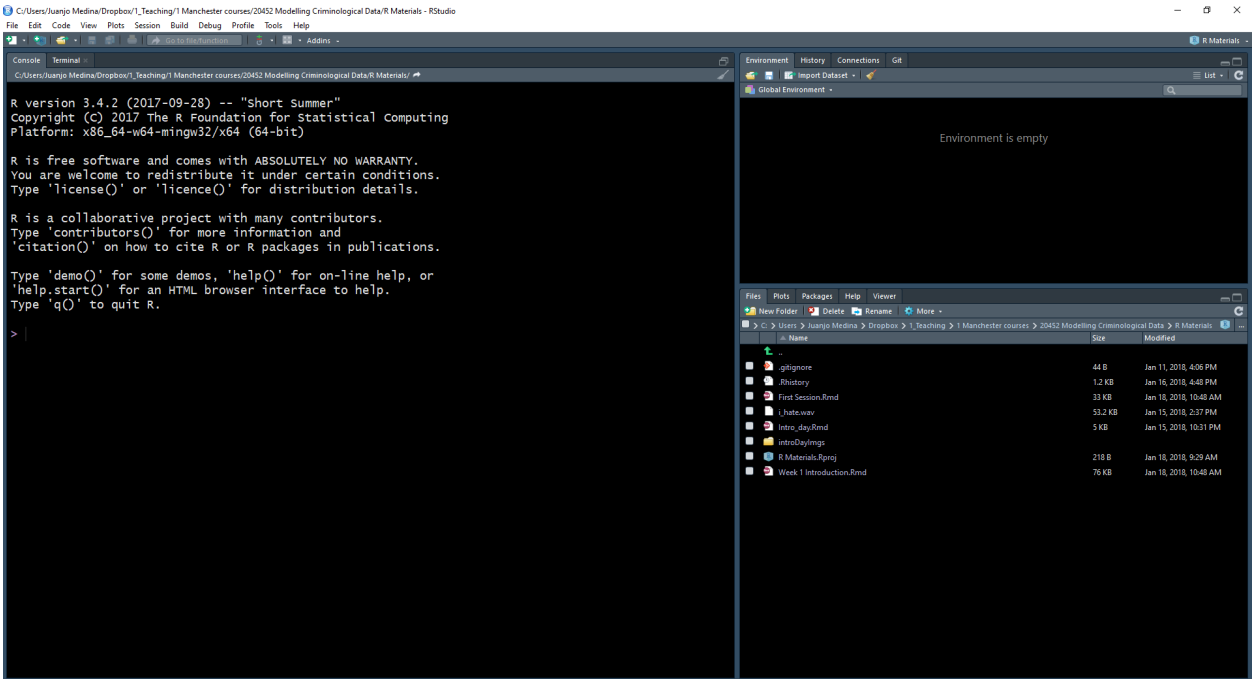
You *don't need to install the software in the computers available in the clusters*, because it is already there. Beware though, the installation may vary a bit across different computer clusters in the University. This, on itself, is another good reason to use your own laptops -for it will provide you with a more stable environment. If you have not already, then please download and install R and R Studio onto your laptops. Otherwise use the cluster machines.

- click here for instructions using Windows or
- here for instructions using a Mac.

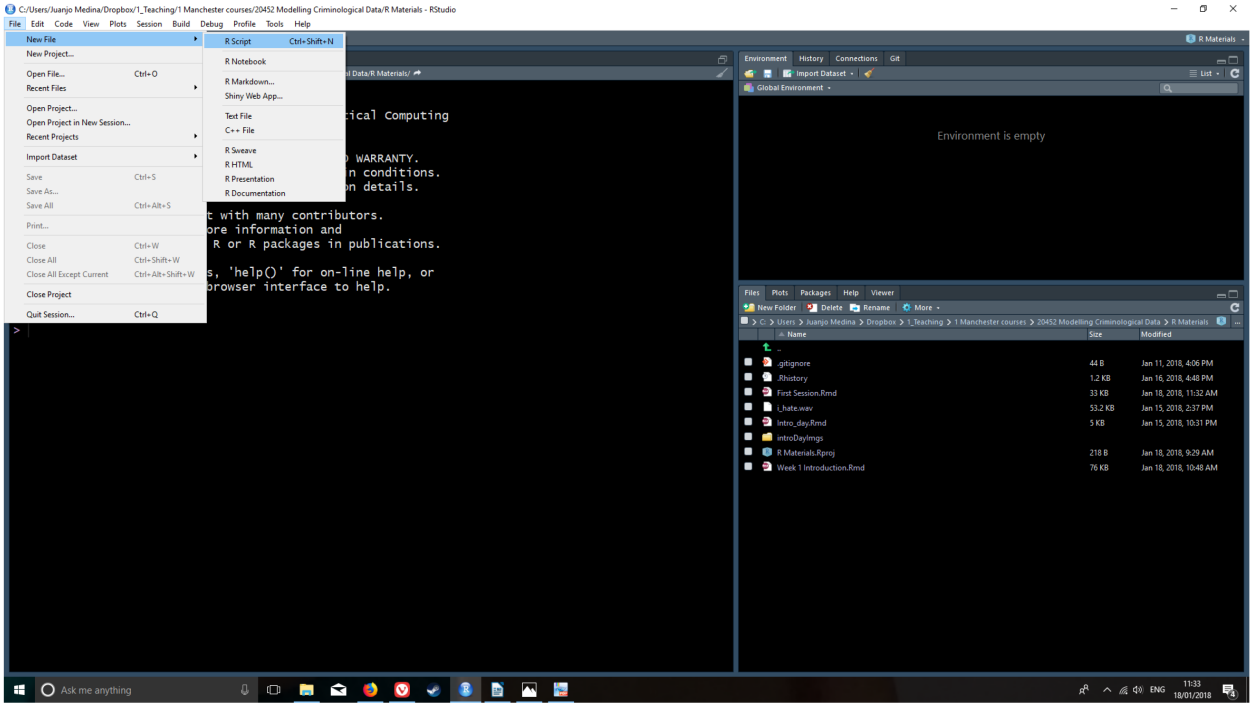
##Open up and explore RStudio

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.

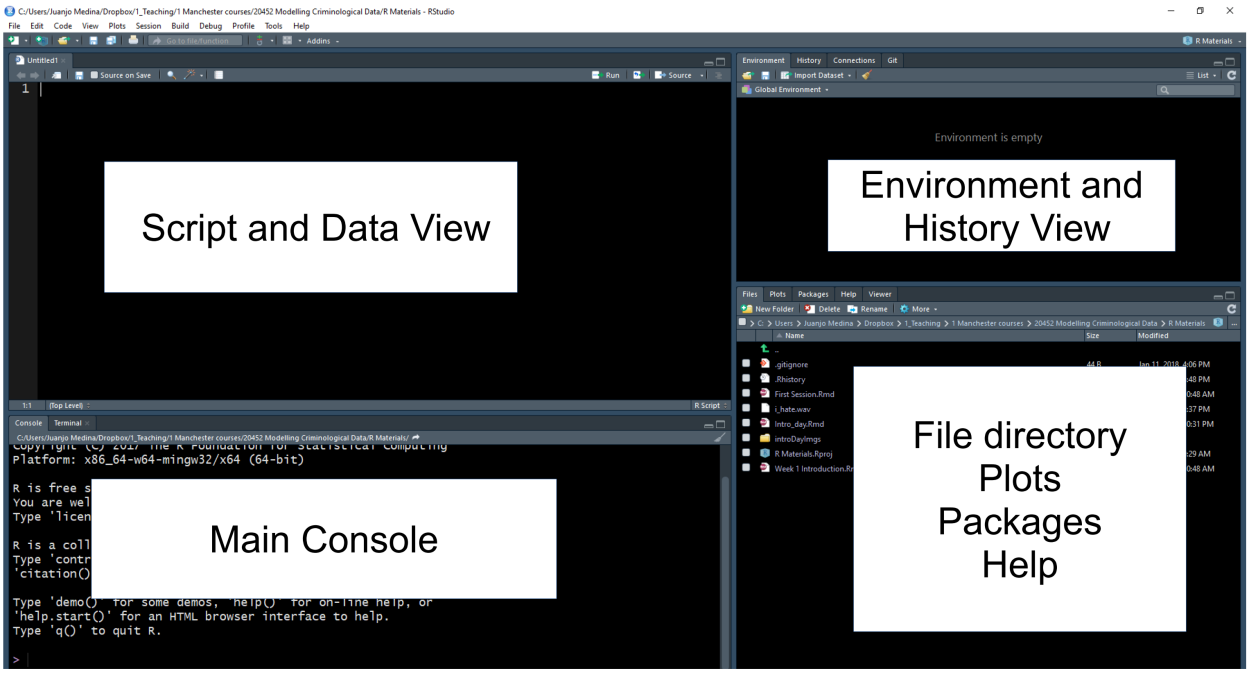
R Studio is what we call an IDE, an **integrated development environment**. It is a fancy way of saying that it is a cool interface designed to write programming code. Every time you open up R Studio you are in fact starting a R session. R Studio automatically runs R in the background. We will be interacting 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 windows. 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. R is constantly being updated.



The view in R Studio is structured so that you have 4 open windows 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?

You can think of R as a Lego monster. You can make the monster scarier and more powerful by adding new bits to it. Packages are those bits. They are modules that expand what R can do. There are thousands of them. Which is pretty cool!!! R can do many more things than Excel. That is down to the fact that researchers all over the world write packages that continuously expand the functionality of R. You can think of a package as another drop down menu that gets added to you menu tab with loads of new options for doing fancy stuff, only they are not really drop down menus. You need to access their added functionality via programming code. So yeah, R is like Excel or SPSS only with over 10,000 "drop down menus." And all for free.

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. A script is nothing but a text file with some code on it. 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 - with code versus with a point and click interface like Excel or SPSS is that you are producing a written record of every step you take in the analysis. First time around it will take you time to write these instructions, it may be slower than pointing and clicking. And unlike with pointing and clicking you need to know the "words" and "grammar" of this language.

Luckily you don't need to memorise or know all these words. 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.

##Customising the RStudio look

R Studio allows you to customise the way it looks. 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.

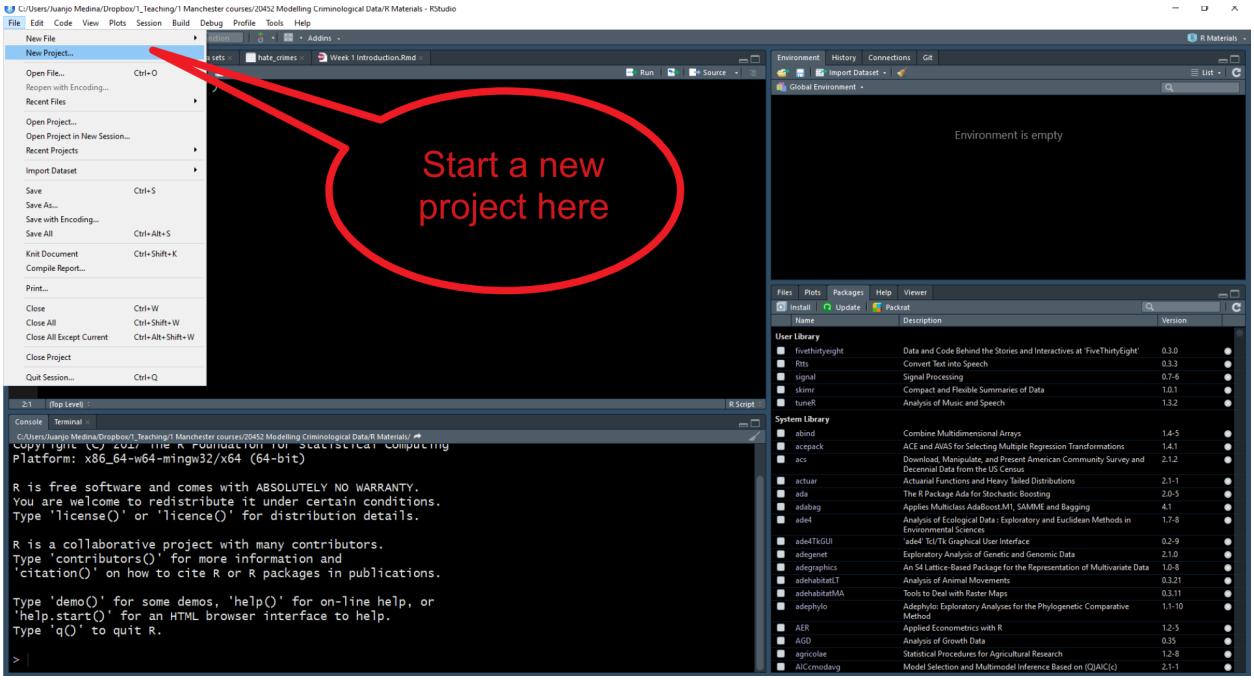
Click in the *Tools* menu and select *Global options*. This will open up a pop up window with various options. Select *Appearance*. In this section you can change the font type and size, but also the kind of theme background that R will use in the various windows. I suffer from poor sight, so I often 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.

##Getting organised: R Projects

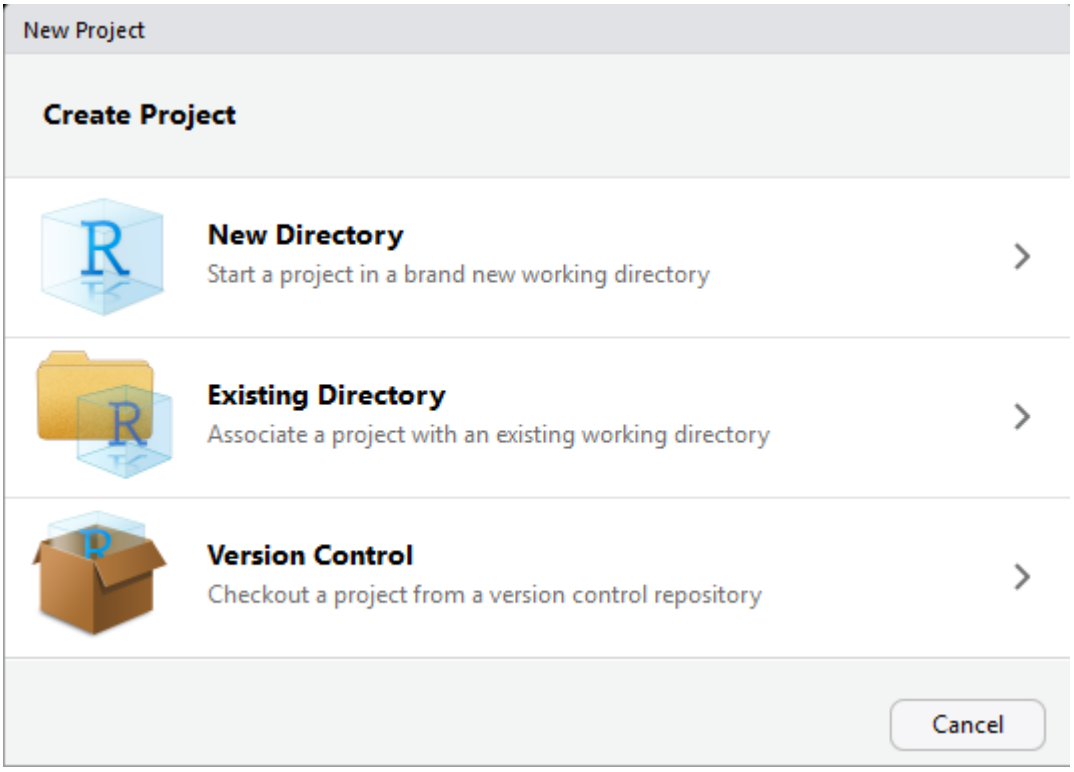
We finished the previous section talking about sharing your analytic code. Let's face it. You would not bring a new partner or somebody that you want to impress to your place before tidying a little bit first, wouldn't you? In the same way, if you know you may have to share your code, if you know you may have guests, you may want to keep your analysis, data, and results tidy. R Studio helps a little bit with that. R Studio helps with this by virtue of something called **R Projects**.

Technically, a R Studio project is just a directory 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).

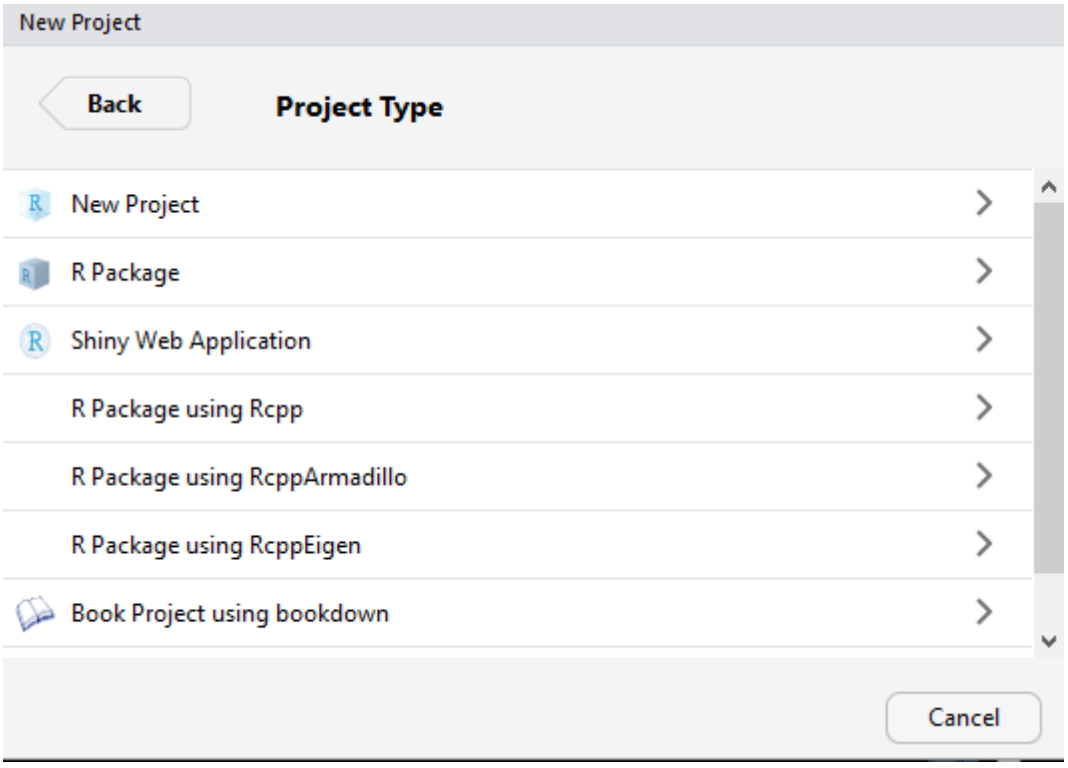
When a project is reopened, R Studio opens every file and data view that was open when the project was closed last time around. Let's learn how to create a project. Go to the *File* drown menu and select *New Project*.



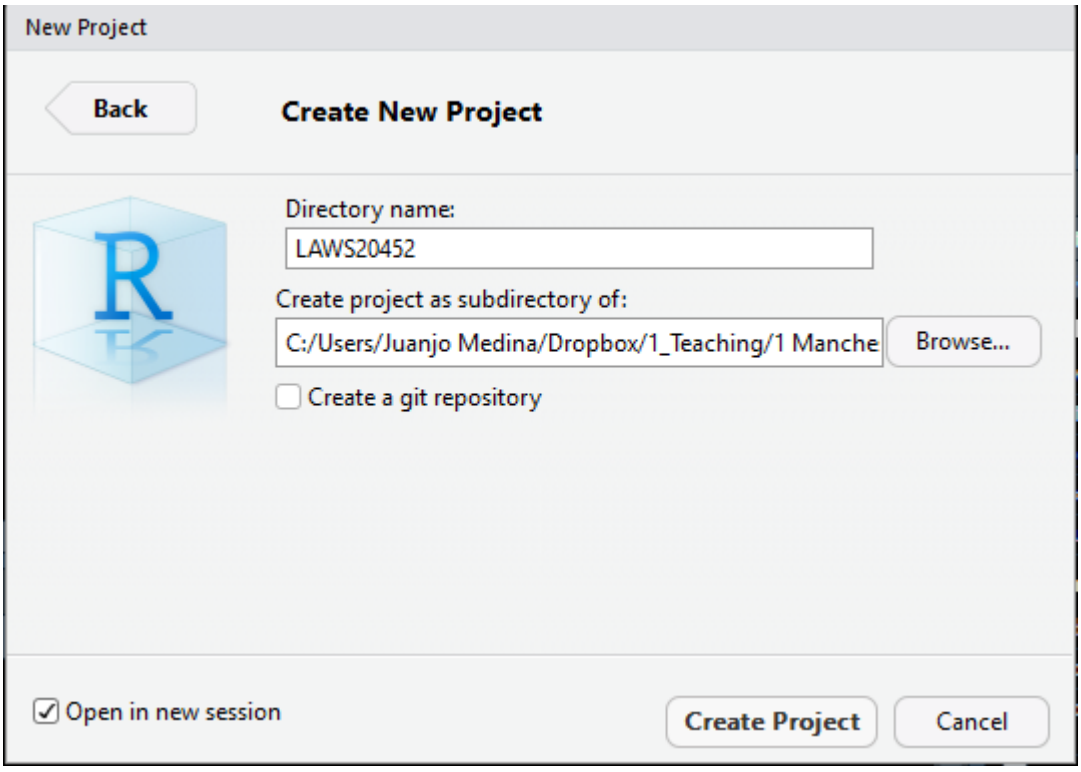
That will open a dialog box where you ask to specify what kind of directory you want to create. Select new working directory in this dialog box.



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 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 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.

If you come to my office, you will see it is a very messy place. But my computer is, in contrast, a very tidy environment. You should aim for your computer workspace to be very organised as well. You can create these subdirectories using Windows Explorer or the Files window in R Studio. Why don't you have a go?

##Talk to your computer

Enough background, let's write some very simple code 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! Though that was a really mean thing to say to your machine!

In these handouts (written in html format) 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. You should start thinking about creating a file with some of the most useful bits of code we cover so that when you do your work you can just cut and paste from this file rather than having to come back to these handouts.

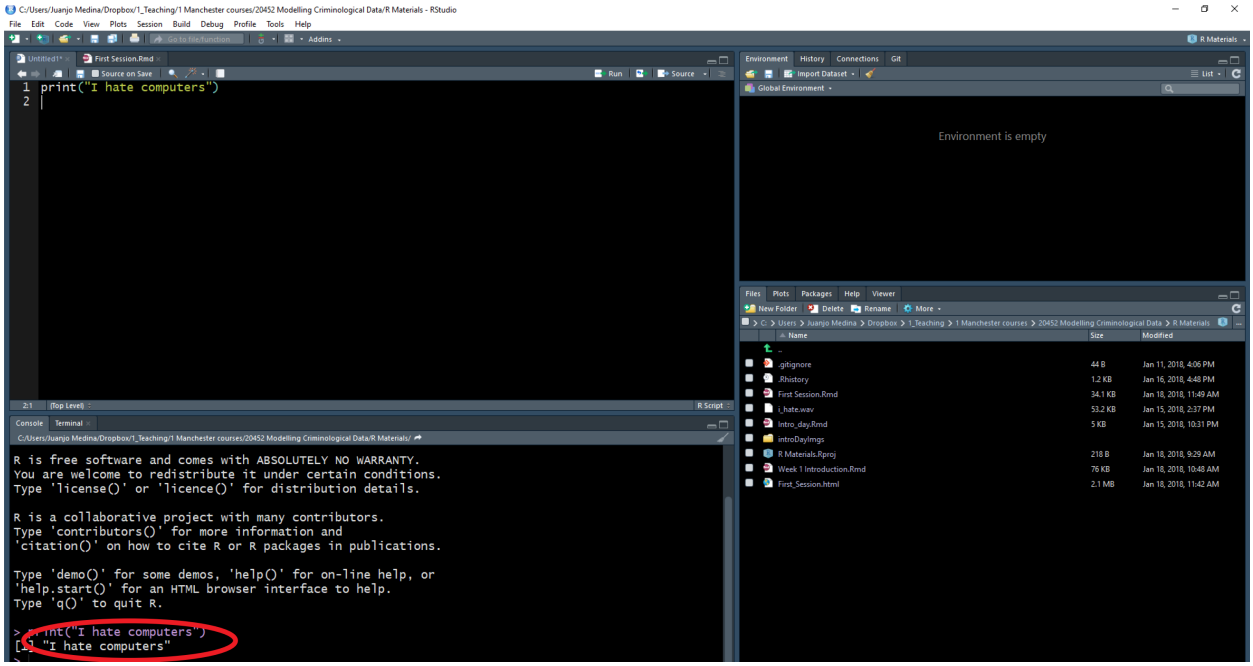
Sometimes in these documents you will see on them the results of running the code, what you see printed in your console or in your plot viewer. The results will appear enclosed in a box as above.

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.



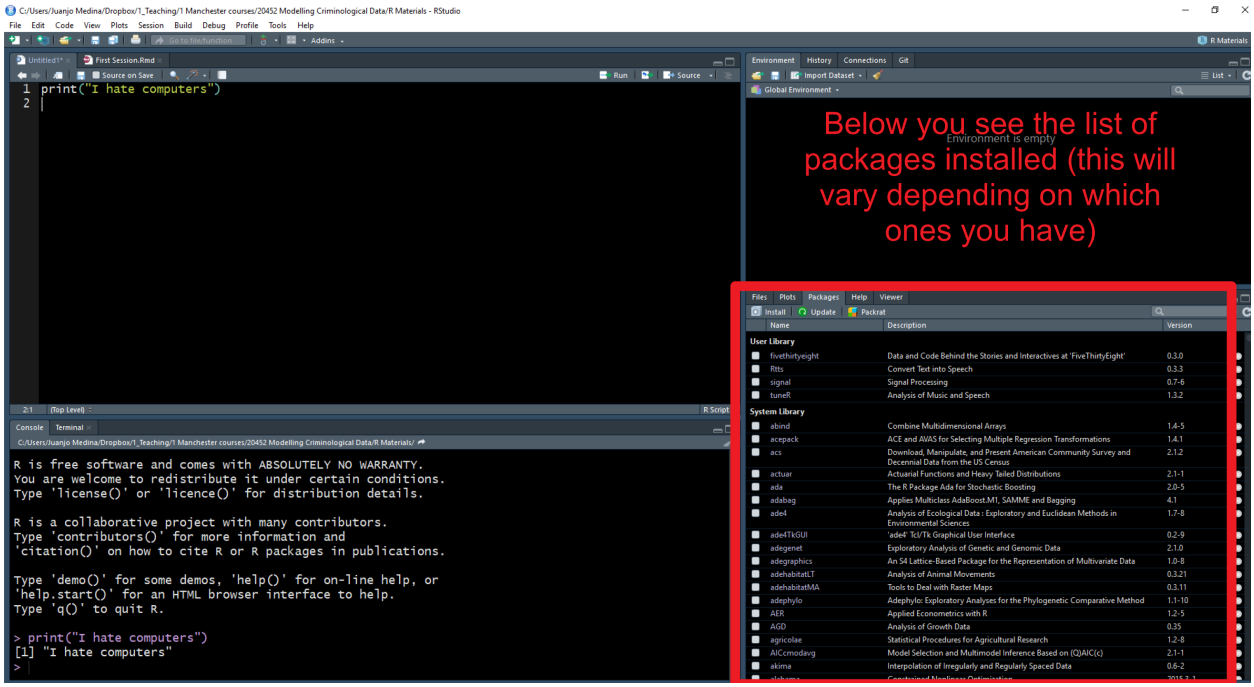
As indicated above, the window in the bottom left corner is the main **console**.You will see that the words "I hate computers" appear printed there. If rather than using R Studio you were working directly from R, that's all you would get: the main console where you can write code interactively (rather 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,

##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 Git Hub (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*. 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 Git Hub.

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 will place this package within a personal library in your Documents folder. Once you install a package is in the machine/location where you install it until you physically delete it. The code we have used by default connects to a cloud repository called CRAN that has a collection of R packages that meet a minimum set of quality criteria. CRAN is the official repository of all things R. It's a fairly safe place to get packages from. But beware, judging whether a package is good or not requires your input. We will come back to this several times during the semester to help you make wise choices regarding packages. Given that you are connecting to an online repository you will need an internet connection every time you want to install a package.

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 (the html link). 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. You will learn to love vignettes.

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? 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. 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")
```

```
## Colors cannot be applied in this environment :( Try using a terminal or RStudio.
```

```
##
##  --------------
## 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")
```

```
## Colors cannot be applied in this environment :( Try using a terminal or RStudio.
```

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

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).

##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, 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")
```

```
## Colors cannot be applied in this environment :( Try using a terminal or RStudio.
```

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

##More on objects

Isn't this a course on data analysis? Yes, of course, but before we get there, you need to get used to the basics of R and R Studio, which is what we will be doing in these early sessions. Let's go through some of these basics a bit more slowly to ensure you get them and then we will bring some data you can look at.

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. 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

Now that you are a bit more familiar with the idea of an object, we can start talking about a particular type of objects, specifically we are going to discuss **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 False, but you shoul
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 class)
```

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 plus
my_1st_vector + my_7th_vector #Adds the two vectors and auto-print the results (note how the sum was do
```

```
## [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"
```

##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 hash-tag 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 hash-tag/ 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 hash-tag on twitter).

##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.

##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!

##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 logical v
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")
data(package="fivethirtyeight") #This function will return all the data frames that are available in th
```

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.

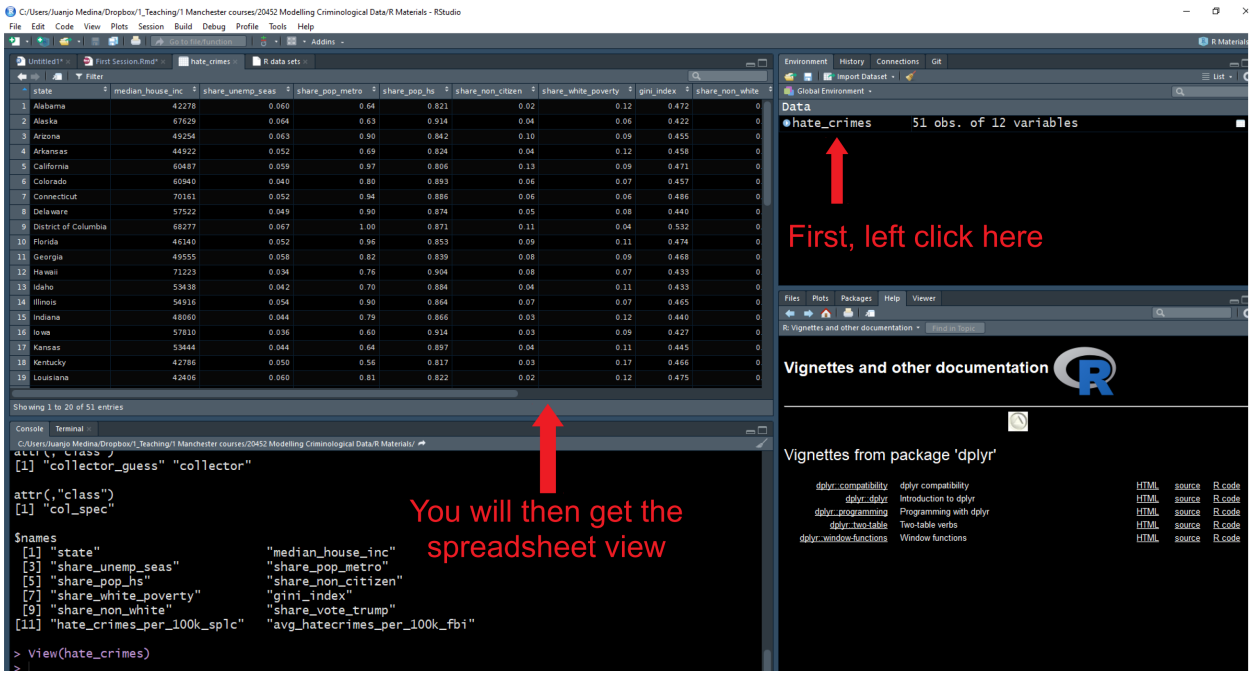
```
attributes(hate_crimes)
```

```
## $class
## [1] "tbl_df"     "tbl"        "data.frame"
##
## $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] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
## [47] 47 48 49 50 51
##
## $spec
## $cols
## $cols$state
## list()
## attr(,"class")
## [1] "collector_character" "collector"
##
## $cols$median_household_income
## list()
## attr(,"class")
## [1] "collector_integer" "collector"
##
## $cols$share_unemployed_seasonal
## list()
## attr(,"class")
## [1] "collector_double" "collector"
##
## $cols$share_population_in_metro_areas
## list()
## attr(,"class")
## [1] "collector_double" "collector"
##
## $cols$share_population_with_high_school_degree
## list()
## attr(,"class")
## [1] "collector_double" "collector"
##
## $cols$share_non_citizen
## list()
## attr(,"class")
## [1] "collector_double" "collector"
##
```

```
## $cols$share_white_poverty
## list()
## attr(,"class")
## [1] "collector_double" "collector"
##
## $cols$gini_index
## list()
## attr(,"class")
## [1] "collector_double" "collector"
##
## $cols$share_non_white
## list()
## attr(,"class")
## [1] "collector_double" "collector"
##
## $cols$share_voters_voted_trump
## list()
## attr(,"class")
## [1] "collector_double" "collector"
##
## $cols$hate_crimes_per_100k_splc
## list()
## attr(,"class")
## [1] "collector_double" "collector"
##
## $cols$avg_hatecrimes_per_100k_fbi
## list()
## attr(,"class")
## [1] "collector_double" "collector"
##
##
## $default
## list()
## attr(,"class")
## [1] "collector_guess" "collector"
##
## attr(,"class")
## [1] "col_spec"
##
## $names
##  [1] "state"                      "median_house_inc"
##  [3] "share_unemp_seas"           "share_pop_metro"
##  [5] "share_pop_hs"               "share_non_citizen"
##  [7] "share_white_poverty"        "gini_index"
##  [9] "share_non_white"            "share_vote_trump"
## [11] "hate_crimes_per_100k_splc"  "avg_hatecrimes_per_100k_fbi"
```

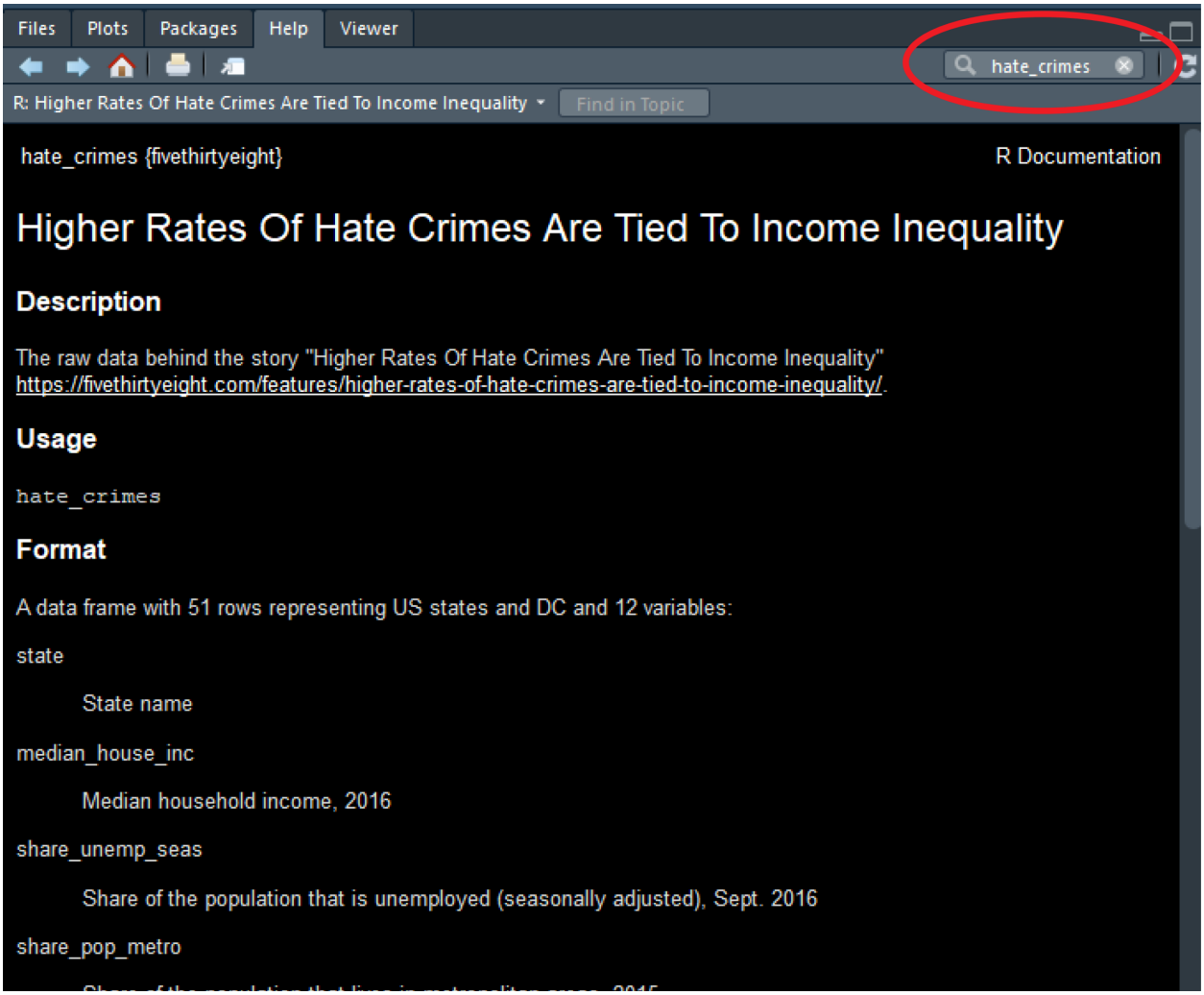
These results printed in the may console may not make too much sense to you at this point. We will return to this next week, so do not worry.

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.



##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
```

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

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. 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 summary statistics
n obs: 51
n variables: 12

Variable type: character

variable	missing	complete	n	min	max	empty	n_unique
state	0	51	51	4	20	0	51

Variable type: integer

variable	missing	complete	n	mean	sd	p0	p25	p50	p75	p100
median_house_inc	0	51	51	55223.61	9208.48	35521	48657	54916	60719	76165

Variable type: numeric

variable	missing	complete	n	mean	sd	p0	p25	p50	p75	p100
avg_hatecrimes_per_100k_fbi	1	50	51	2.37	1.71	0.27	1.29	1.99	3.18	10.95
gini_index	0	51	51	0.45	0.021	0.42	0.44	0.45	0.47	0.53
hate_crimes_per_100k_splc	4	47	51	0.3	0.25	0.067	0.14	0.23	0.36	1.52
share_non_citizen	3	48	51	0.055	0.031	0.01	0.03	0.045	0.08	0.13
share_non_white	0	51	51	0.32	0.16	0.06	0.2	0.28	0.42	0.81
share_pop_hs	0	51	51	0.87	0.034	0.8	0.84	0.87	0.9	0.92
share_pop_metro	0	51	51	0.75	0.18	0.31	0.63	0.79	0.9	1
share_unemp_seas	0	51	51	0.05	0.011	0.028	0.042	0.051	0.058	0.073
share_vote_trump	0	51	51	0.49	0.12	0.04	0.41	0.49	0.57	0.7
share_white_poverty	0	51	51	0.092	0.025	0.04	0.075	0.09	0.1	0.17

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.

There's also many different ways of producing graphics in R. In this course we rely on a package called *ggplot2*. It is already in the clusters, but if you are using your own laptop will need to install it first and then load it.

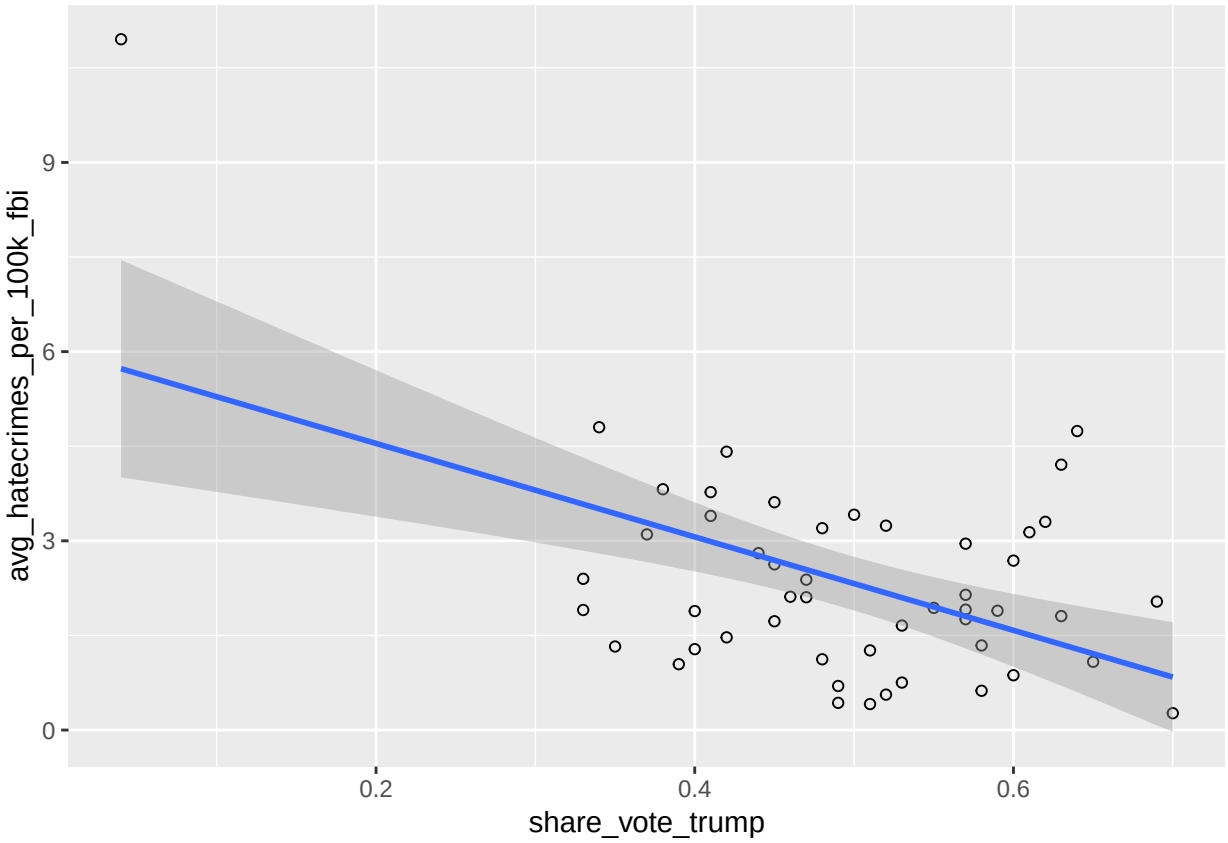
```
library(ggplot2)
```

Then we will use one of its functions to create a scatterplot. Don't worry about understanding this code below, we will have a whole session on the ggplot function:

```
ggplot(hate_crimes, aes(x=share_vote_trump, y=avg_hatecrimes_per_100k_fbi)) +
    geom_point(shape=1) +
     geom_smooth(method=lm)
```

```
## Warning: Removed 1 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 1 rows containing missing values (geom_point).
```



What do you think this graphic is telling you?

##Quitting RStudio

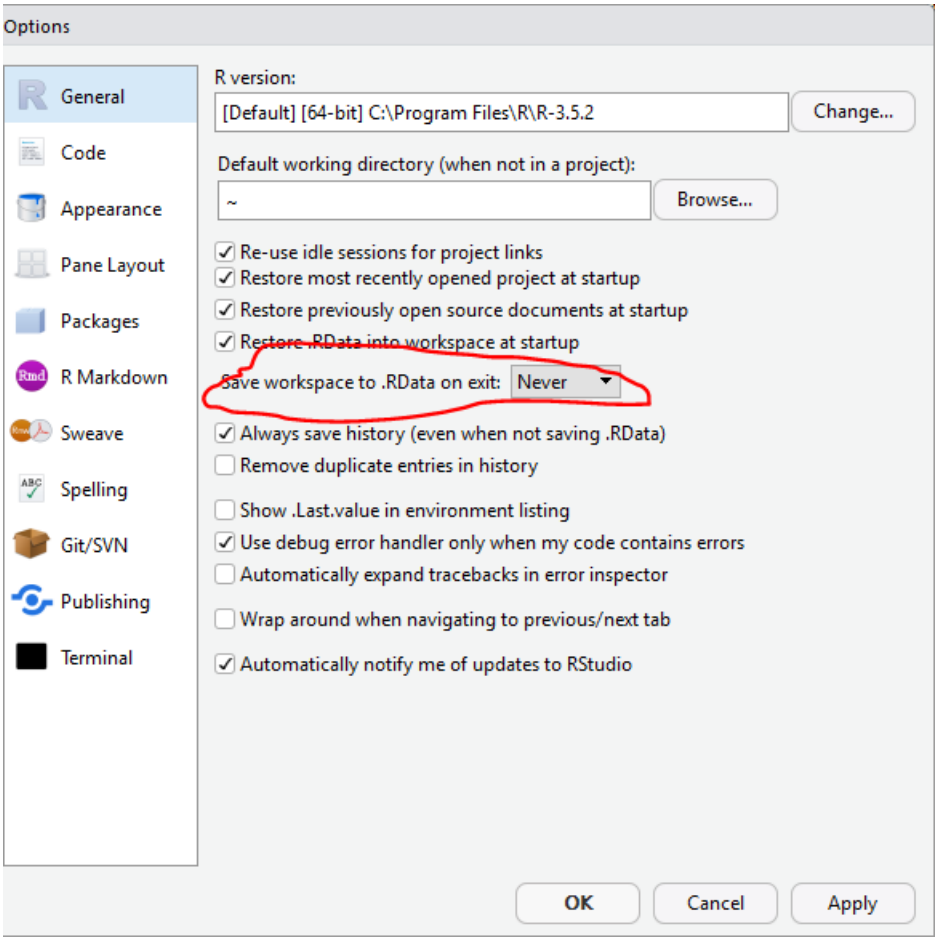
At some point, you will quit your R/R Studio session. I know, hard to visualise, right? Why would you want to do that? Anyhow, when that happens R Studio will ask you a hard question: "Save work space image to bla bla bla/.RData?" What to do? What does that even mean?

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.



#Causality in randomised experiments (Week 2)

0.1 Causality in social sciences

In today's week we will refresh some of the themes you have explored in previous research methods courses. We are going to talk specifically about causality. This is one of the central concepts 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 typically involves making comparisons between cases that have been subject to an intervention or that present some sort of attribute that we think may have a causal effect to cases that have not being subject to the causal process we are trying to establish. But by your previous training, you should already know that not all kind of comparisons are the same. In the first semester we discussed the differeces between experimental and observational studies and we established how 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 and more in living "la vida loca".

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 profesisonal 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 randomized 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.

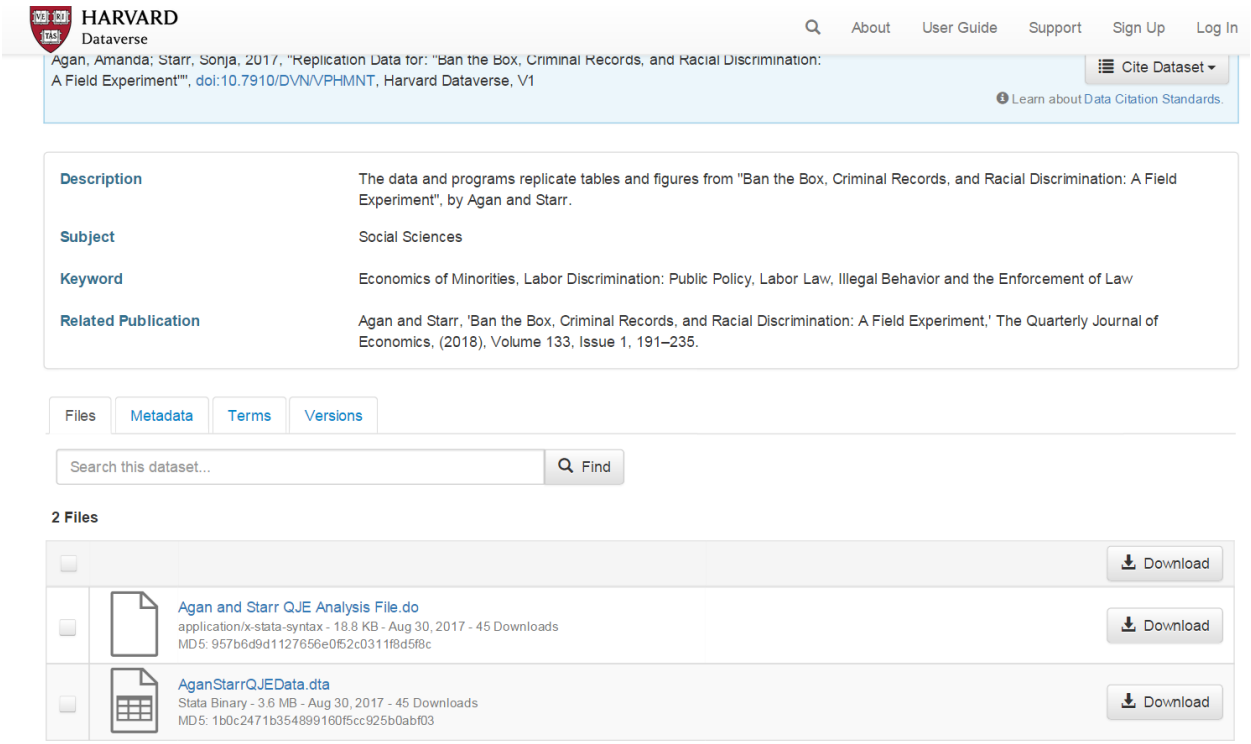
In this session we will look at data from a randomised trial that tried to establish whether there is discrimination in the labour market against former offenders. In doing so, we will also learn various functions used in R to read data, transform data, and to obtain summary statistics for groups. We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

0.2 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. In this research project, the authors tried to answer the question of whether criminal antecendents and other personal characteristics have an impact on access to employment.

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 fictious 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 reproducibly standards when doing their research we can access this data from the *Harvard Dataverse* (a repositoty 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 analytical code pertaning 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. 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 should now have a new object in your global environment. Check the dimensions of the file.

0.3 Getting a sense for your data

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.

As Mara Averick (somebody you want to follow in twitter at **?** 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; mislabelled 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 you will see that each column has a variable name and underneath a longer and more meaninful *variable label* that tells you what each variable means.

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. That's what the class() function also says, but they introduce some tweaks to make life easier.

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 a modern dialect of R. This set of packages is called the **tidyverse**. Tibbles are dataframes that have been optimised within 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 if its an integer (number) or factor.

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

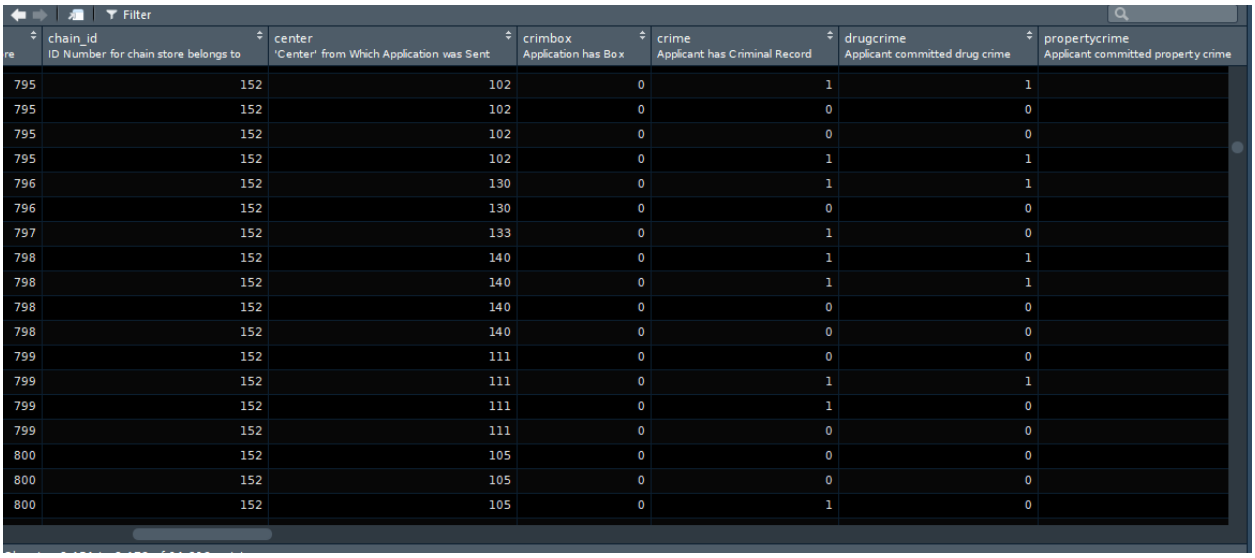
```
class(banbox$crime)
```

```
## [1] "haven_labelled"
```

```
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.



	chain_id ID Number for chain store belongs to	center 'Center' from Which Application was Sent	crimbox Application has Box	crime Applicant has Criminal Record	drugcrime Applicant committed drug crime	propertycrime Applicant committed property crime
795	152	102	0	1	1	
795	152	102	0	0	0	
795	152	102	0	0	0	
795	152	102	0	1	1	
796	152	130	0	1	1	
796	152	130	0	0	0	
797	152	133	0	1	0	
798	152	140	0	1	1	
798	152	140	0	1	1	
798	152	140	0	0	0	
798	152	140	0	0	0	
799	152	111	0	0	0	
799	152	111	0	1	1	
799	152	111	0	1	0	
799	152	111	0	0	0	
800	152	105	0	0	0	
800	152	105	0	0	0	
800	152	105	0	1	0	

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. 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"
##
```

```
## $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.

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.

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))
```

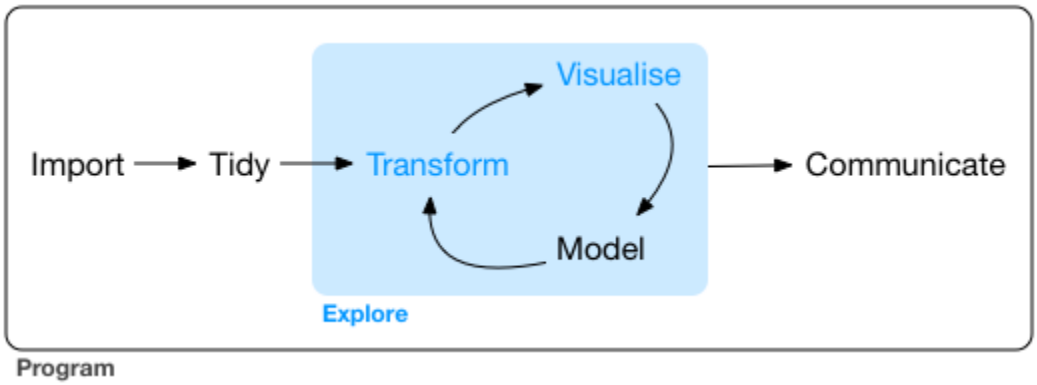
```
## [1] 0.4999662
```

This code is exploiting the mathematical fact that the mean of binary outcomes (0 or 1) gives you the proportion of 1s in your data. 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 documenation 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.

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, 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.

0.4 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.

R offers a great deal of flexibility in how to do this kind of thing, Here we are going to illustrate some of the functionality of the *dplyr* package for data carpentry. 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. 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.
```

0.5 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.

```
#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 than 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 funciton 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.

0.6 Using dplyr for grouped operations

So far we have used dplyr sinle 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 (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 asignation 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.

HOMEWORK 1: Use what we have learned so far to see if there is an interaction effect between race and criminal records. That is, what would happen if we compare 4 groups whites with no records, whites with records, blacks with no records, and blacks with records. What do you think? Is there an interaction effect? In other words, is the impact of having a criminal record more accentuated for either of the two racial groups? Be ready to answer these questions before you attempt the Blackboard test

0.7 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.

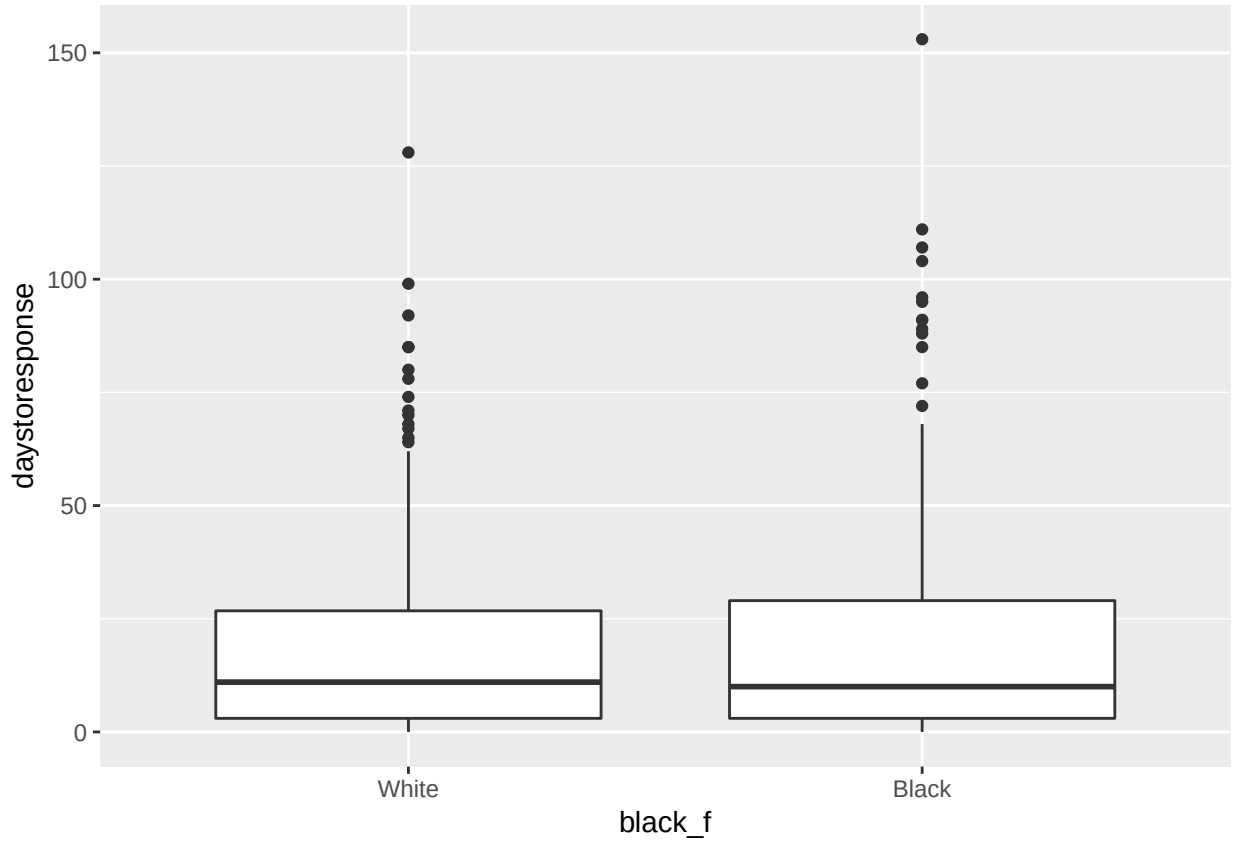
We could do as before and get results by groups. Let's look at the impact of race on days to response:

```
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
```

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?

Homework 2: For this course unit you need to write a coursework assignment in which you will use data from a social survey to explore relationships between various variables.

Specifically, you will be allowed to use. So you will need to pick one of them and then select a topic (of those listed below) to explore with each of these surveys: - The 2014 wave of the European Social Survey, which you can access here and explore here. From this dataset you will be able to explore the following topic: "Perception of an association between crime and immigration - The 2013-2014 Crime Survey for England and Wales (the Teaching Dataset version) which you can access here or explore here. From this dataset you will need to focus on one of two topics"violent victimisation","perceptions of antisocial behaviour", or"confidende in the police".

As part of this week homework you need to let us know which survey and which topic you want to focus on for your coursework assignment. *Think about which of these topics appeals the most to you when making your decision. Be ready to answer questions about this when starting the Blackboard test.*

You can explore the datasets and what other variables are available. For your coursework you will need to identify other variables in these surveys that you think may be associated or helpful to understand why people vary in worry of crime, attitudes toward punishment, etc. At this point you don't need to identify those other variables, but you may find it useful to explore what's available in those datasets using NESSTAR (a guide for how to use NESSTAR is available here)and the existing documentation for those studies

```
## [1] "C/C/C/C/C/en_GB.UTF-8"
```

#Data visualisation with R (Week 3)

##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 thorugh them, I 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 ggplot is to implement the grammar of graphics. The `ggplot2` package has excellent online documentation.

If you don't already have the package installed, 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 defines various components of the graphic. Some of the most important are:

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

-**The geoms**: They describe the objects that represent the data (e.g., points, lines, polygons, etc.).

-**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.

##Anatomy of a plot

Essentially the philosophy behind this as 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:

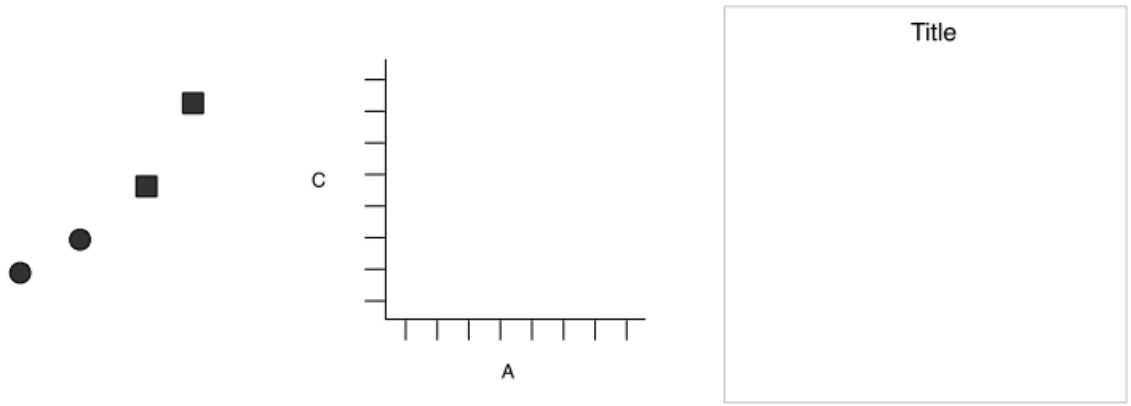


Figure 1. Graphics objects produced by (from left to right): geometric objects, scales and coordinate system, plot annotations.

This will result in a final plot:

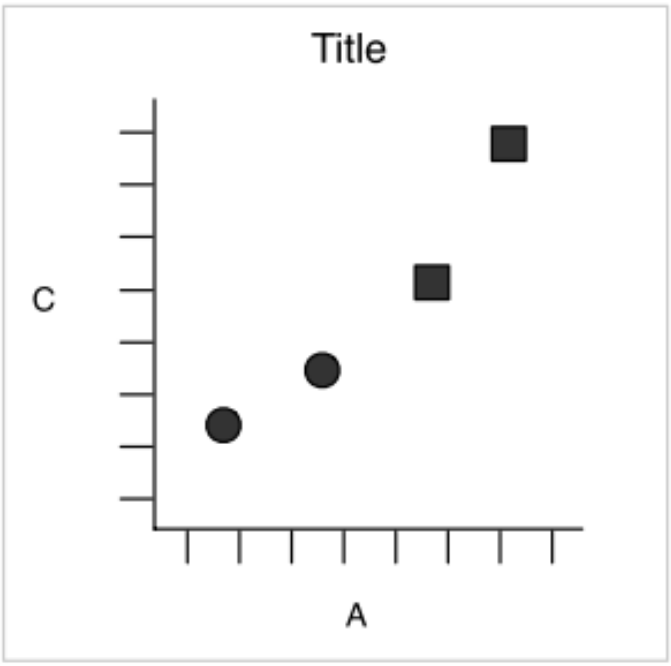


Figure 2. The final graphic, produced by combining the pieces in Figure 1.

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

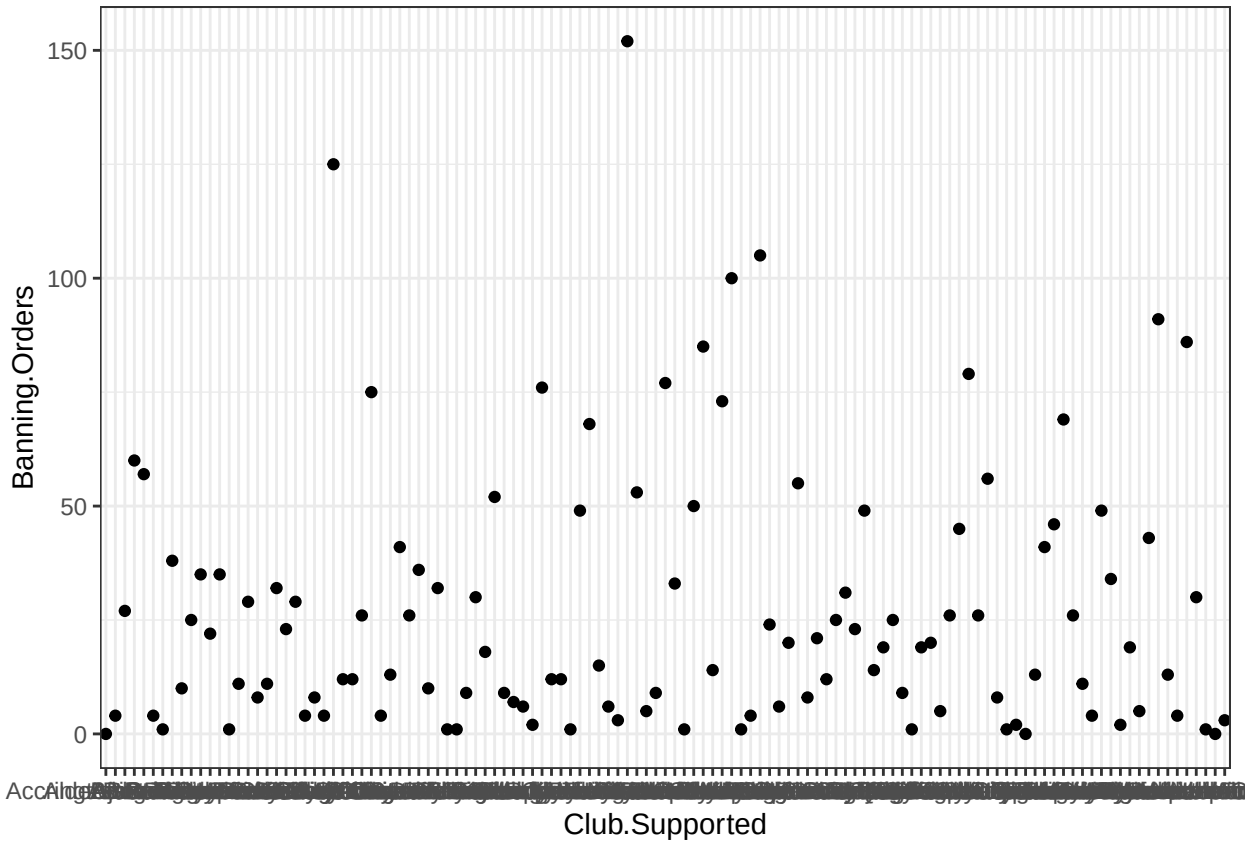
Let's have a look at some data about banning orders for different football clubs.

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

```
fbo_url <- "https://raw.githubusercontent.com/maczokni/R-for-Criminologists/master/fbo-by-club-supporte
fbo <- read.csv(url(fbo_url))
```

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
```

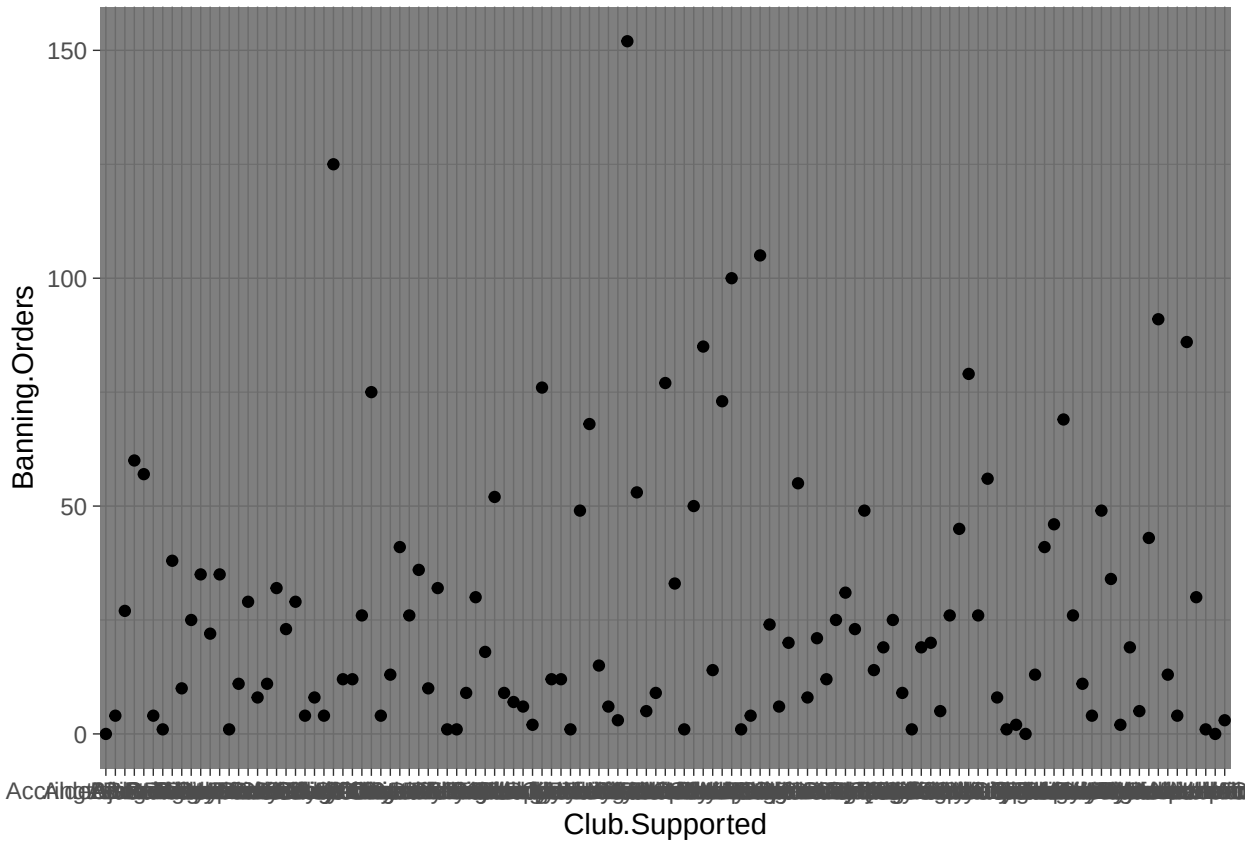


The first line above begins a plot by calling the `ggplot()` function, and putting the data into it. You have to name your dataframe, and then, within the `aes()` command you pass the specific variables which you want to plot. In this case, we only want to see the distribution of one variable, banning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

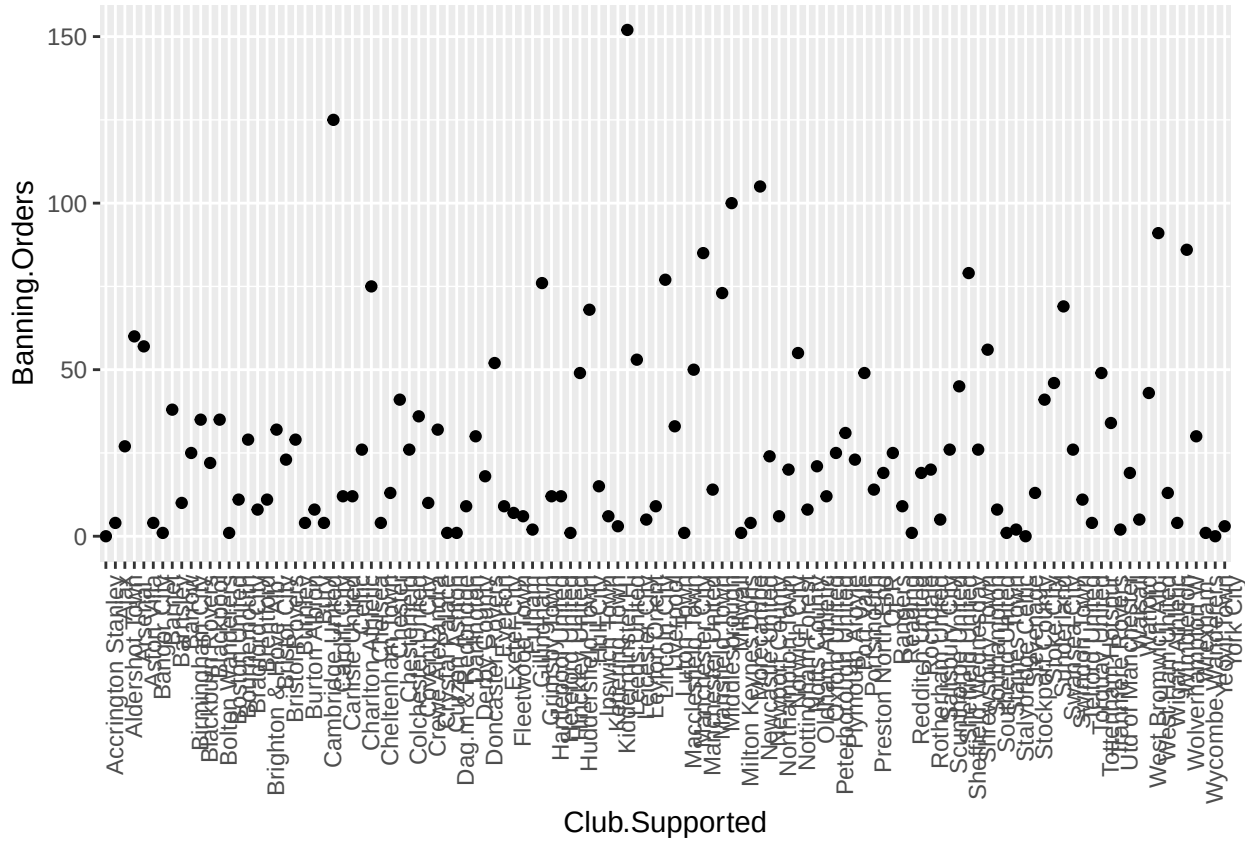
The third line is where we can tweak the display of the graph. Here I used theme_bw() which is a nice clean theme. You can try with other themes. To get a list of themes you can also see the resource here.

```
ggplot(data = fbo, aes(x = Club.Supported, y=Banning.Orders)) +      #data
   geom_point() +                                                    #geometry
  theme_dark()                                                       #backgroud coordinate system
```



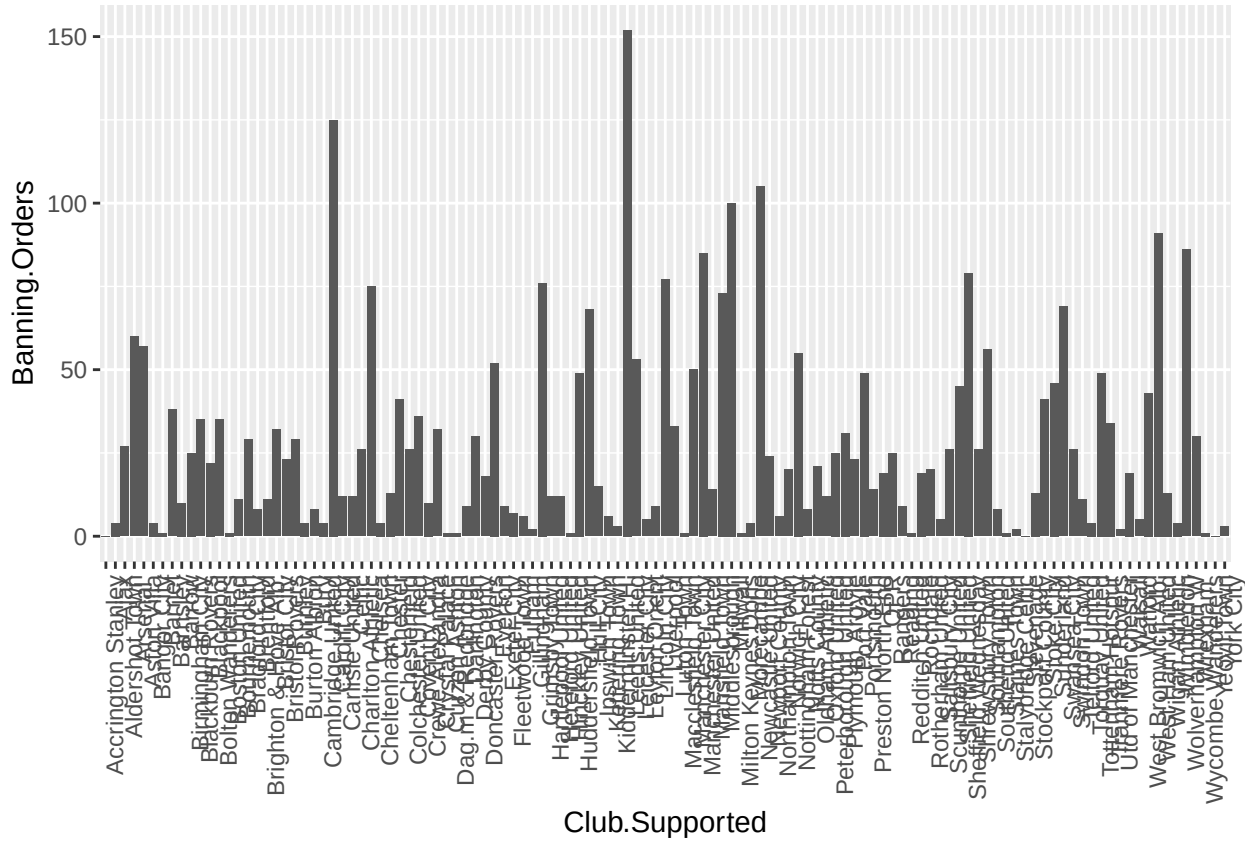
Changing the theme is not all you can do with the third element. For example here you can't really read the axis labels, because they're all overlapping. One solution would be to rotate your axis labels 90 degrees, with the following code: `axis.text.x = element_text(angle = 90, hjust = 1)`. You pass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

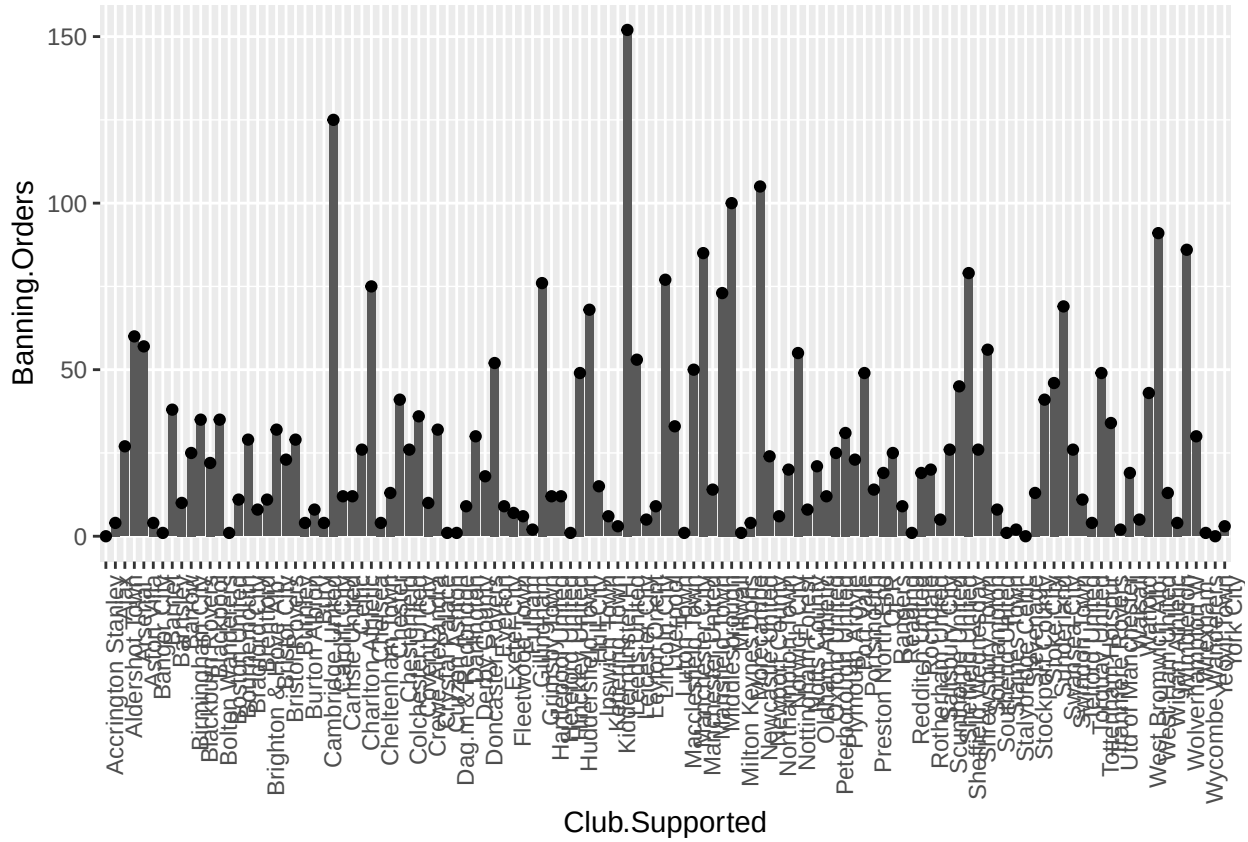


You might notice here we pass an argument `stat = "identity"` to `geo_bar()` function. This is because you can have a bar graph where the height of the bar shows frequency (stat = "count"), or where the height is taken from a variable in your dataframe (stat = "identity"). Here we specified a y-value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

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

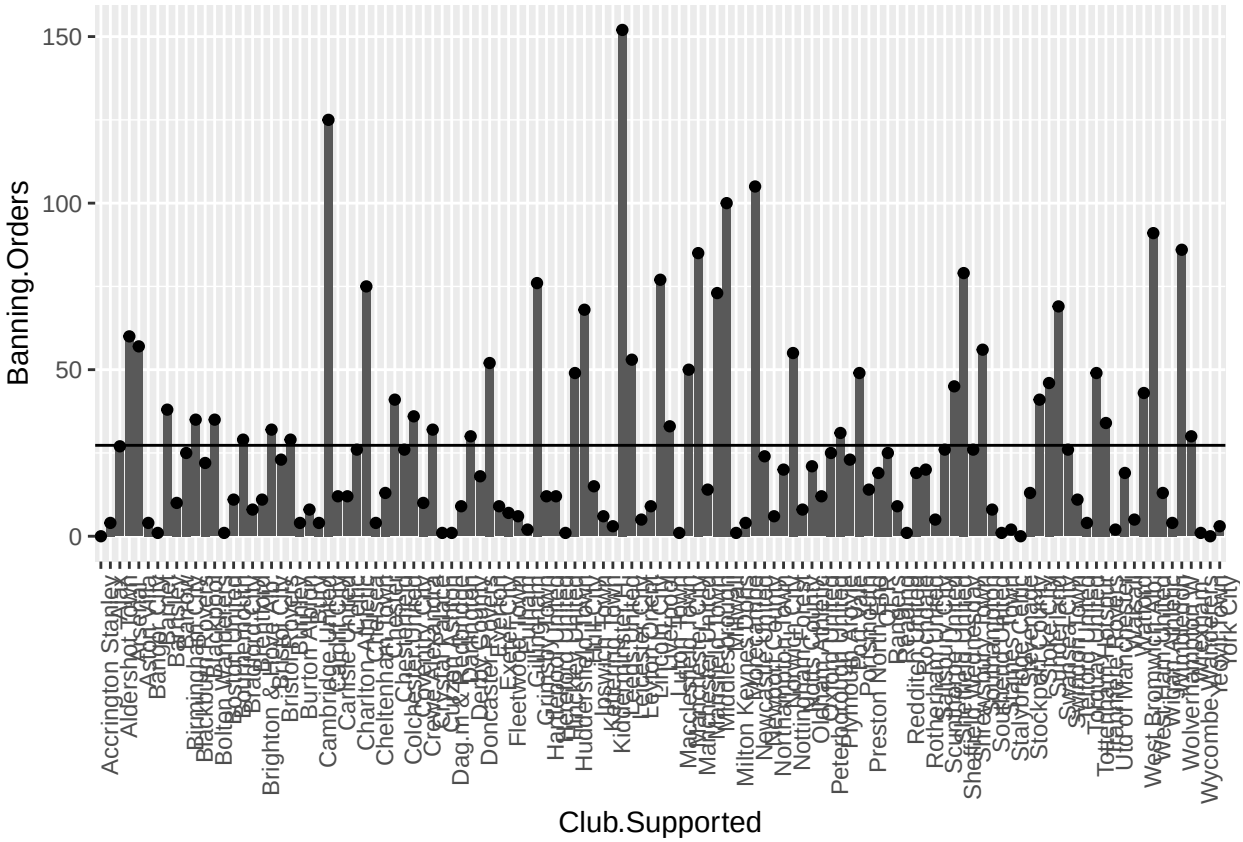
Well this is the beauty of the layering approach of ggplot2. You can layer on as many geoms as your little heart desires! XD

```
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))
```

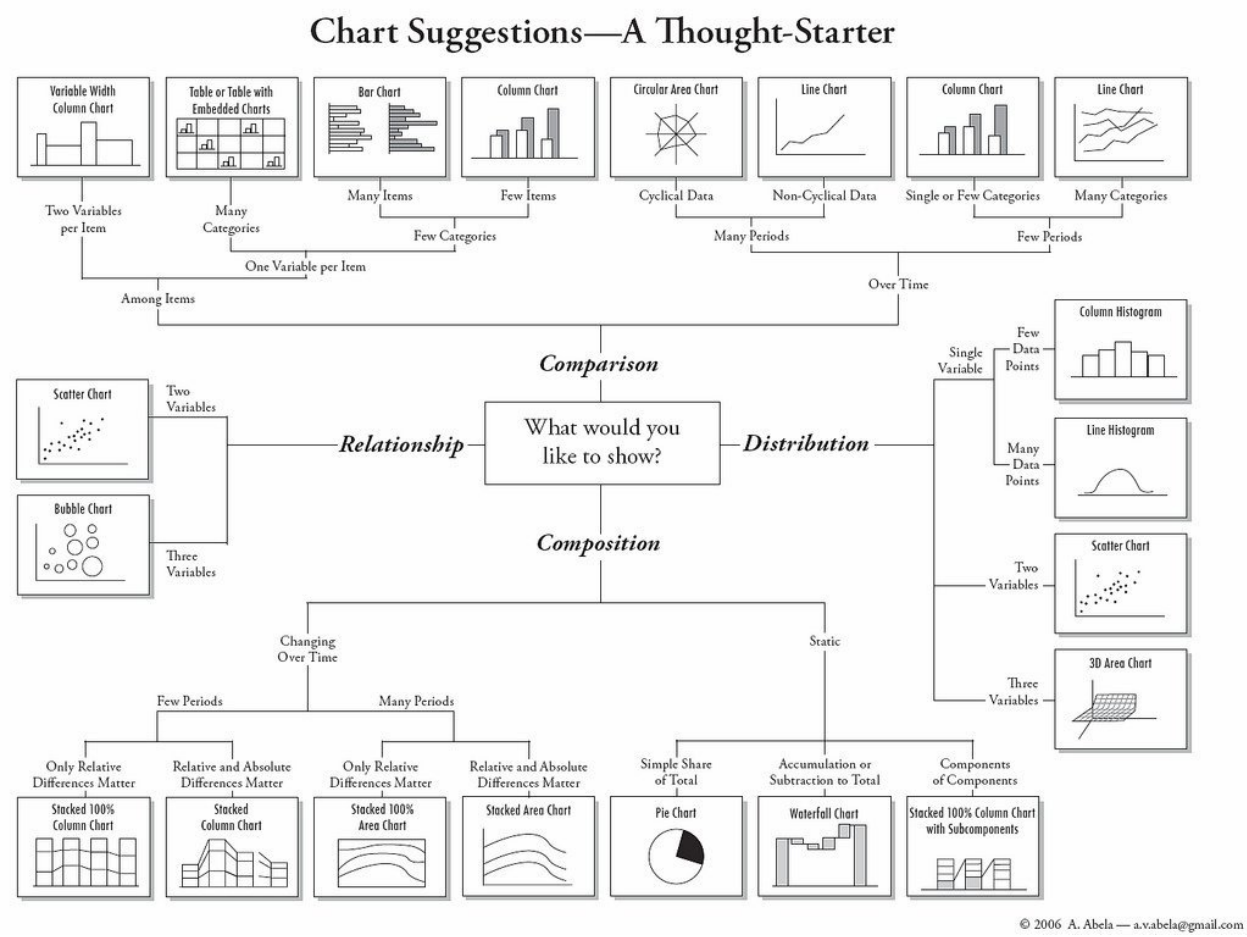


This is basically all you need to know to build a graph! So far we have introduced a lot of code some of which you may not fully understand. Don't worry too much, we just wanted to give you a quick introduction to some of the possibilities. Later in the session we will go back to some of these functions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

##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.

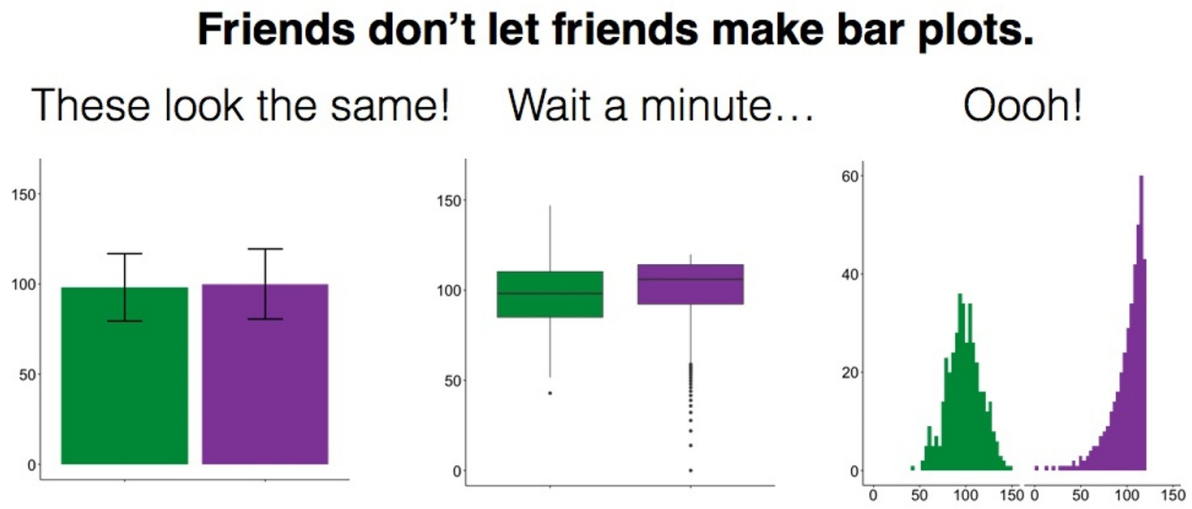
There is also a vast amount of research into what works in displaying quantitative information. The classic book is by Edward Tufte [1], but since him there are many other researchers as well who focus on approaches to displaying data. Two useful books you may want to consider are Few (2012) [2] and Cairo (2016) [3]. Claus Wilke is also producing a textbook that is work in progress but 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 villanified. For example, the pie chart is one such example. A lot of people really dislike pie charts, eg 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 can hide important features of your data, and might not be the most appropriate means for comparison:

[1] Various of these can be obtained invoking the `pR2()` function from the `pscl` package.
[2] Alternatively one could use the `ROCR` package. For details, see here.
[3] This is a reasonable explanation of how to interpret R-Squared.



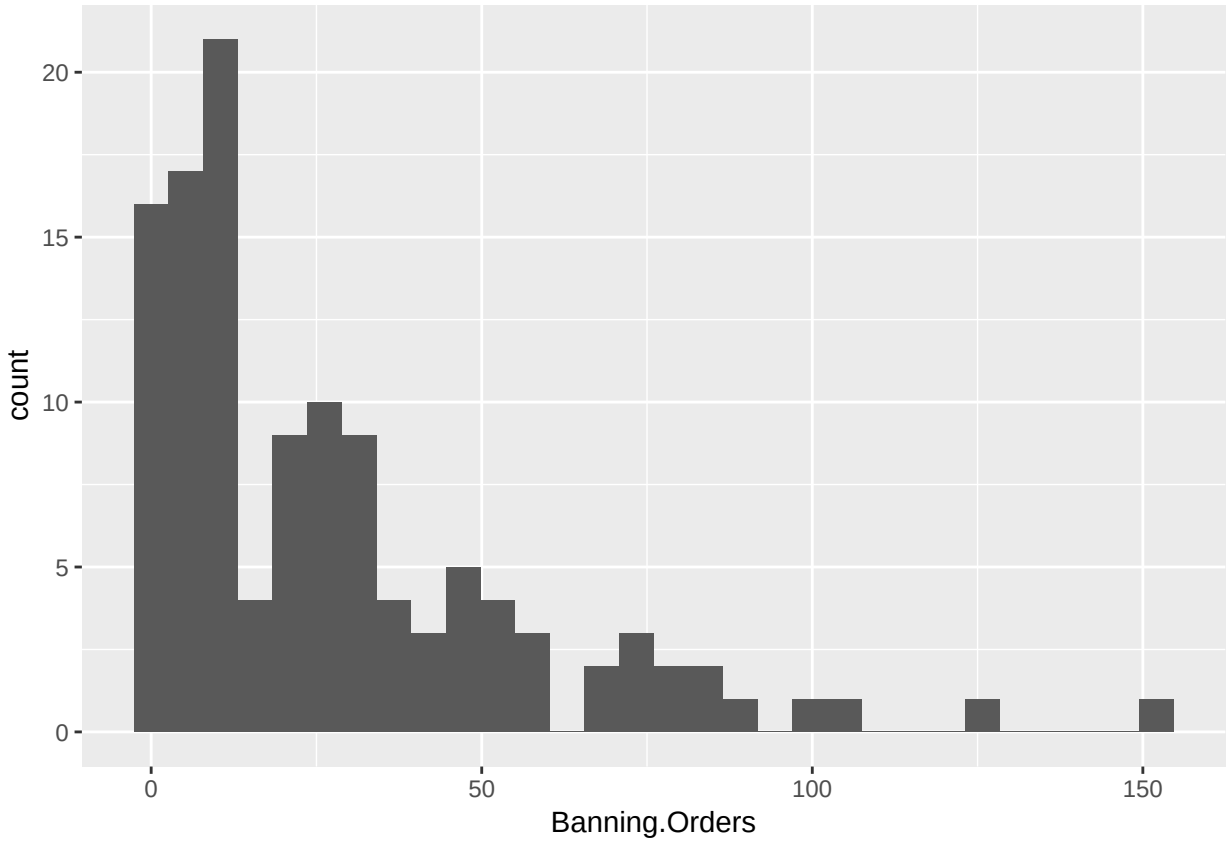
This has lead to a kickstarter campaign around actually banning bar plots...!

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. Other books that focus on

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

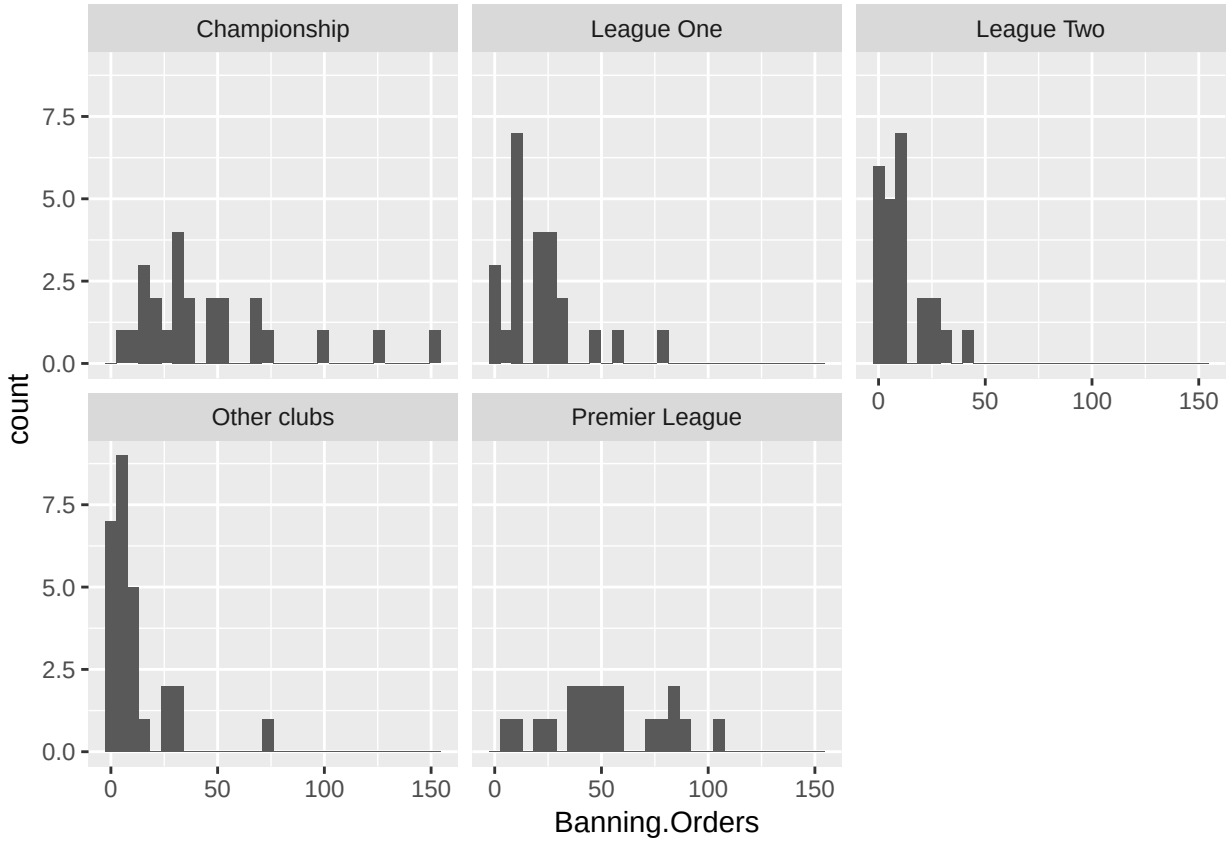
One suggestion is to make a histogram for each one. You can use ggplot's facet_wrap() option to split graphs by a grouping variable. For example, to create a histogram of banning orders you write:

```
ggplot(data = fbo, aes(x = Banning.Orders)) +
  geom_histogram()
```



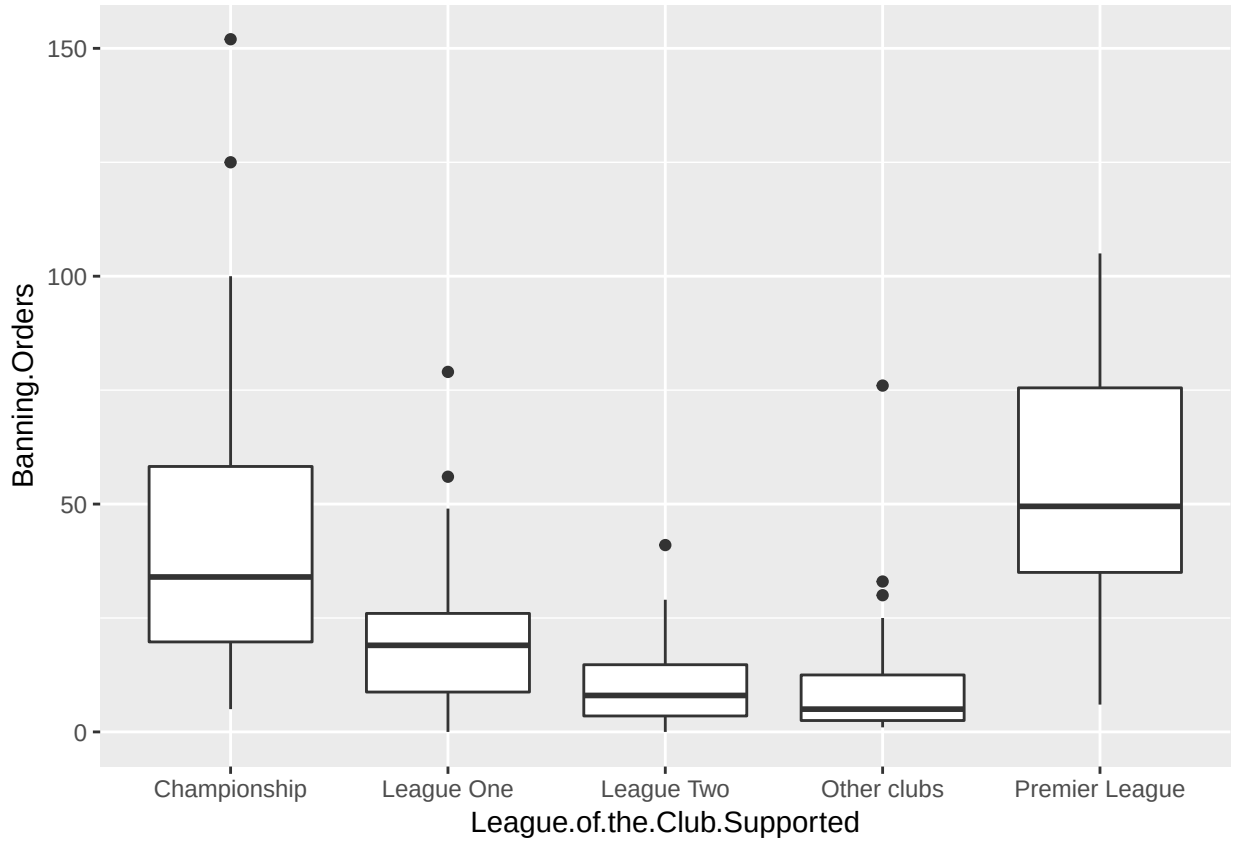
Now to split this by League.of.the.Club.Supported, you use facet_wrap() in the coordinate layer of the plot.

```
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 for showing distribution. So let's try:

```
ggplot(data = fbo, aes(x = League.of.the.Club.Supported, y = Banning.Orders)) +
  geom_boxplot()
```

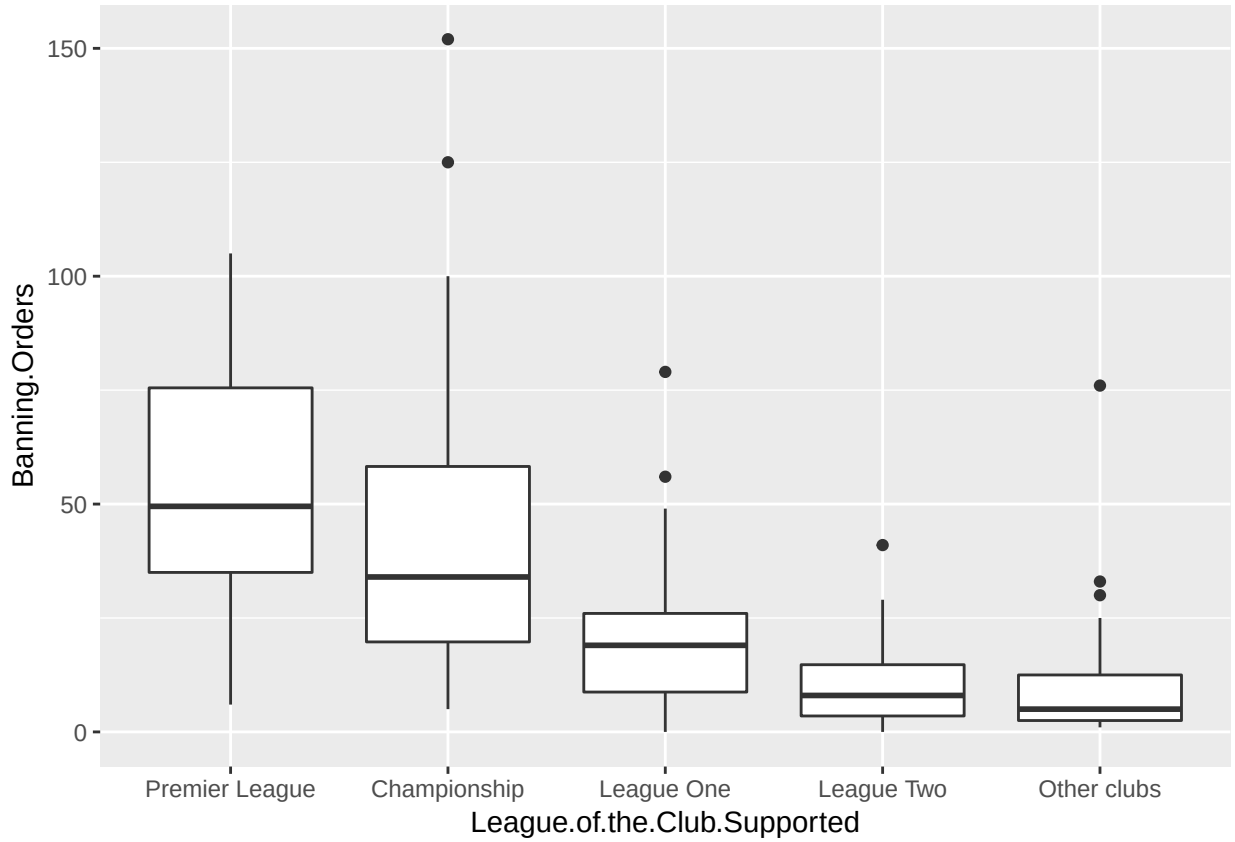


This makes the comparison significantly easier, right? But the order is strange! Remember we talked about factors in previous weeks? Well the good thing about factors is that we can arrange them in their natural order. If we dont' describe an order, then R uses the alphabetical order. So let's reorder our factor. To do that we are specifying the levels in the order in which we want to be embedded witin 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("Premier League
```

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 ggplot package stopping a bit more on each of them.

##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)
```

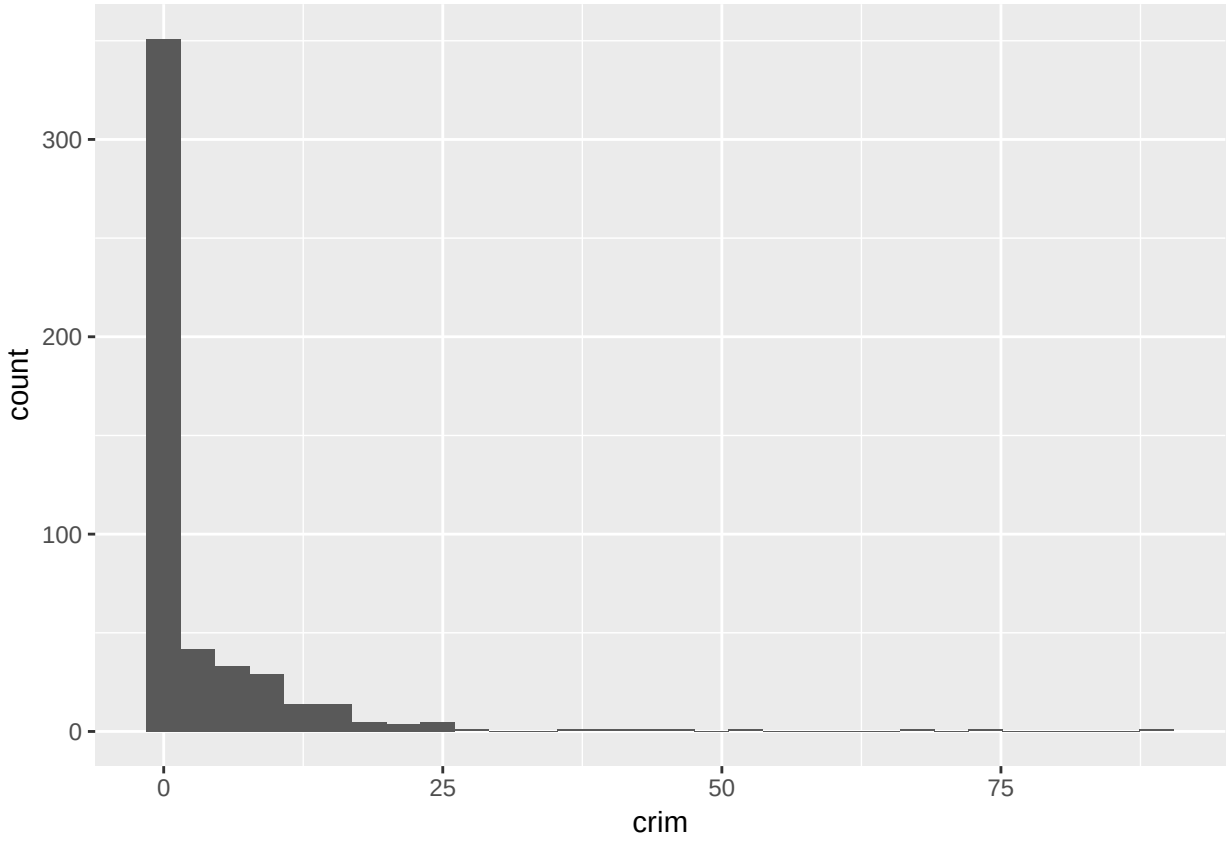
This package has a dataframe called "Boston". This has data about Housing Values in suburbs of Boston (USA). To access the codebook (how you find out what variables are) use the "?".

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 equivalent code:

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

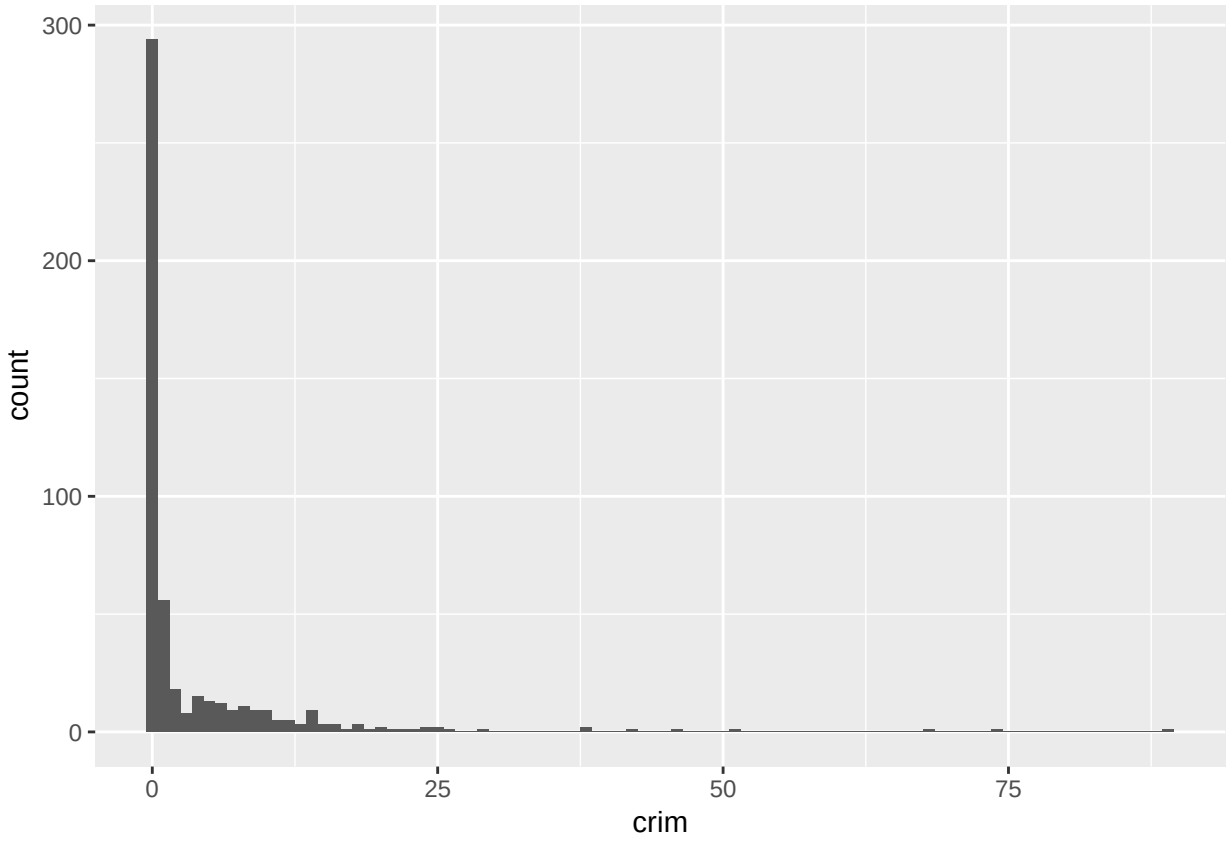
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



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, 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.
- **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"
```

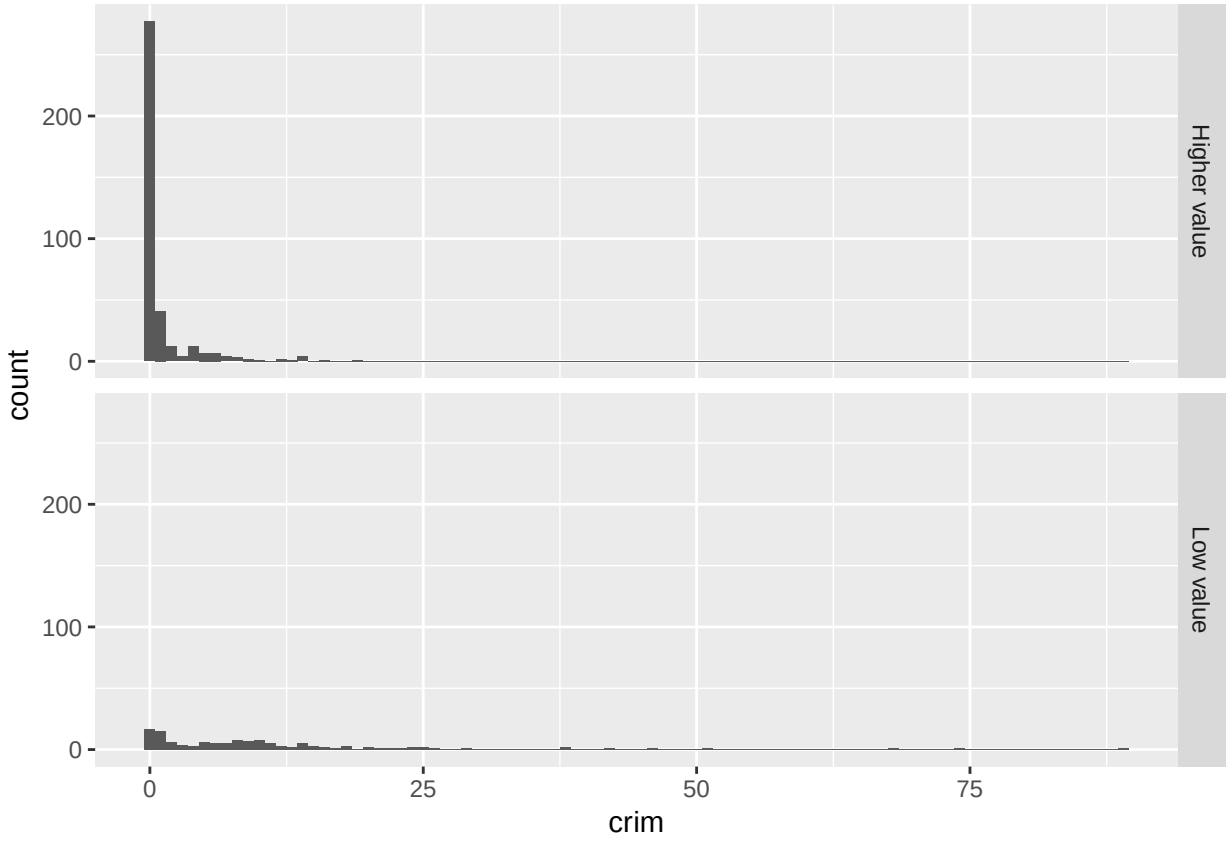
```
Boston$lowval <- as.factor(Boston$lowval)
class(Boston$lowval)
```

```
## [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 ~ .)
```

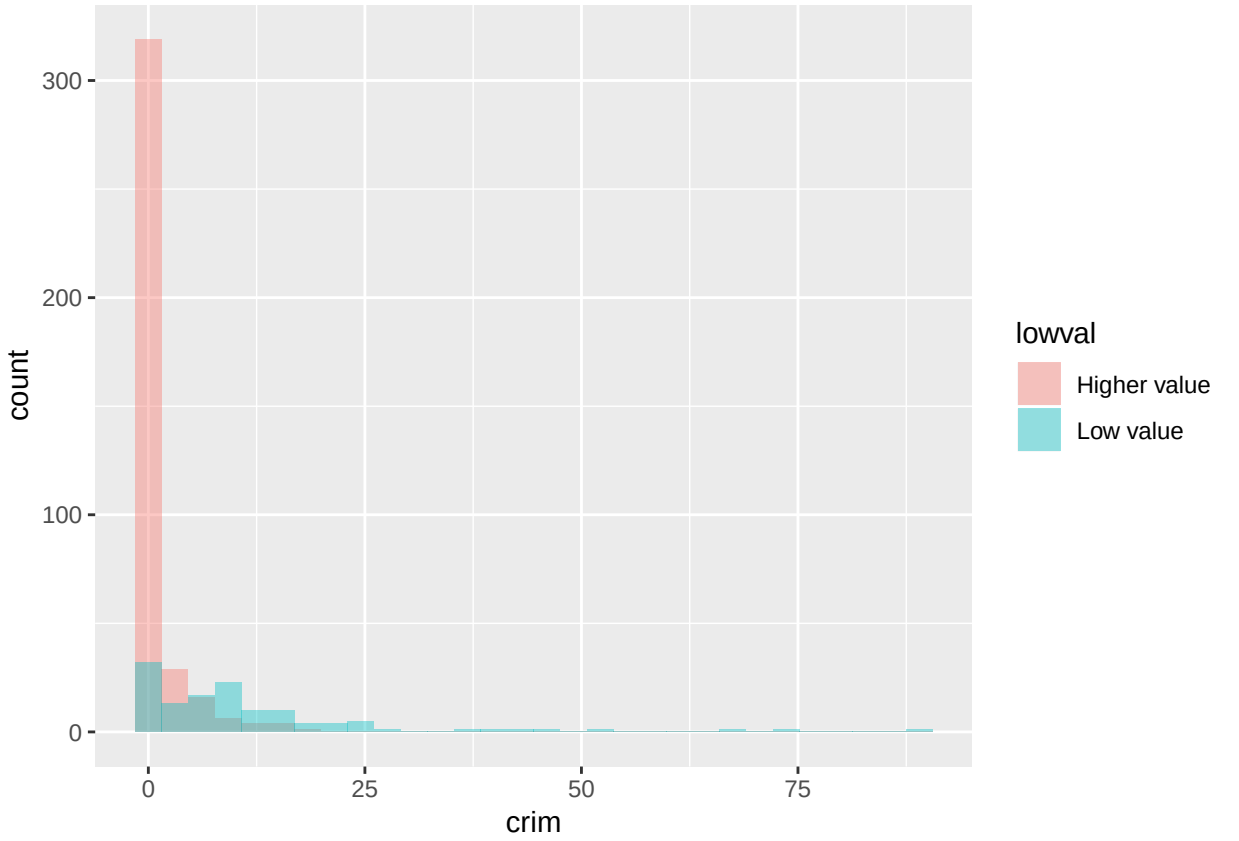
[4]This blog post provides a nice animation of the confidence interval and hypothesis testing.



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").

Instead of using facets, we could overlay the histograms with a bit of transparency. Transparencies work better 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)
```

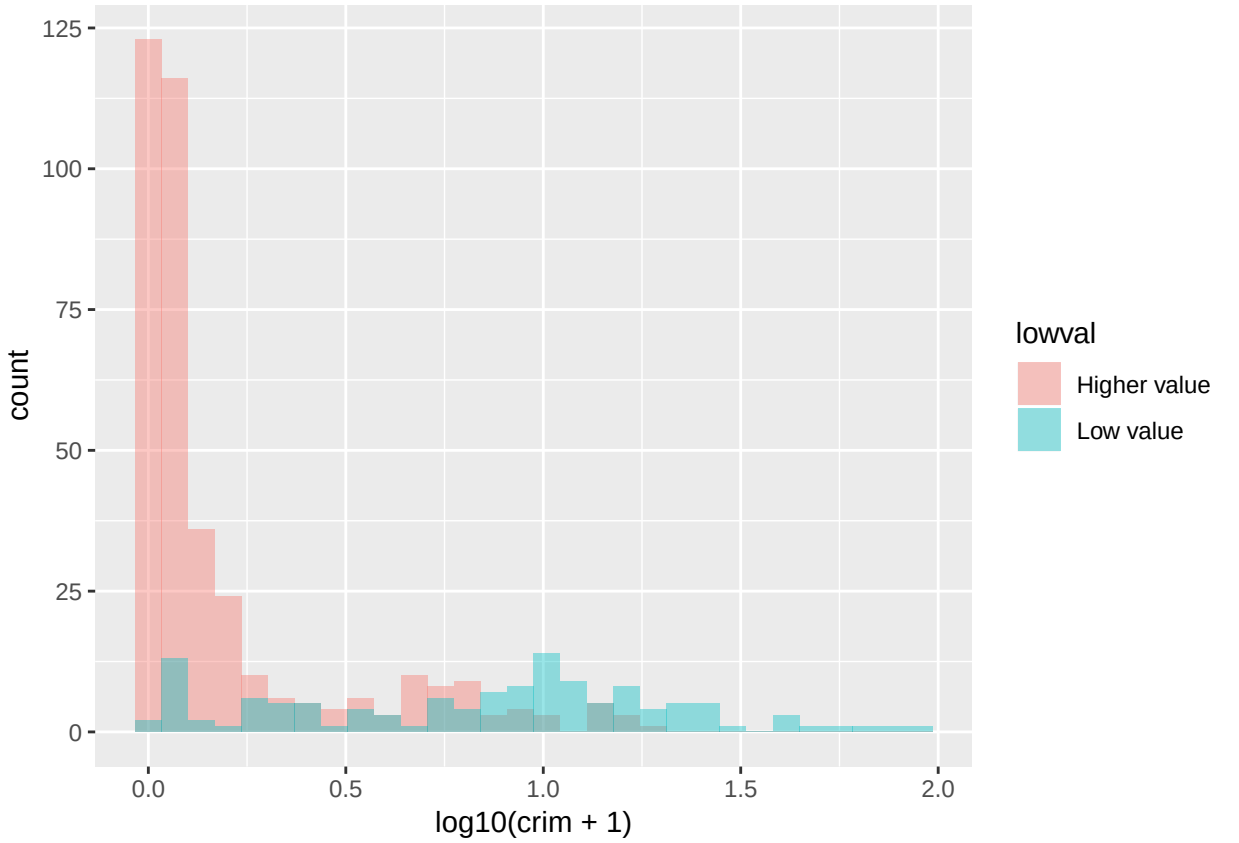


In the code above, the `fill` argument identifies the factor variable in the dataset grouping the cases. Also, `position = identity` tells R to overlay the distributions and `alpha` asks for the degree of transparency, a lower value (e.g., 0.2) will be more transparent.

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 a nice illustration of the Simpon's Paradox, a well known example of ommitted variable bias.

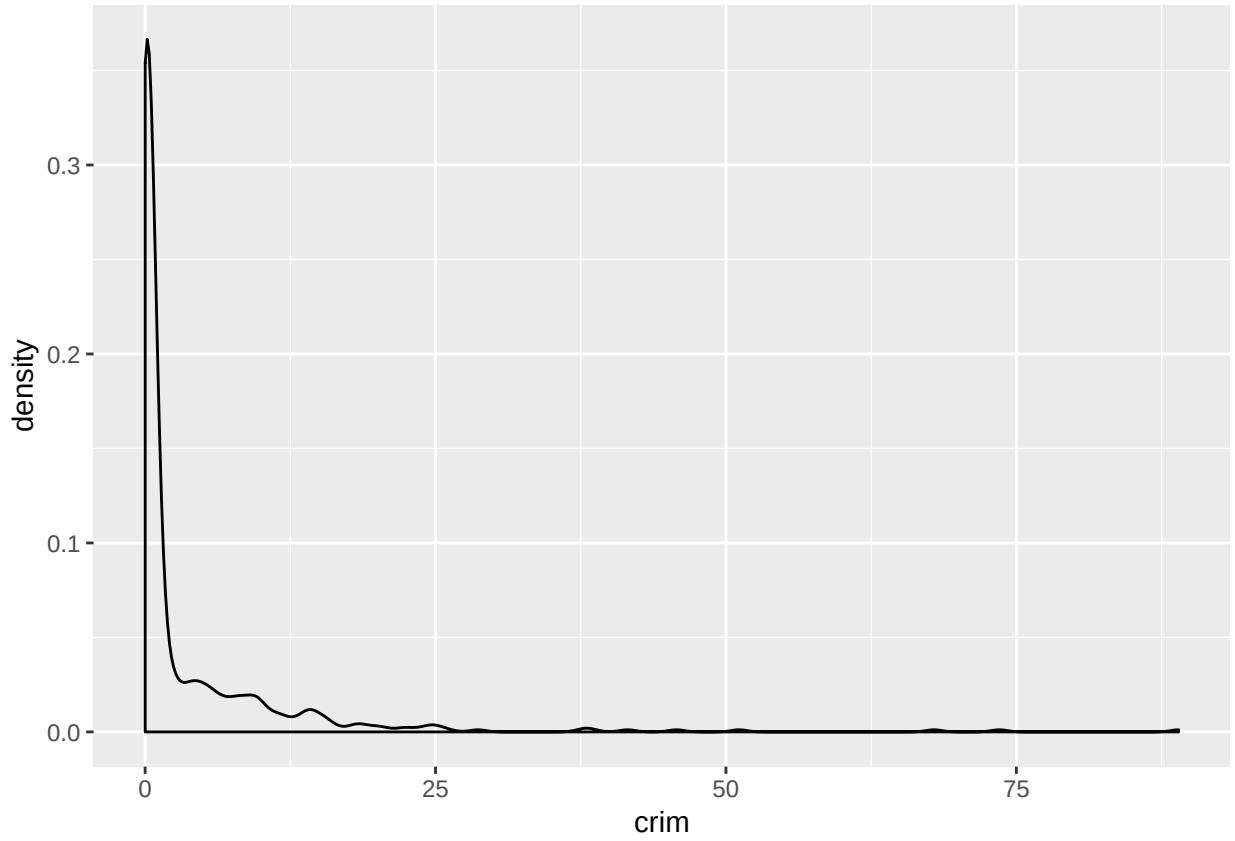


The plot now is a bit clearer. It seems pretty evident that the distribution of crime is slightly different between these two types of towns.

##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()
```

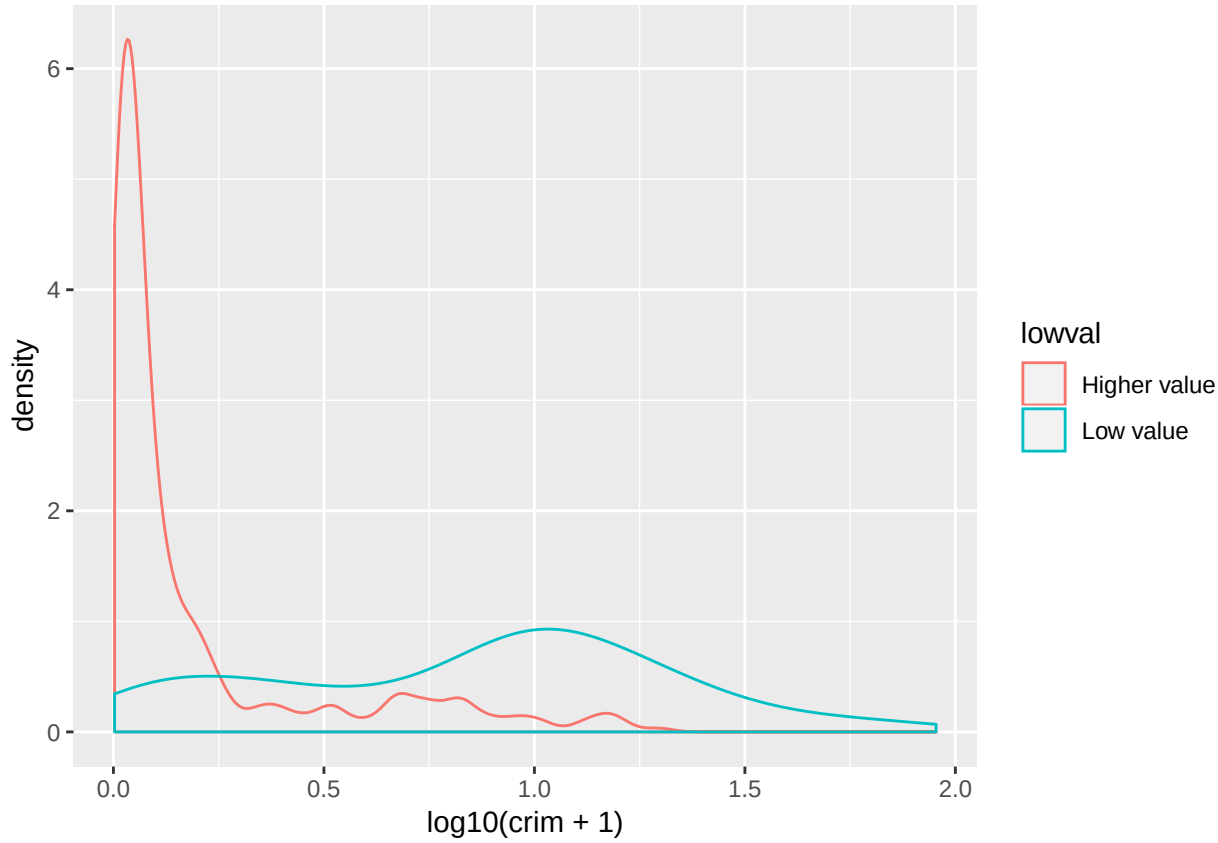


In a density plot, we attempt to visualize the underlying probability distribution of the data by drawing an appropriate continuous curve. So in a density plot then the area under the lines sum to 1 and the Y, vertical, axis now gives you the estimated (guessed) probability for the different values in 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).

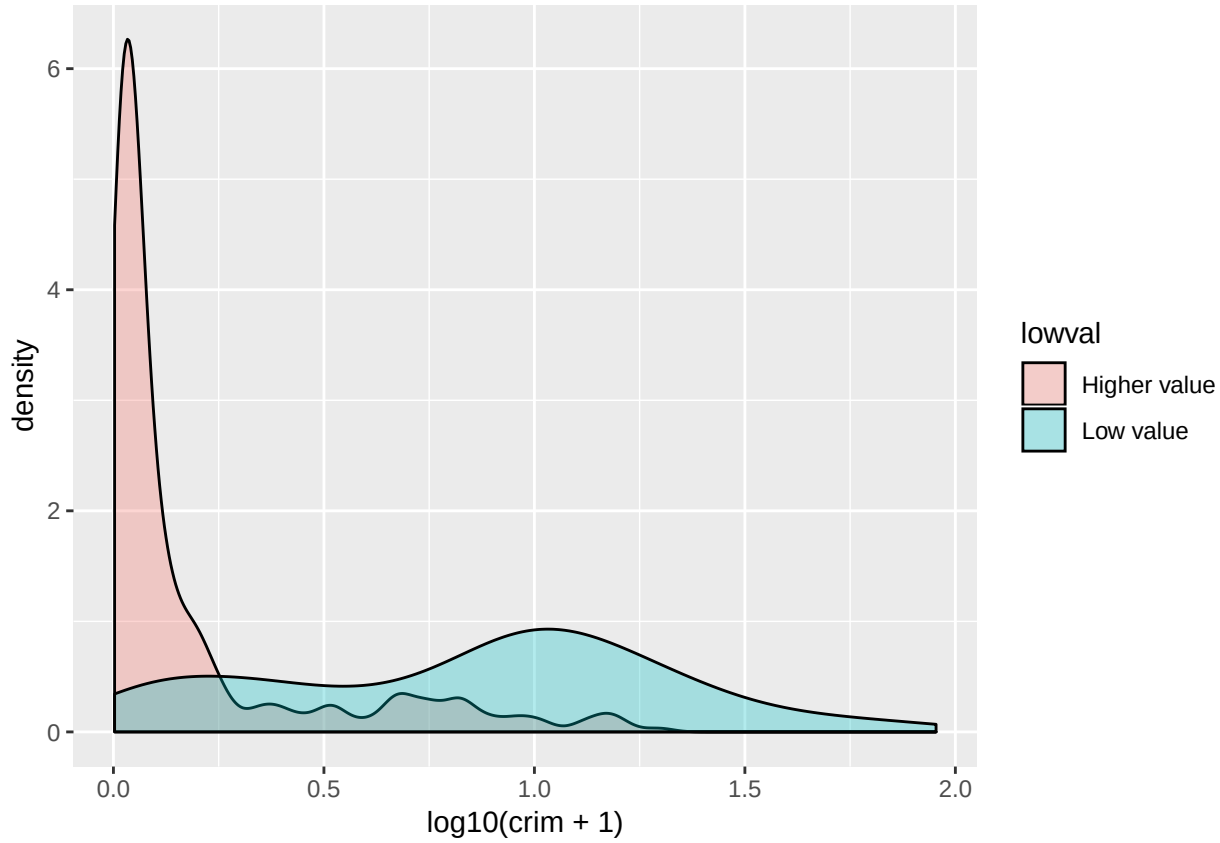
You can also use this to compare the distribution of a quantitative variable across the levels in a categorical variable (factor) and, as before, is possibly better to take the log of skewed variables such as crime:

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



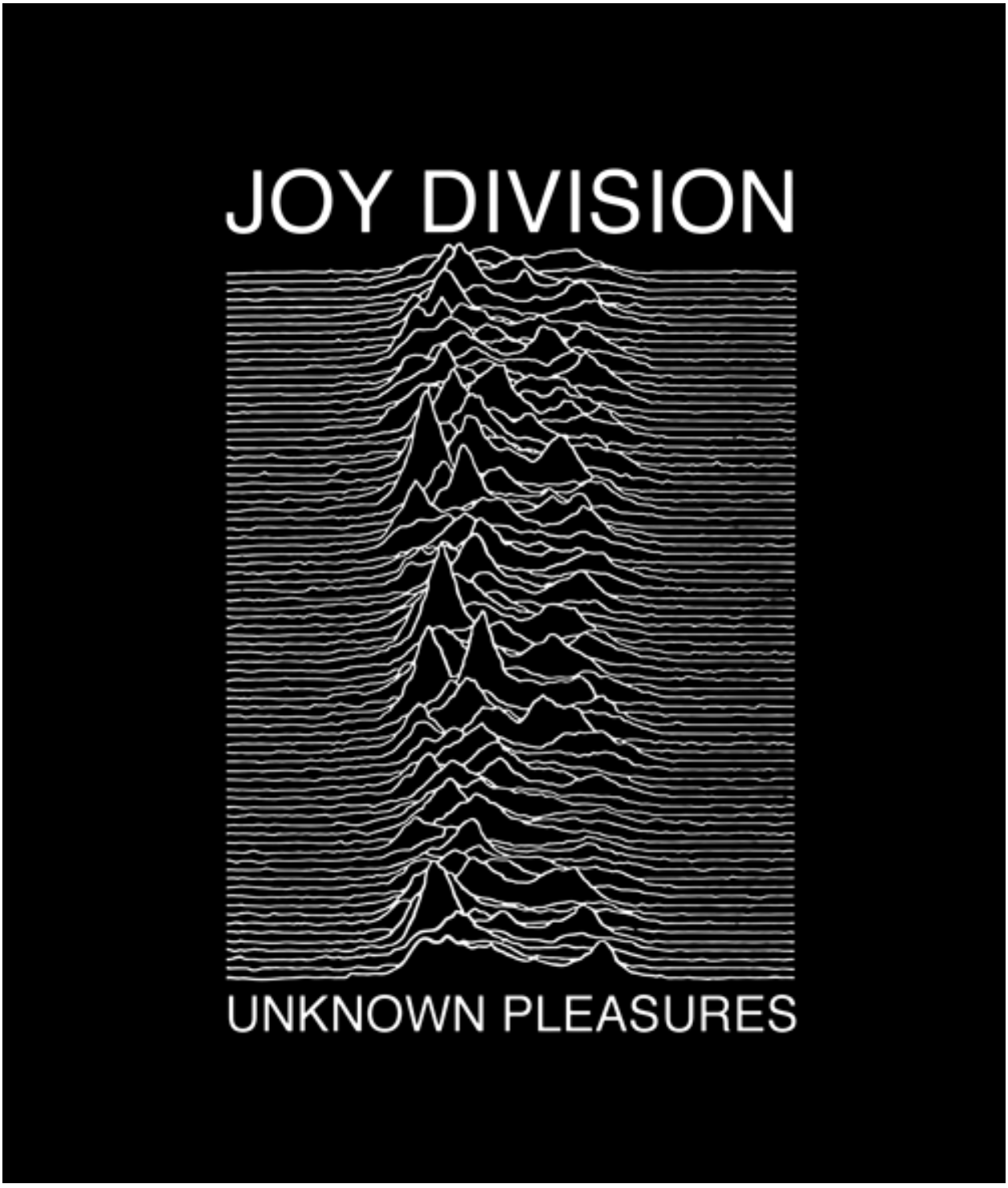
Or you could use transparencies:

```
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 nummerica 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)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following object is masked from 'package:MASS':
##
##     select
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
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. 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 50 51 50 51 51 50 51 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 ggplot 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`.

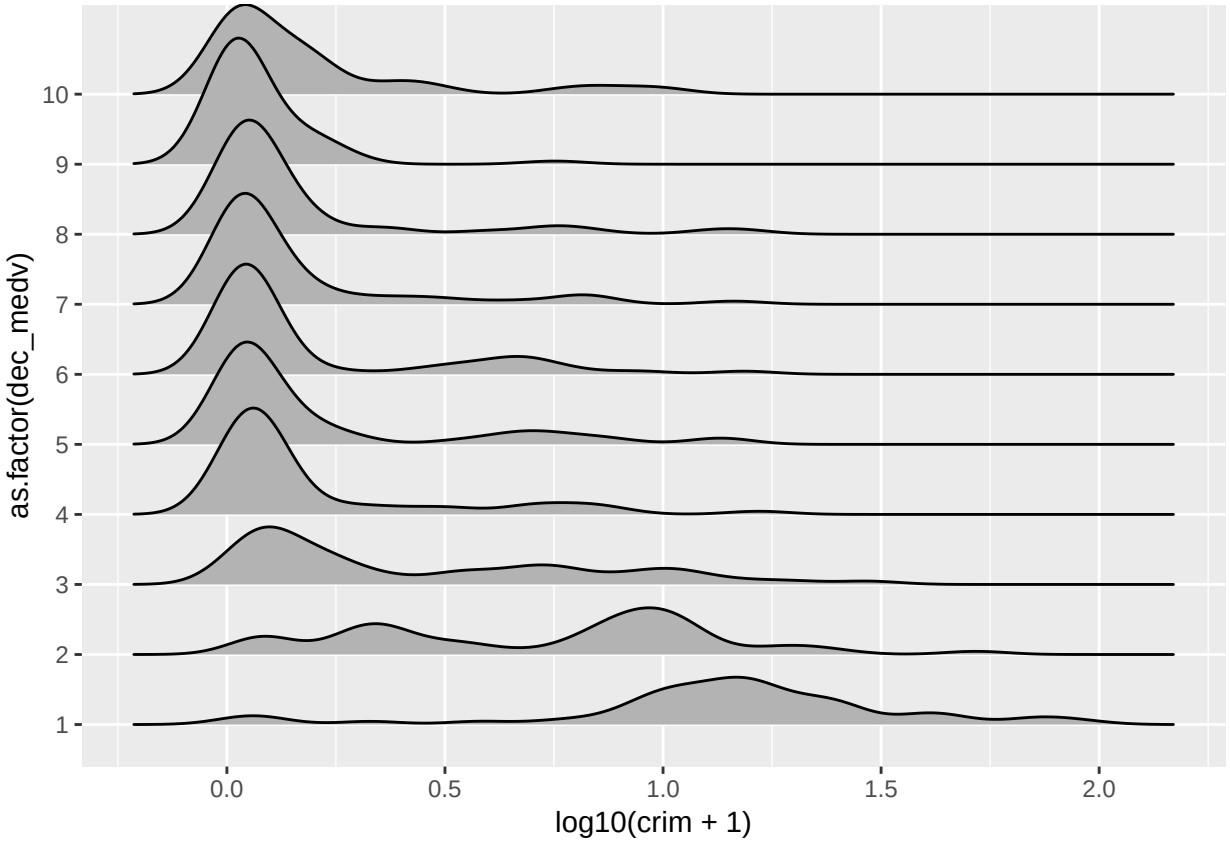
```
library(ggridges)
```

```
##
## Attaching package: 'ggridges'
```

```
## The following object is masked from 'package:ggplot2':
##
##     scale_discrete_manual
```

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

```
## Picking joint bandwidth of 0.072
```



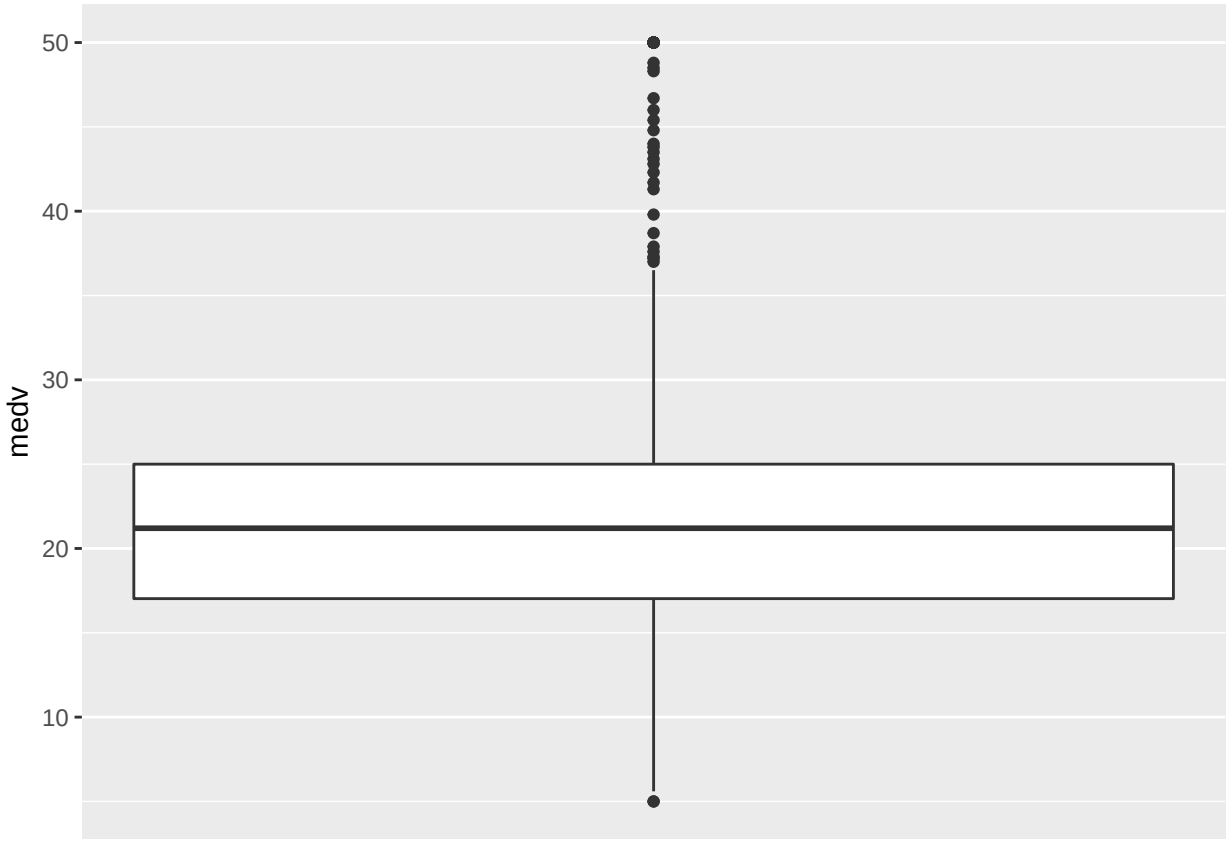
We can see that the distribution of crime is particularly different when we focus in the three deciles with the lowest level of income. For more details you can read the vignette for this package.

##Box plots

Box plots are an interesting way of presenting the 5 number summary 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.

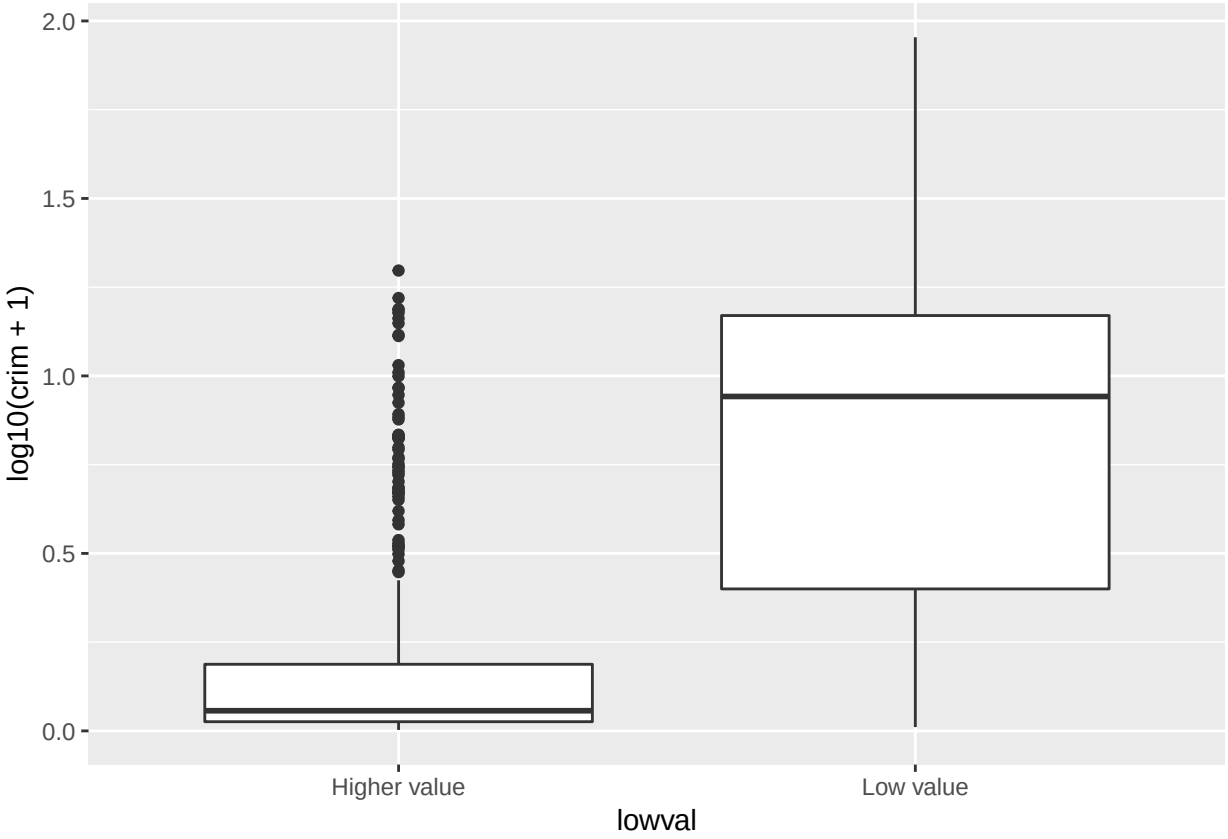
For this illustration, I am going to display the distribution of the median value of property in the various towns instead of crime.

```
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.

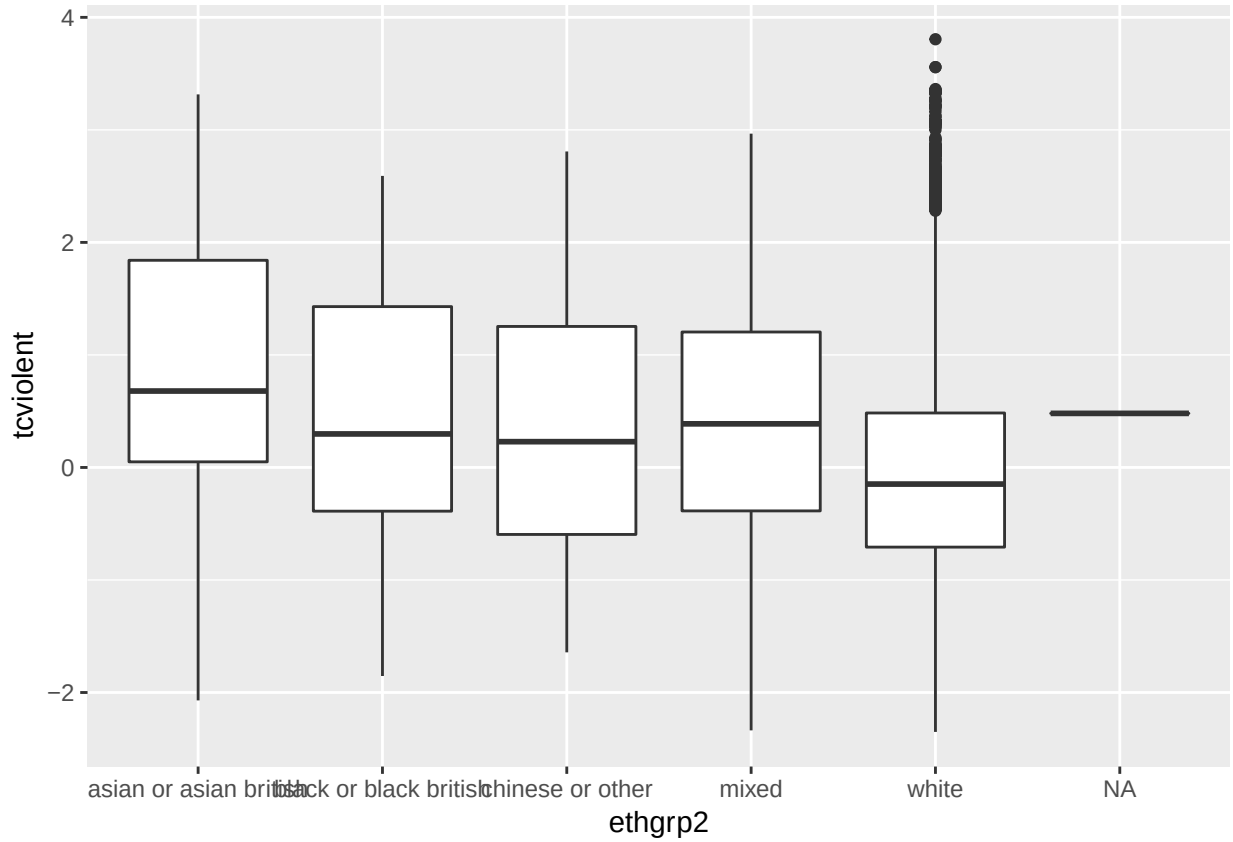
This can be even more helpful when you have various groups. Let's try an example using the `BCS0708` data frame. This is a dataset from the 207/08 British Crime Survey.You can use the code below to download it.

```
##R in Windows have some problems with https addresses, that's why we need to do this first:
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`).

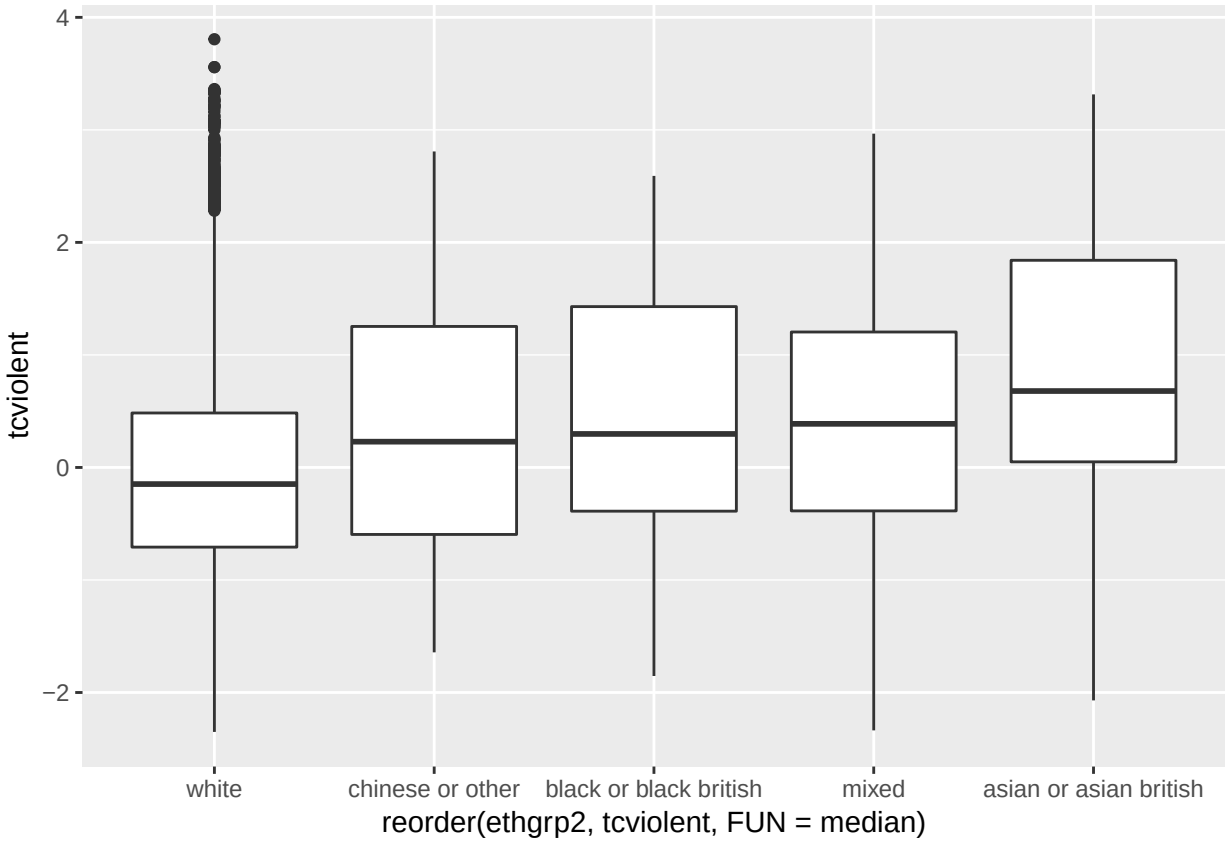
```
#A comparative boxplot of ethnicity and worry about violent crime
ggplot(BCS0708, aes(x = ethgrp2, y = tcviolent)) +
  geom_boxplot()
```

```
## Warning: Removed 3242 rows containing non-finite values (stat_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(subset(BCS0708, !is.na(ethgrp2) & !is.na(tcviolent)),
       aes(x = reorder(ethgrp2, tcviolent, FUN = median), y = tcviolent)) +
        geom_boxplot()
```



The `subset` function 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 os saying "the contrary of is NA for the specified variable". 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 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`).

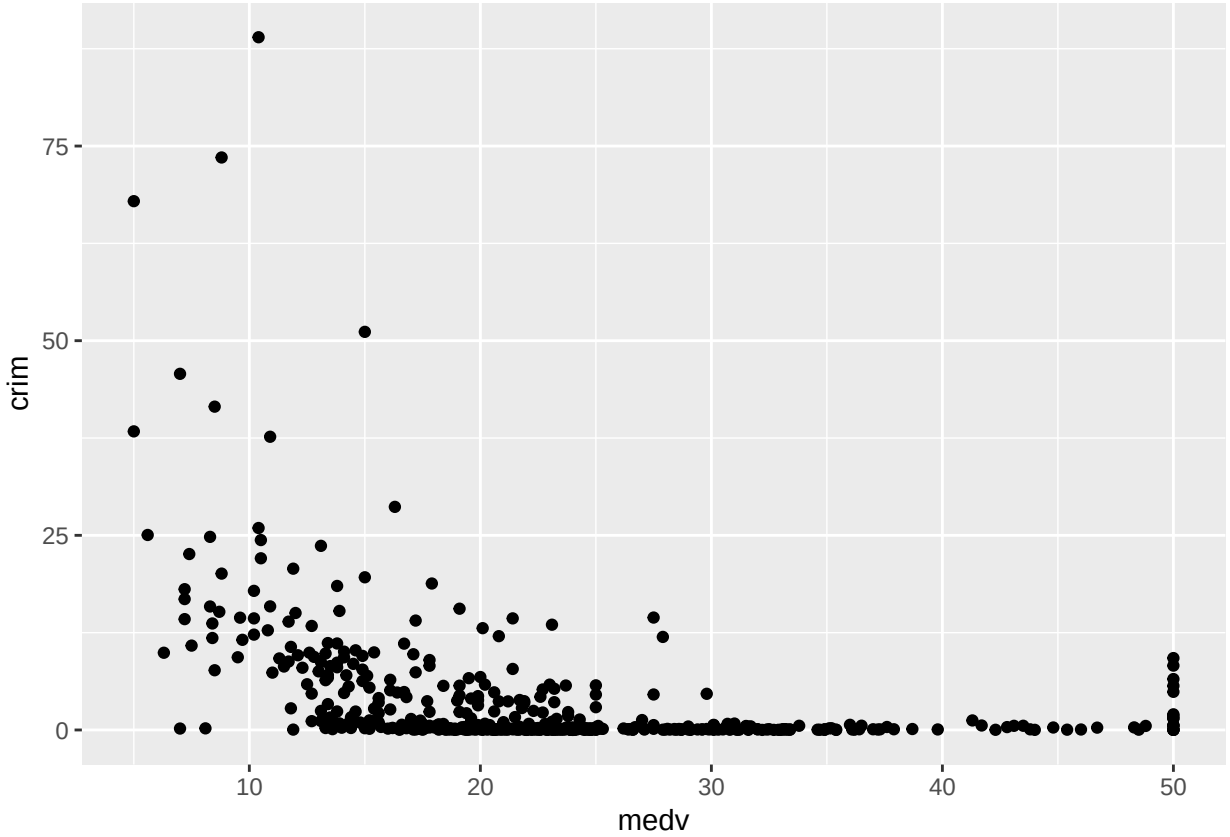
##Scatter plots with two variables

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.

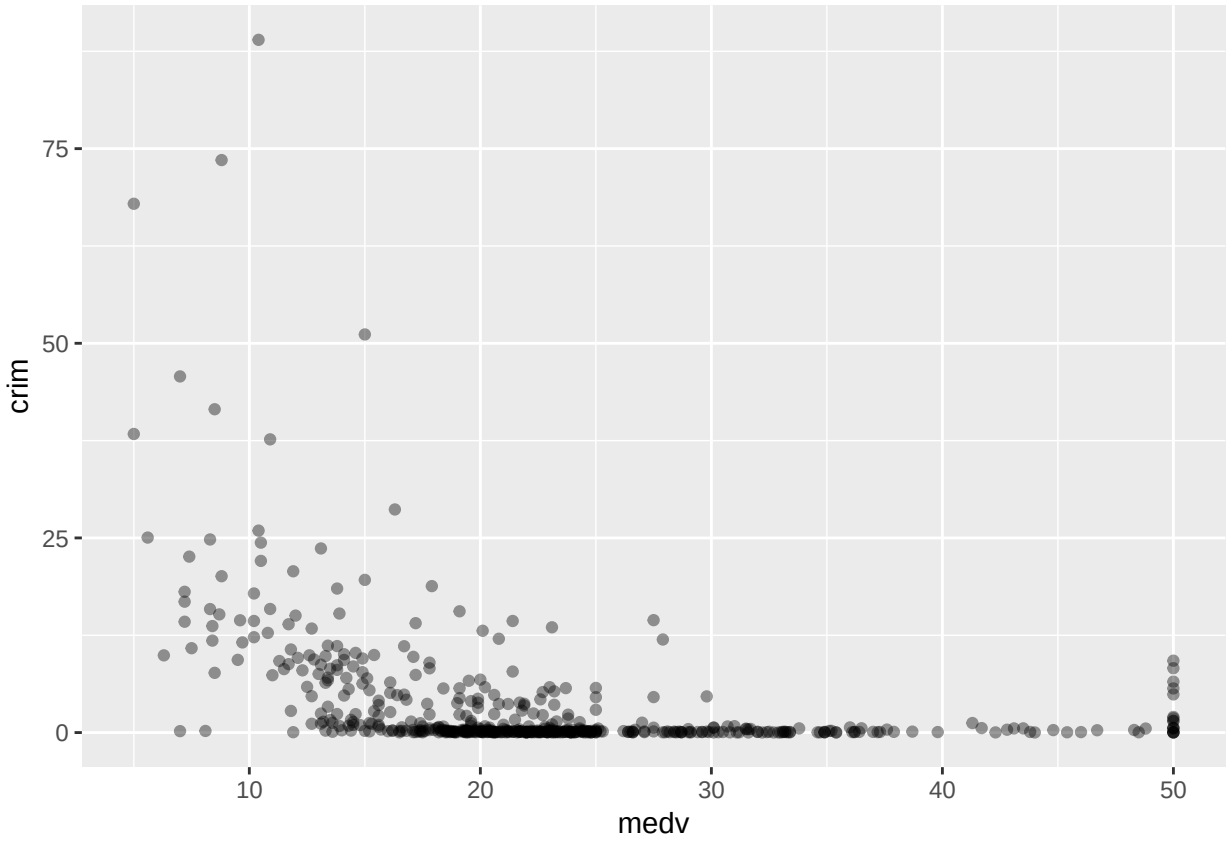
What do you look for in a scatterplot? You want to assess global and local patterns, as well as deviations. We can see clearly that at low levels of `medv` there is a higher probability in this data that the level of crime is higher. Once the median value of the property hits $30,000 the level of crime 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 decent level of crime. What's going on? To be honest I have no clue (although I have some hypothesis)! This is my first look at this dataset. But the pattern at the higher level of property value is indeed odd.

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. 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 mark. If you were analysing this for real, you would want to know what's driving this pattern (e.g., find out about the original data collection, the codebook, etc.): perhaps the maximum median value was capped at $50K 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. There's a variety of ways of dealing with overplotting. One possibility is to add some **transparency** to the points:

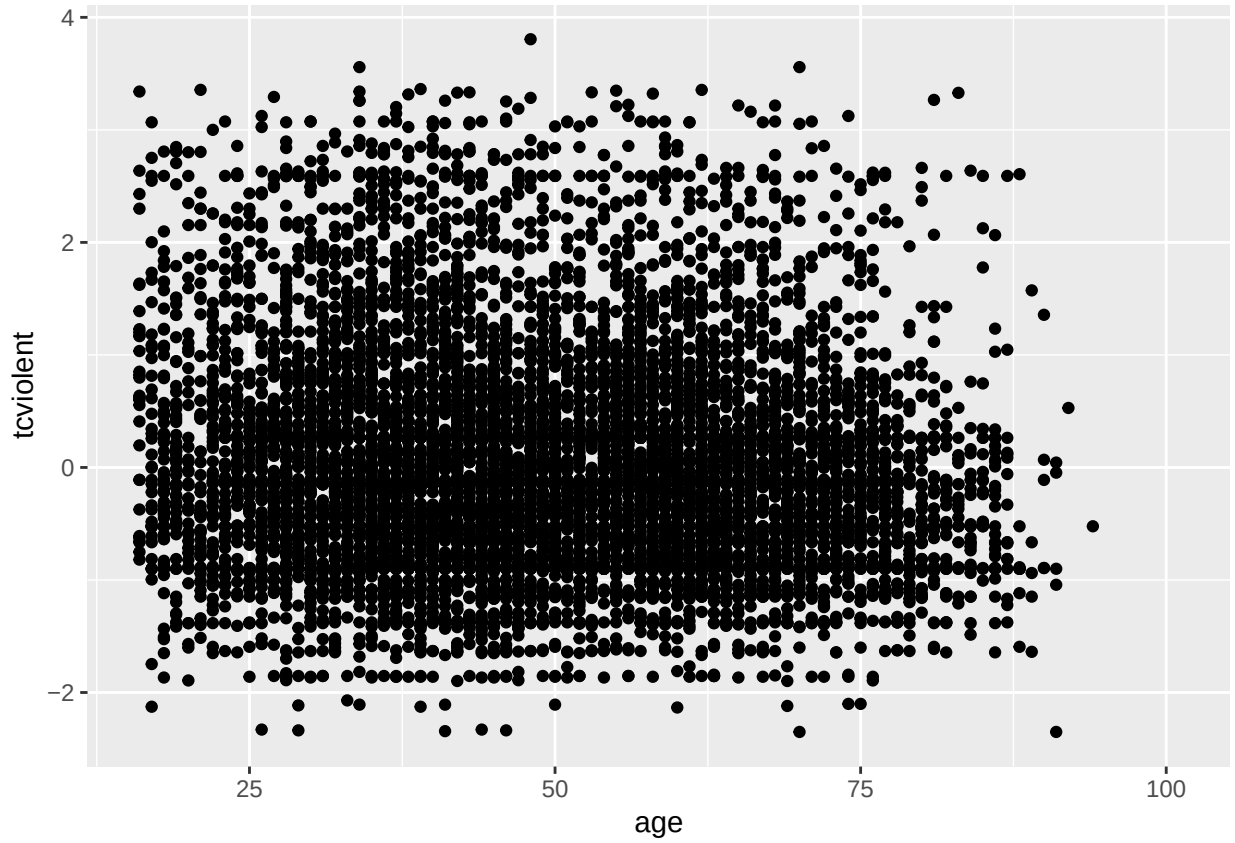
```
ggplot(Boston, aes(x = medv, y = crim)) +
  geom_point(alpha=.4) #you will have to test different values for alpha
```



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

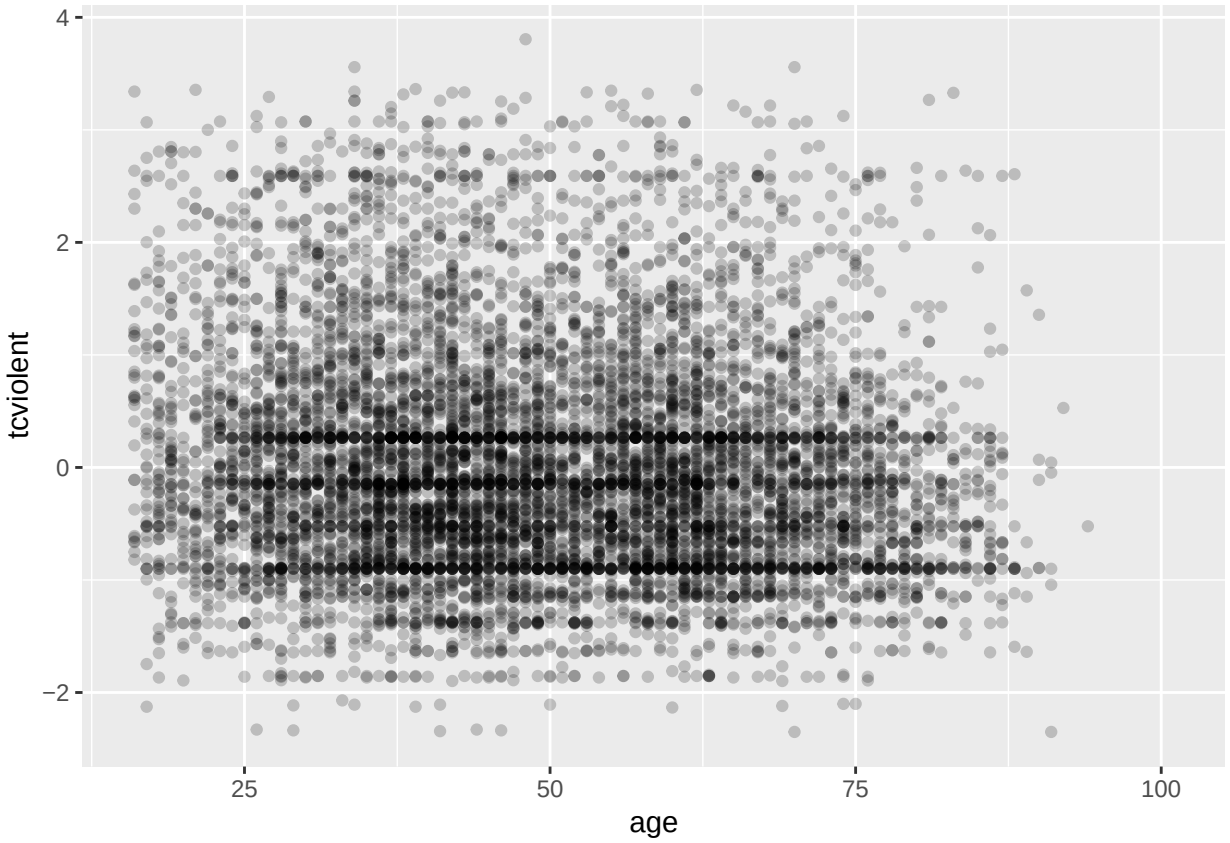
```
ggplot(BCS0708, aes(x = age, y = tcviolent)) +
  geom_point()
```

```
## Warning: Removed 3252 rows containing missing values (geom_point).
```




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

```
## Warning: Removed 3252 rows containing missing values (geom_point).
```



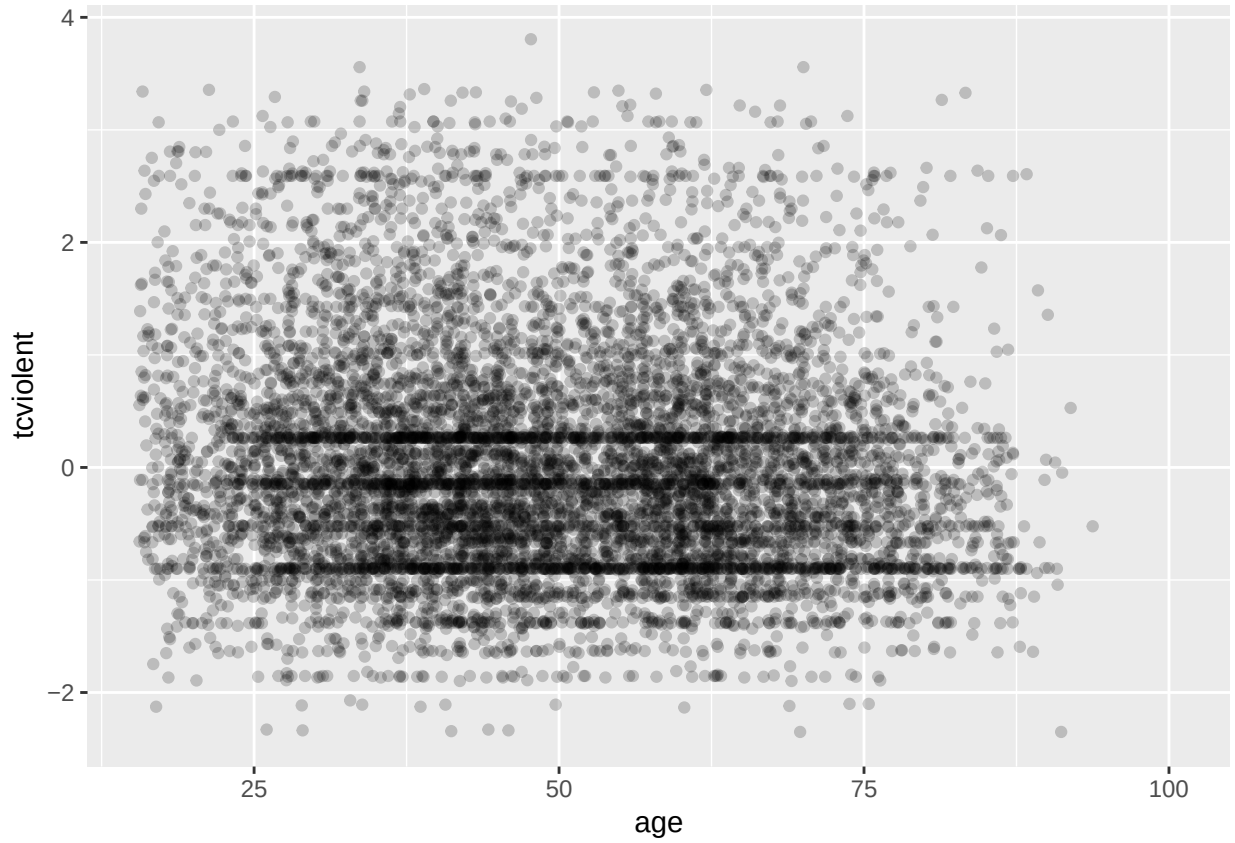
The second plot gives us a better idea of where the observations seem to concentrate in a way that we could not see with the first.

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() with geom_jitte
```

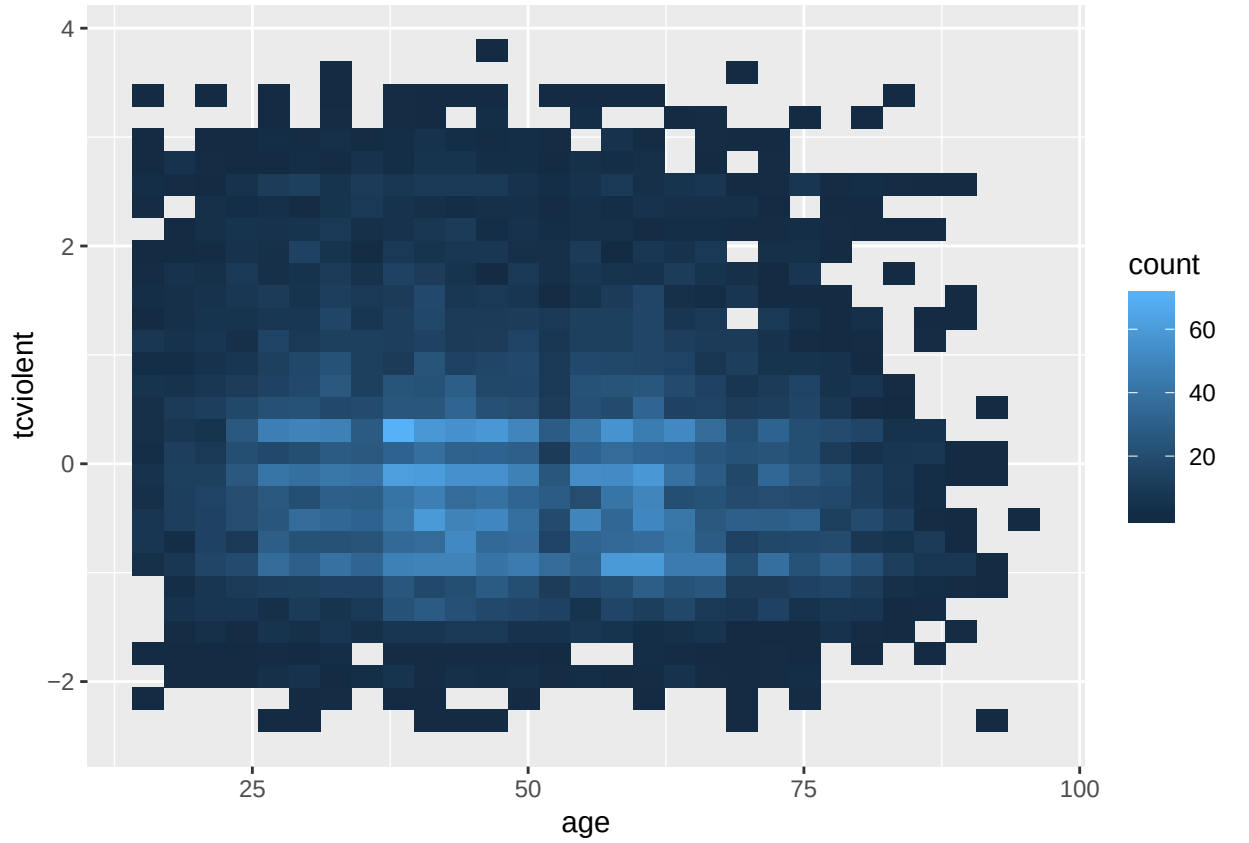
```
## Warning: Removed 3252 rows containing missing values (geom_point).
```



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.

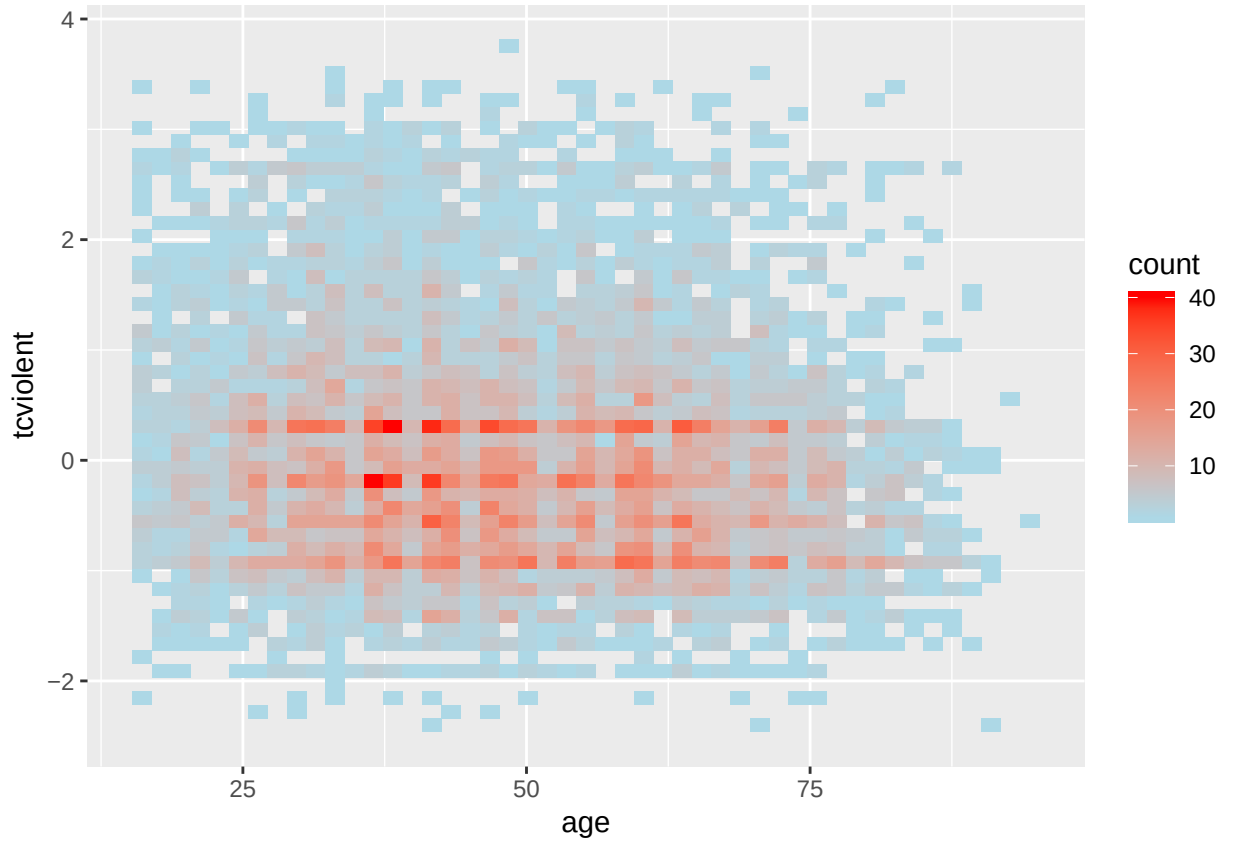
```
ggplot(BCS0708, aes(x = age, y = tcviolent)) +
  stat_bin2d()
```

```
## Warning: Removed 3252 rows containing non-finite values (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
```

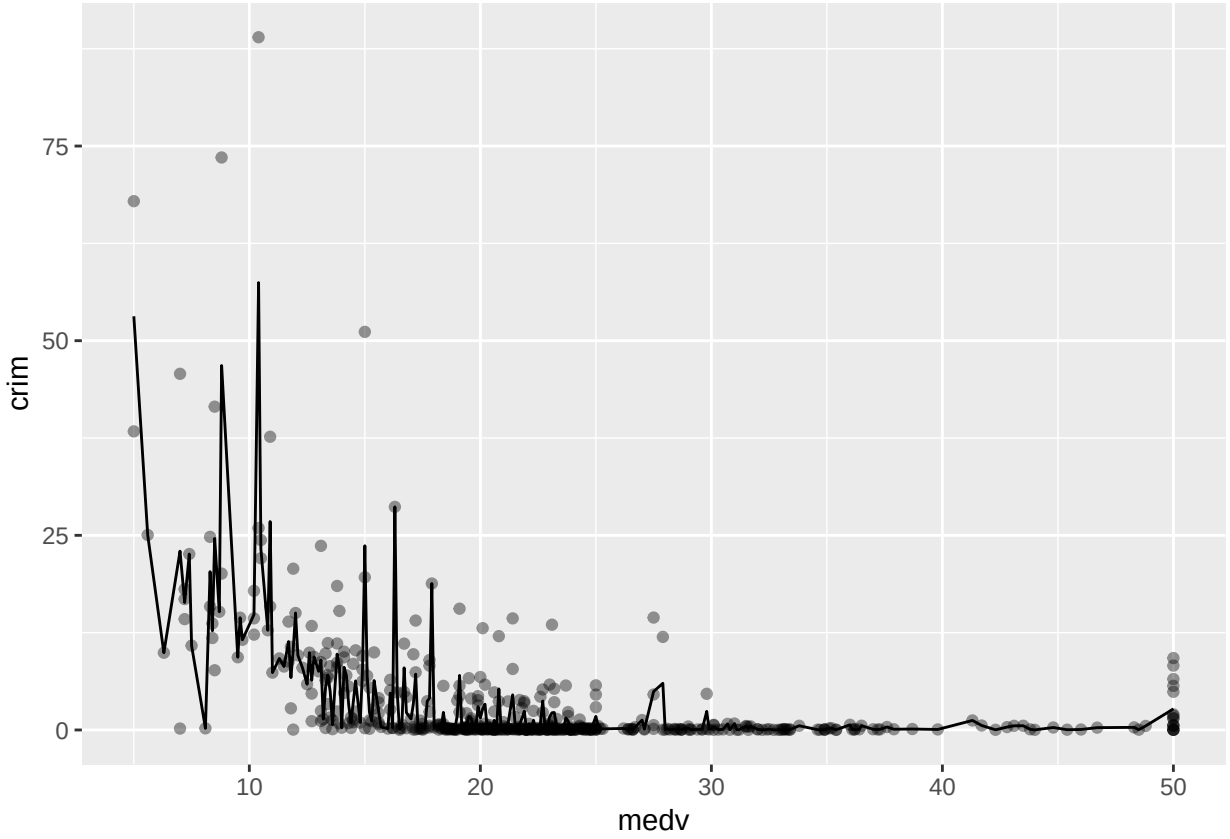
```
## Warning: Removed 3252 rows containing non-finite values (stat_bin2d).
```



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.

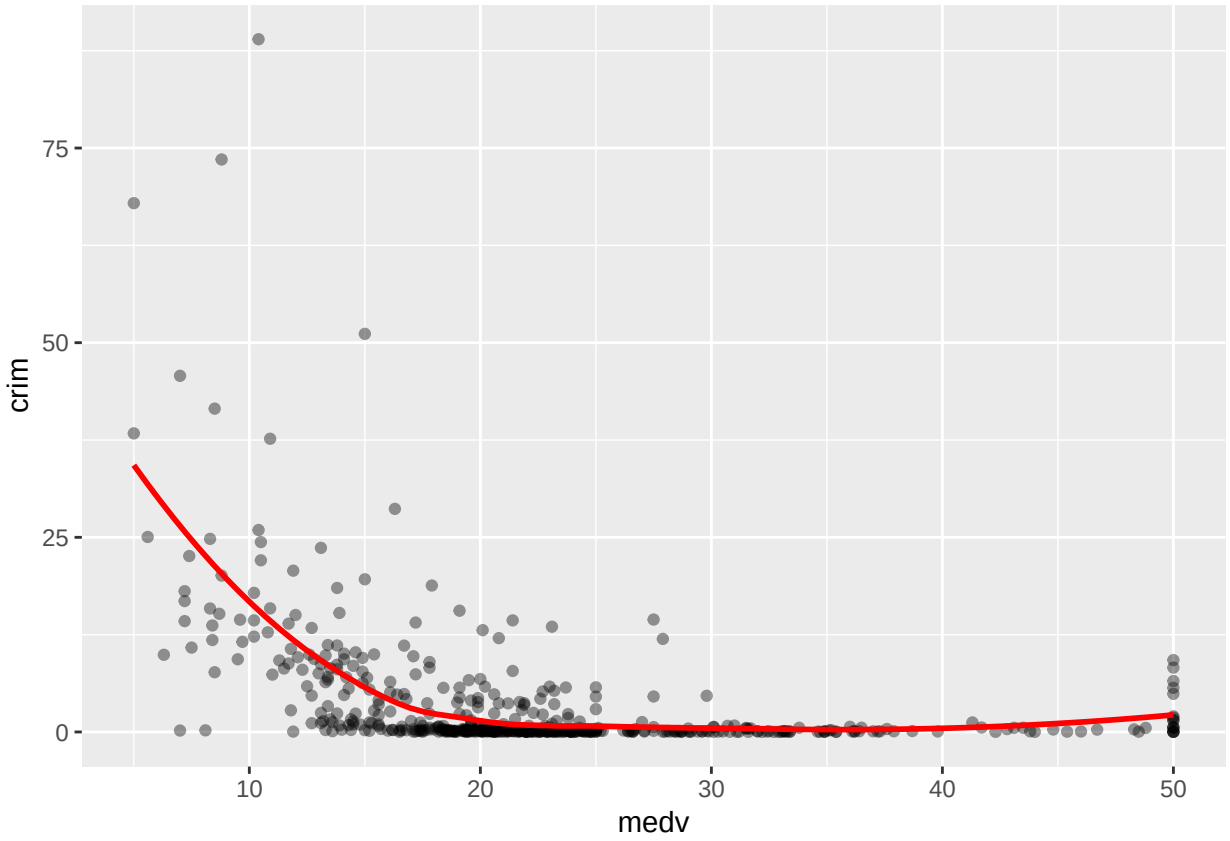
```
ggplot(Boston, aes(x = medv, y = crim)) +
  geom_point(alpha=.4) +
  geom_line(stat='summary', fun.y=mean)
```



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 the se argument d
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



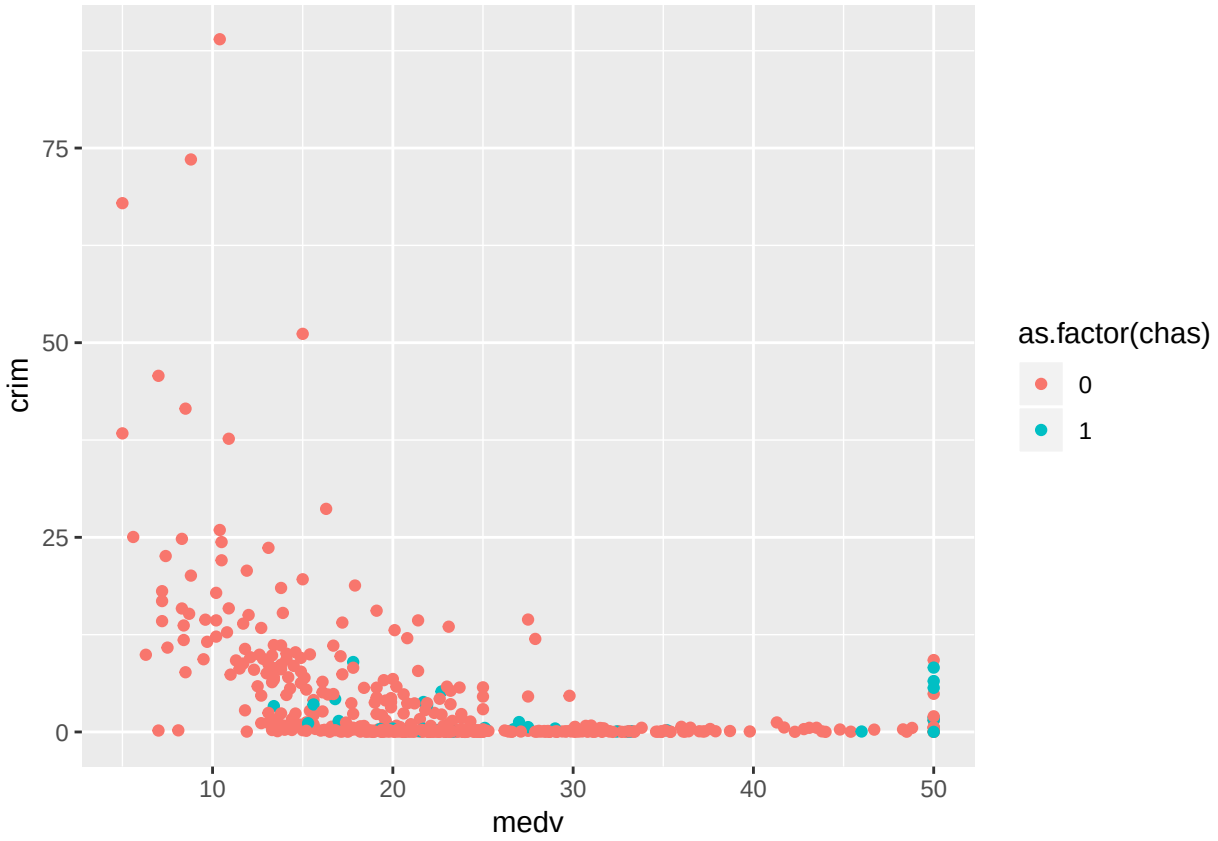
As you can see here you produce a smoother line than with the conditional means. The line, as the scatterplot, seems to be suggesting an overall curvilinear relationship that almost flattens out once property values hit $20k.

##Scatter plots 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 to tell "chas",
ggplot(Boston, aes(x = medv, y = crim, colour = as.factor(chas))) +
  geom_point()
```

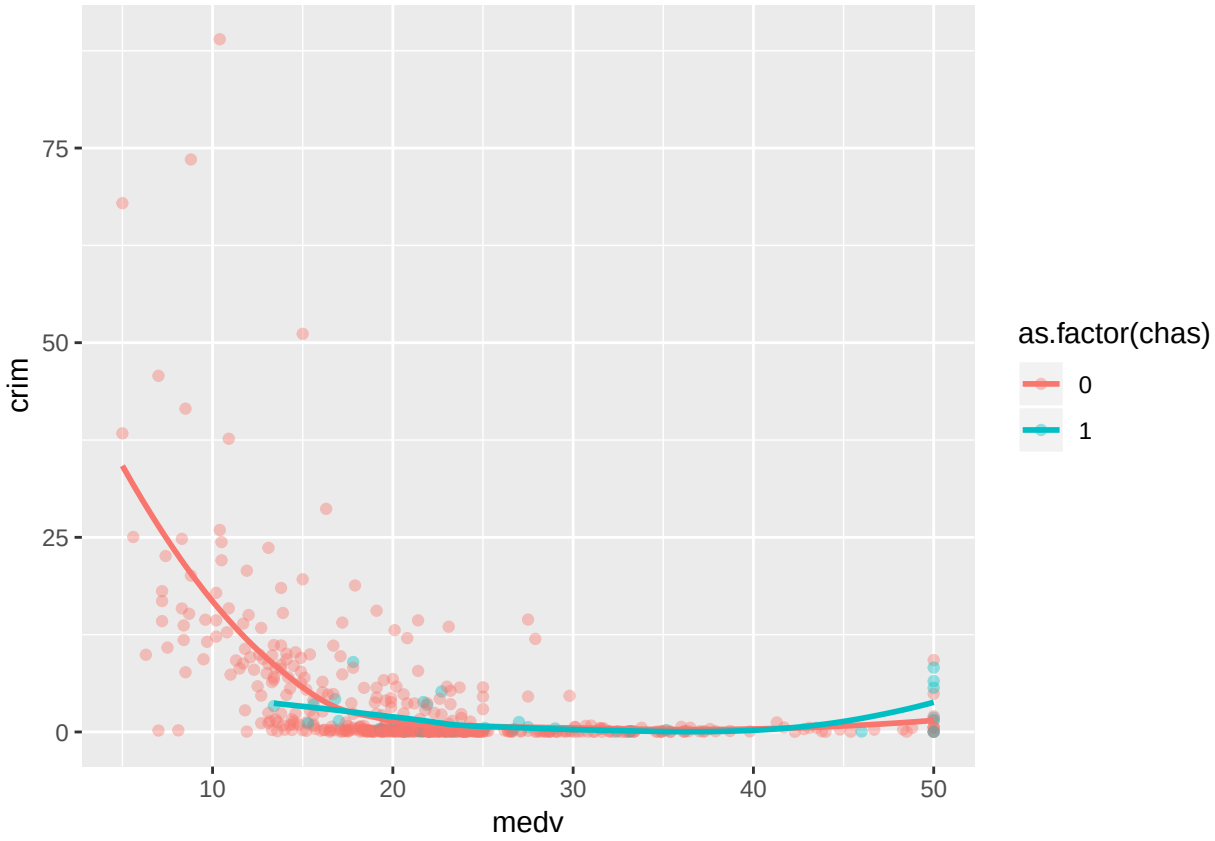


Curiously, we can see that there's quite a few of those expensive areas with high levels of crime that seem to be located by the river.

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
```

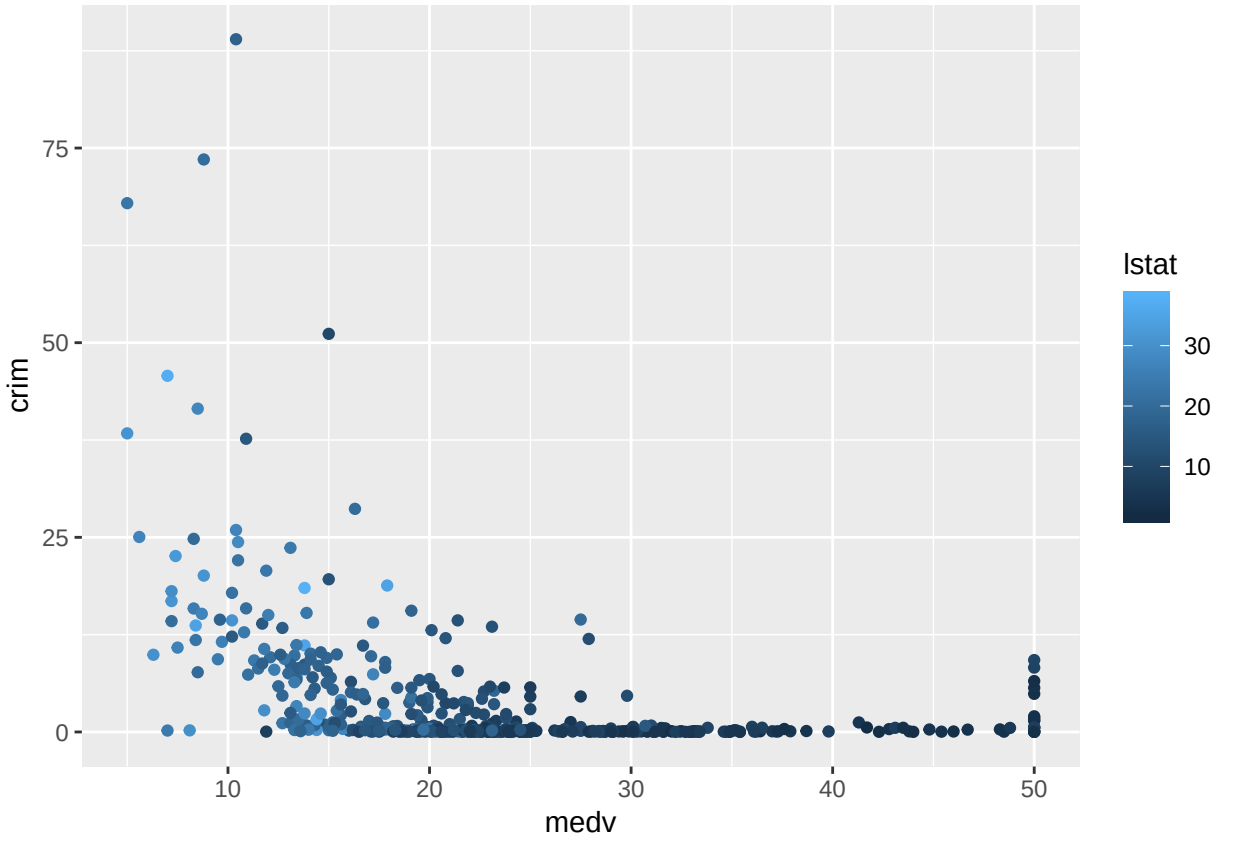
```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



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).

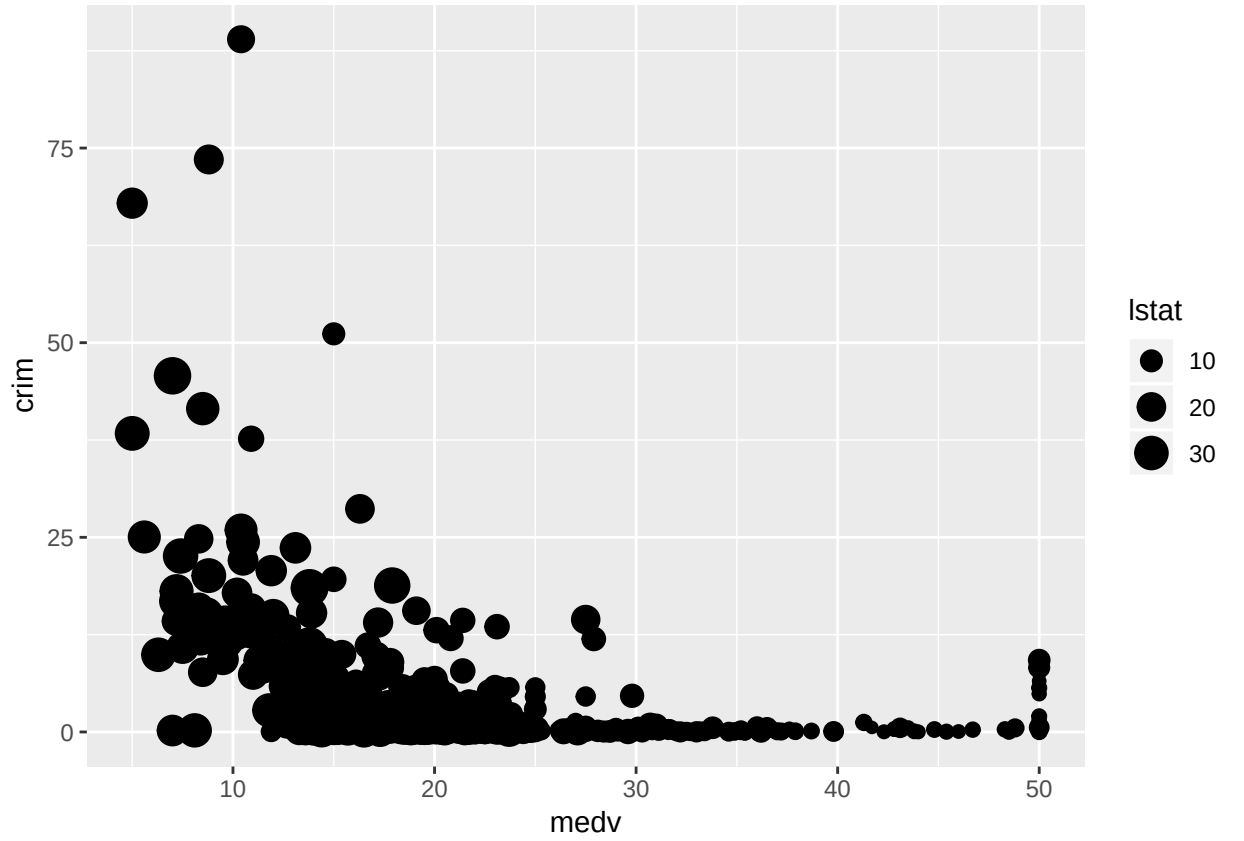
```
ggplot(Boston, aes(x = medv, y = crim, colour = lstat)) +
  geom_point()
```



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.

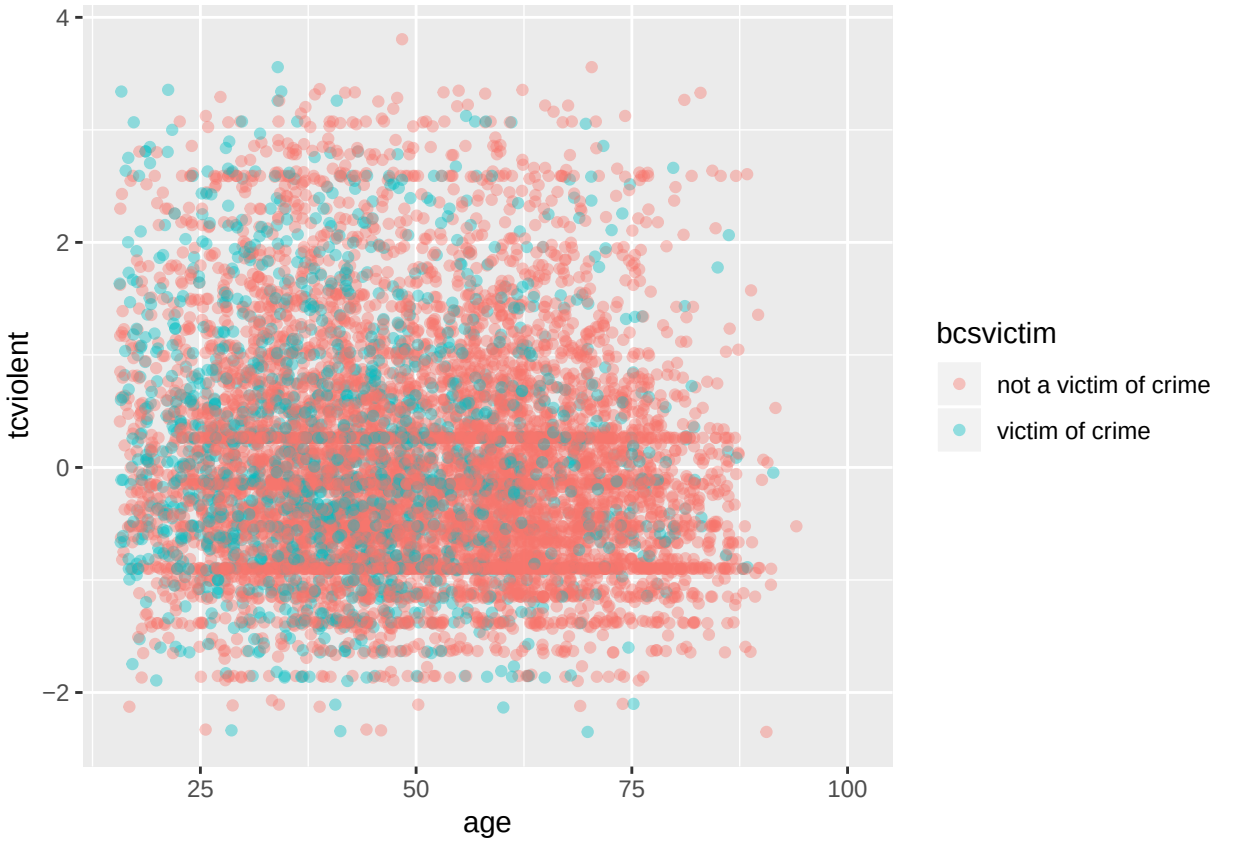
```
ggplot(Boston, aes(x = medv, y = crim, size = lstat)) +
  geom_point() #You may want to add alpha for some transparency here.
```



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")
```

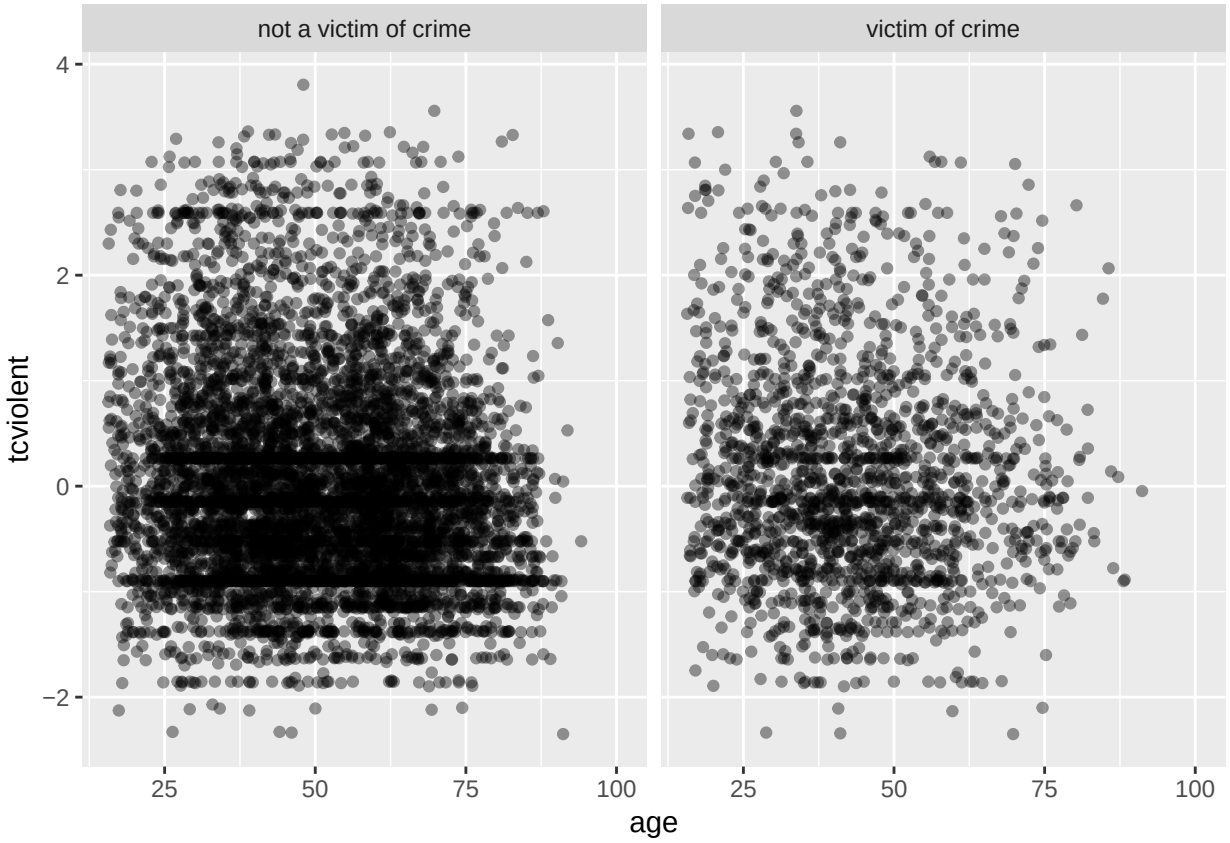
```
## Warning: Removed 3252 rows containing missing values (geom_point).
```



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)
```

```
## Warning: Removed 3252 rows containing missing values (geom_point).
```



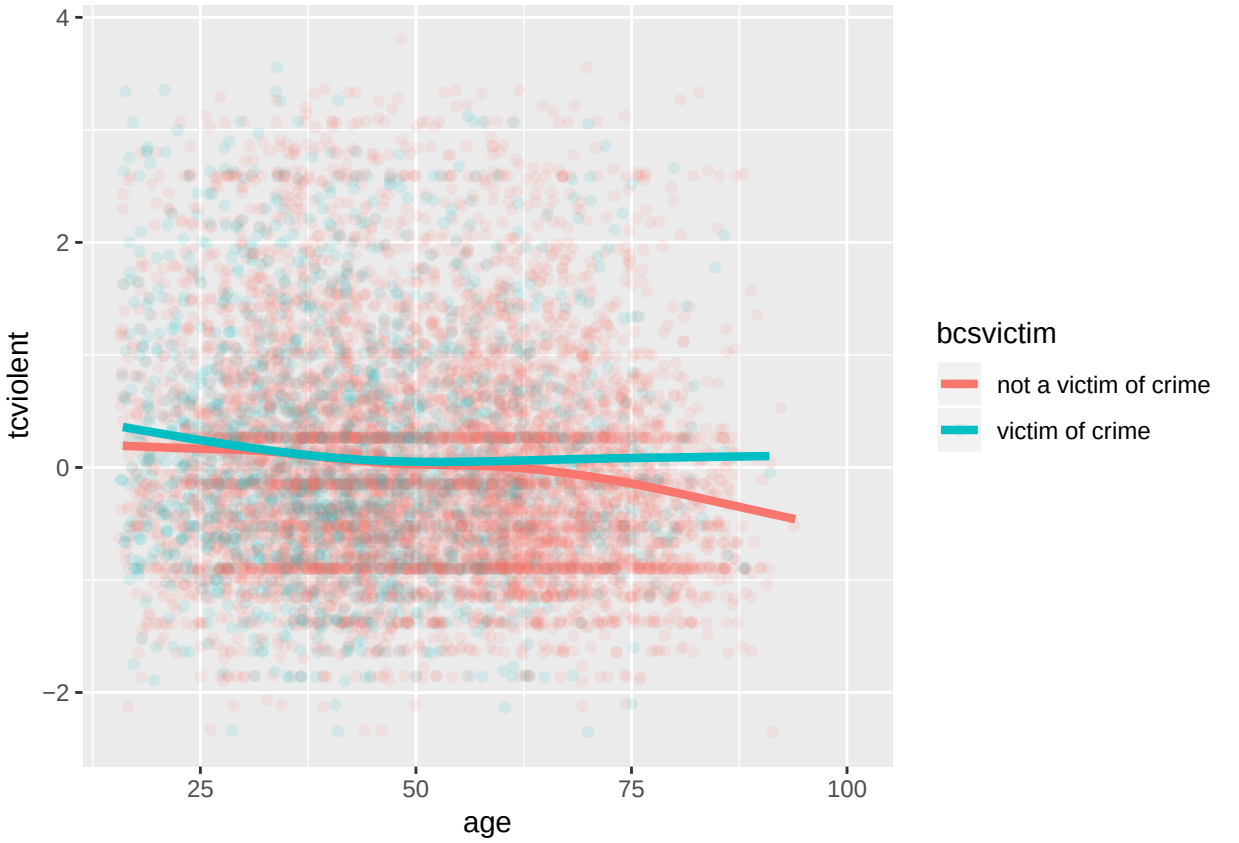
It is still hard to see anything, though perhaps you can notice the lower density the points in the bottom right corner in the facet displaying victims of crime. In a case like this may be helpful to draw a smooth line

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

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```

```
## Warning: Removed 3252 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 3252 rows containing missing values (geom_point).
```



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).

##Scatterplot matrix

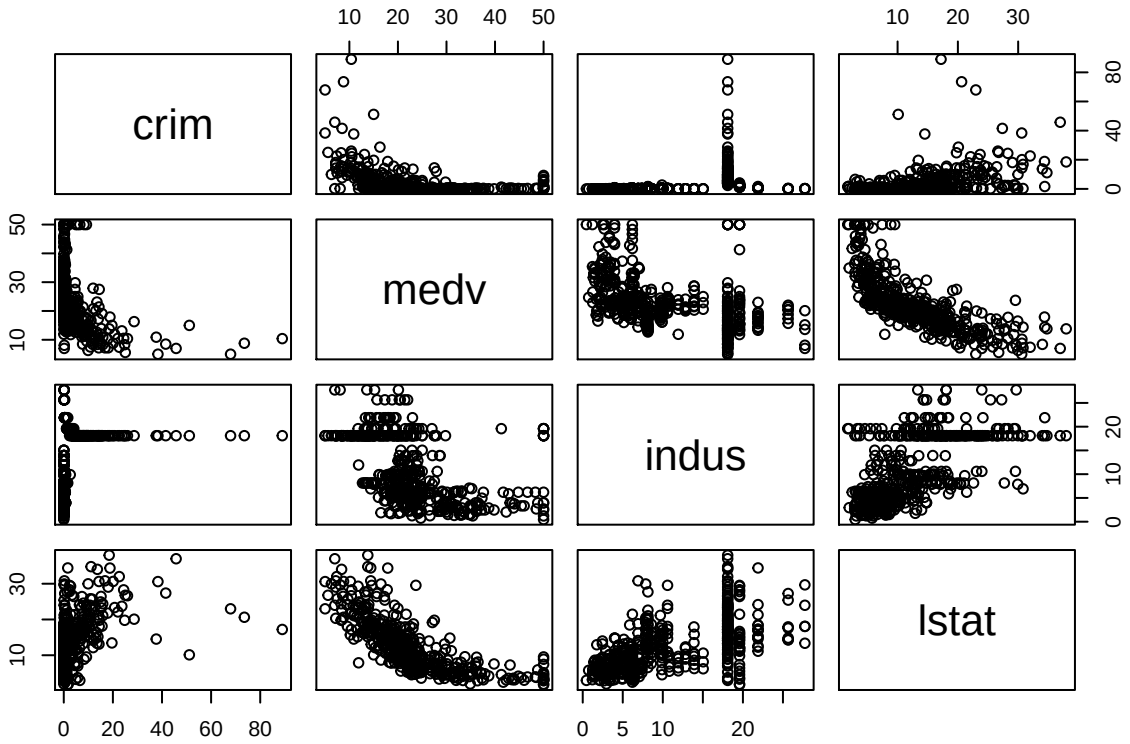
Sometimes you want to produce many scatterplots simultaneously to have a first peak at the relationship between the various variables in your data frame. The way to do this is by using a scatterplot matrix. Unfortunately, `ggplot2` doesn't do scatterplot matrix too well. For this it is better to use the graphics tools of base R.

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 calli
Boston_SM <- subset(Boston, select = c(crim, medv, indus, lstat))
```

Then we run the scatterplot matrix using the **pairs()** function:

```
pairs(Boston_SM)
```



We can modify this code to put additional information in the matrix. As you may have noticed the matrix is symmetrical, for every scatterplot in the upper left side there is a similar on the the bottom right side. We are going to modify this matrix so that the diagonal boxes not only display the name of the variable but also a histogram for the variables and to display the correlation coefficients in the upper left side (instead of printing twice the same scatterplot). We will also make the points smaller and will add a smooth line within each scatterplot.

The code for this is a bit complicated. We are going to introduce **customised functions** for the panels (I took them from Winston Chang book). First we are going to create these two customised functions. One to produce the correlations between each pair of variables and another for producing the histograms for each of the variables.

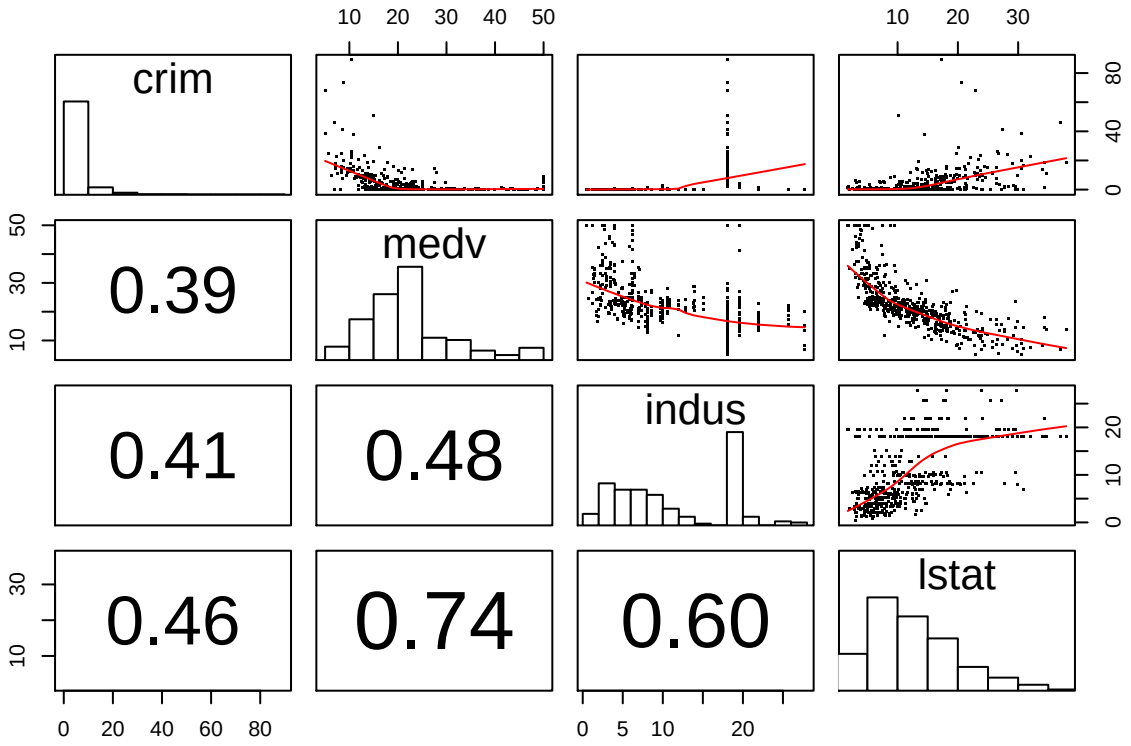
After that we will use the `pairs()` function inserting the inputs that result form using these customised functions. You are likely not to understand the code for the first part. Particularly because apart from doing computations is controlling the aspect of the results. Don't worry, I'm not expecting to understand this code. Just cut and paste. But I think it is important you understand that apart from using the functions created by others and included in packages, once you become proficient in R you can start producing your own functions to expand the range of what you can do with R.

```
panel.cor <- function(x, y, digits=2, prefix="", cex.cor, ...){
  usr <- par("usr")
  on.exit(par(usr))
  par(usr = c(0, 1, 0, 1))
  r <- abs(cor(x, y, use="complete.obs"))
```

```
  txt <- format(c(r, 0.123456789), digits=digits)[1]
  txt <- paste(prefix, txt, sep="")
  if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
  text(0.5, 0.5, txt, cex = cex.cor * (1+r)/2)
}
panel.hist <-function(x, ...){
  usr <- par("usr")
  on.exit(par(usr))
  par(usr = c(usr[1:2], 0, 1.5))
  h <- hist(x, plot = FALSE)
  breaks <- h$breaks
  nB <- length(breaks)
  y <- h$counts
  y <- y/max(y)
  rect(breaks[-nB], 0, breaks[-1], y, col="white", ...)
}
```

We can now run the `pairs()` function using our customised functions as inputs.

```
pairs(Boston_SM, pch=".", #Produces smaller points to make it easier to see
      upper.panel=panel.smooth, #Adds for smoothed lines to be added to the lower panel scatterplots
      lower.panel=panel.cor, #Ask for correlation coeffients in the upper panel using our customised fu
      diag.panel=panel.hist) #Ask for histograms in the diagonal using our customised function
```



We will explain what correlation coefficients are and when they are appropriate later this semester. For

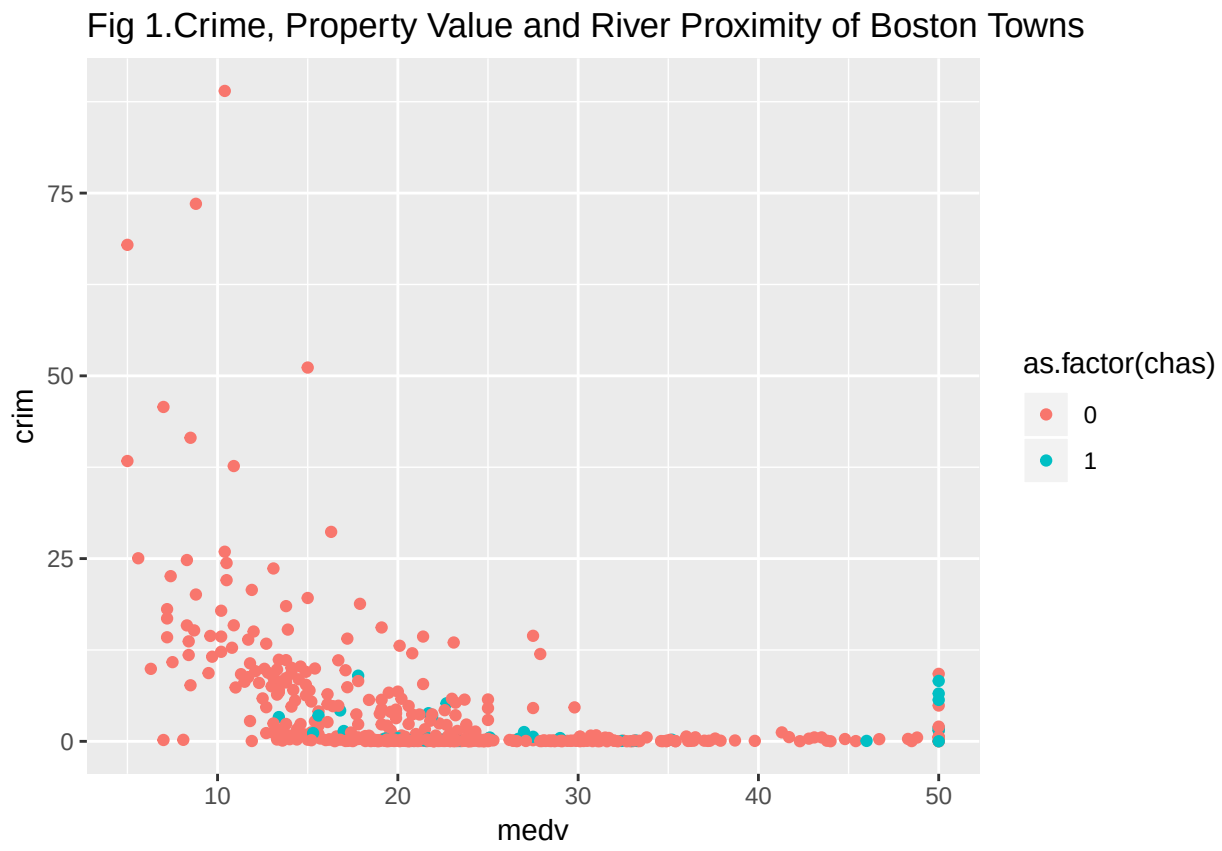
now just focus in trying to understand what the plots show. What do you think may be going on with the distribution of "indus" (proportion of non-retail business acres per town)?

As you can see 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. 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".

##Titles and legends in ggplot2

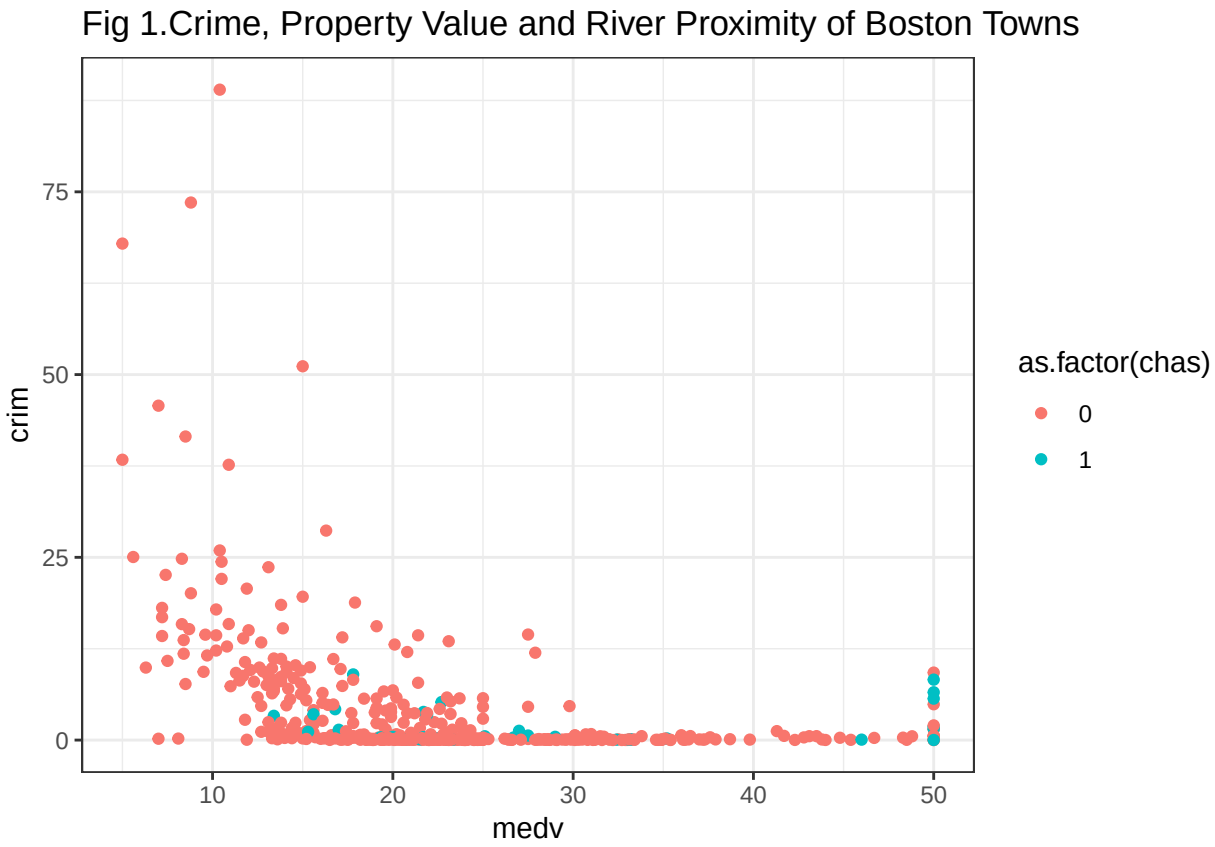
We have introduced a number of various graphical tools, but what if you want to customise the way the produce graphic looks like? Here I am just going to give you some code for how to modify the titles and legends you use. For adding a title for a `ggplot` graph you use `ggtitle()`.

```
#Notice how here we are using an additional function to ask R to treat the variable chas, which is nume
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")
```



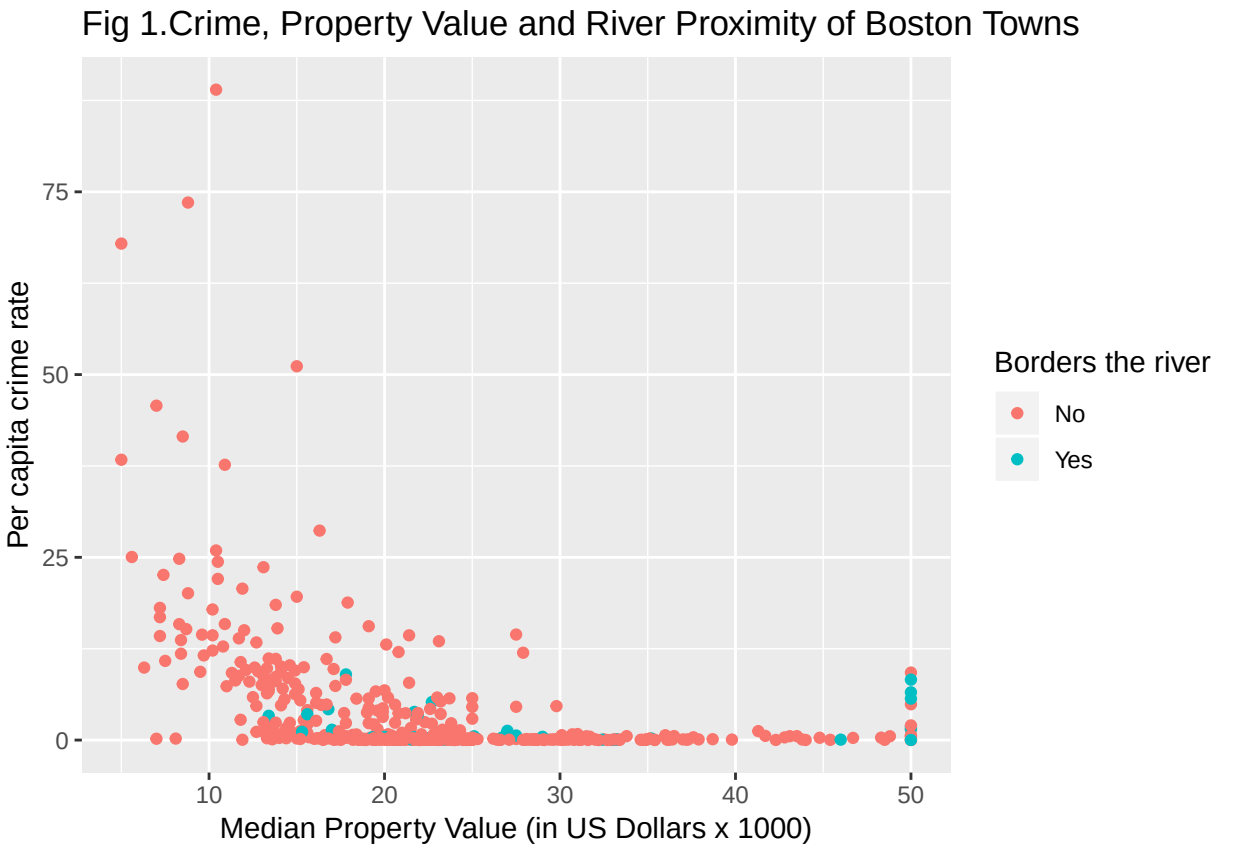
If you don't like the default background theme for `ggplot` you can use a theme as discussed at the start, for example with creating a black and white background by adding `theme_bw()` as a layer:

```
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()
```



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"))
```



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)
```

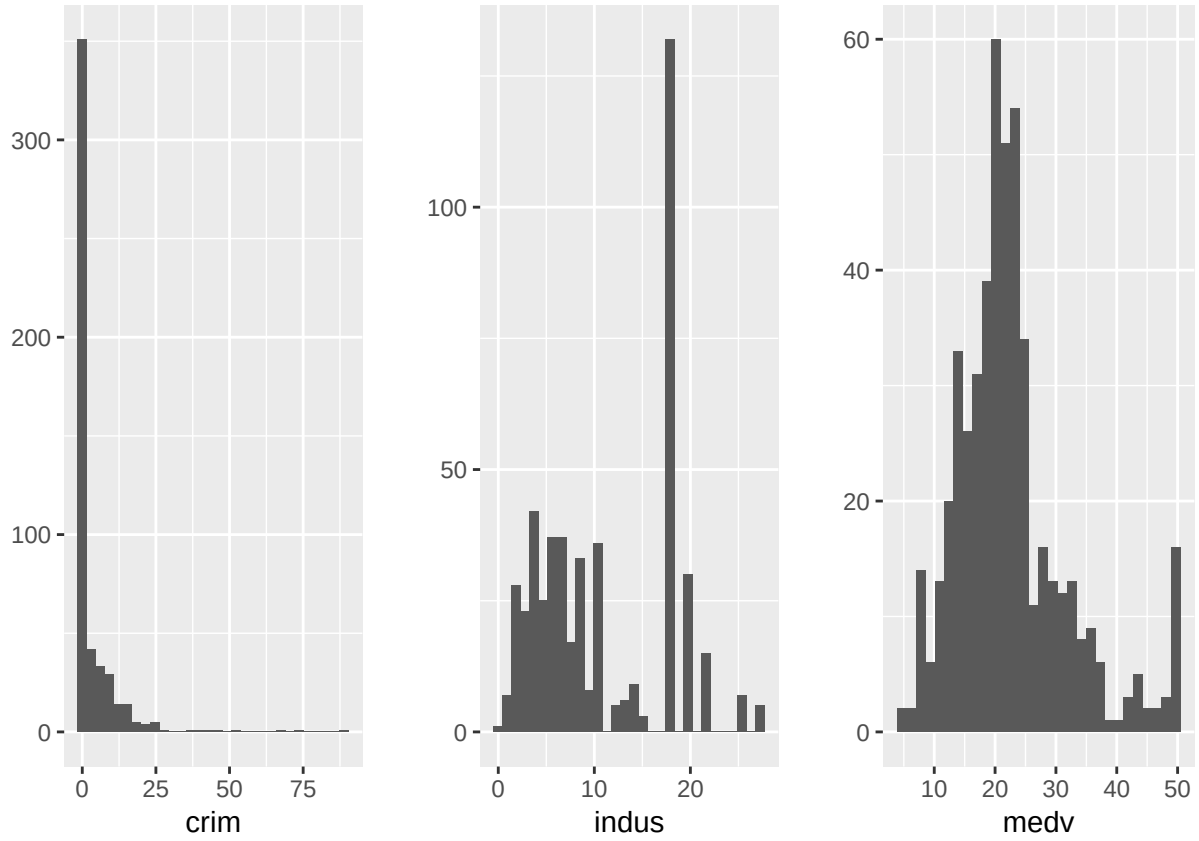
```
##
## Attaching package: 'gridExtra'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
#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 them one in top o
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

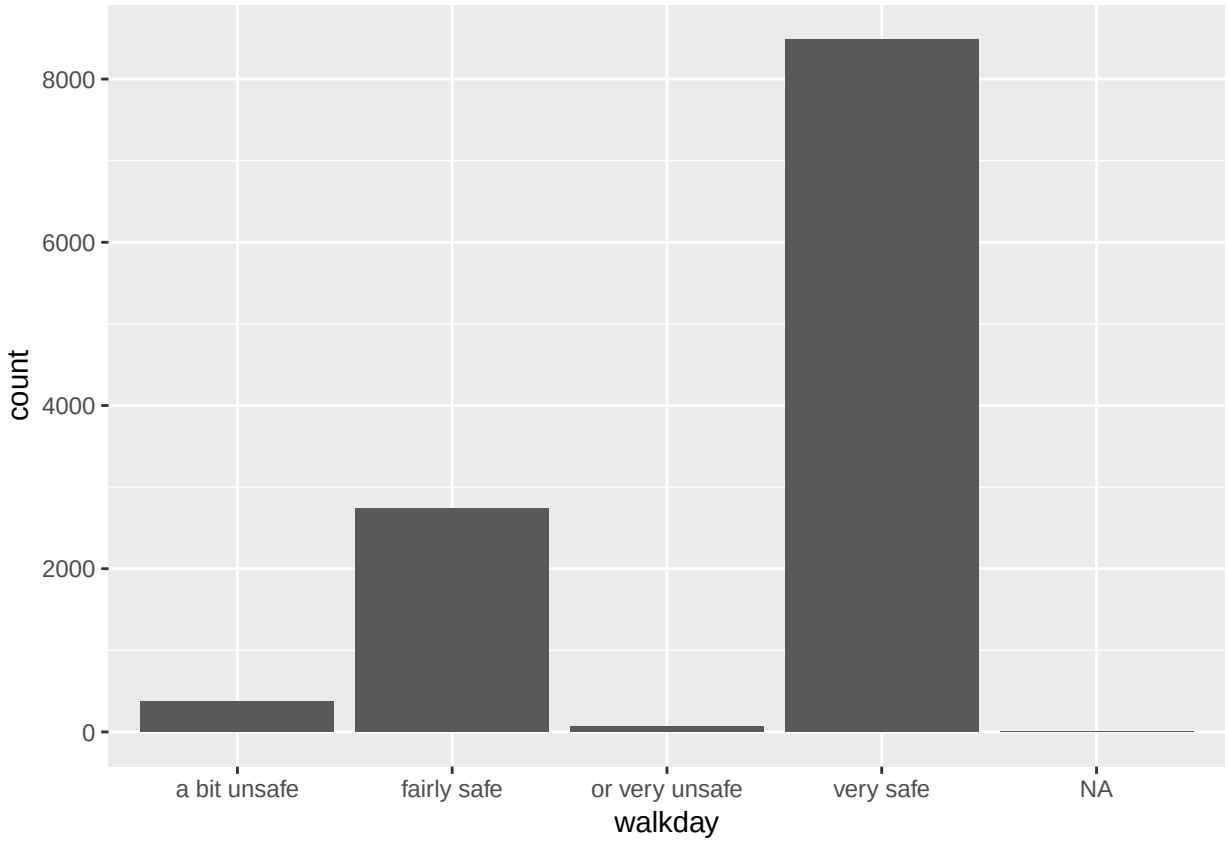
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



##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()
```

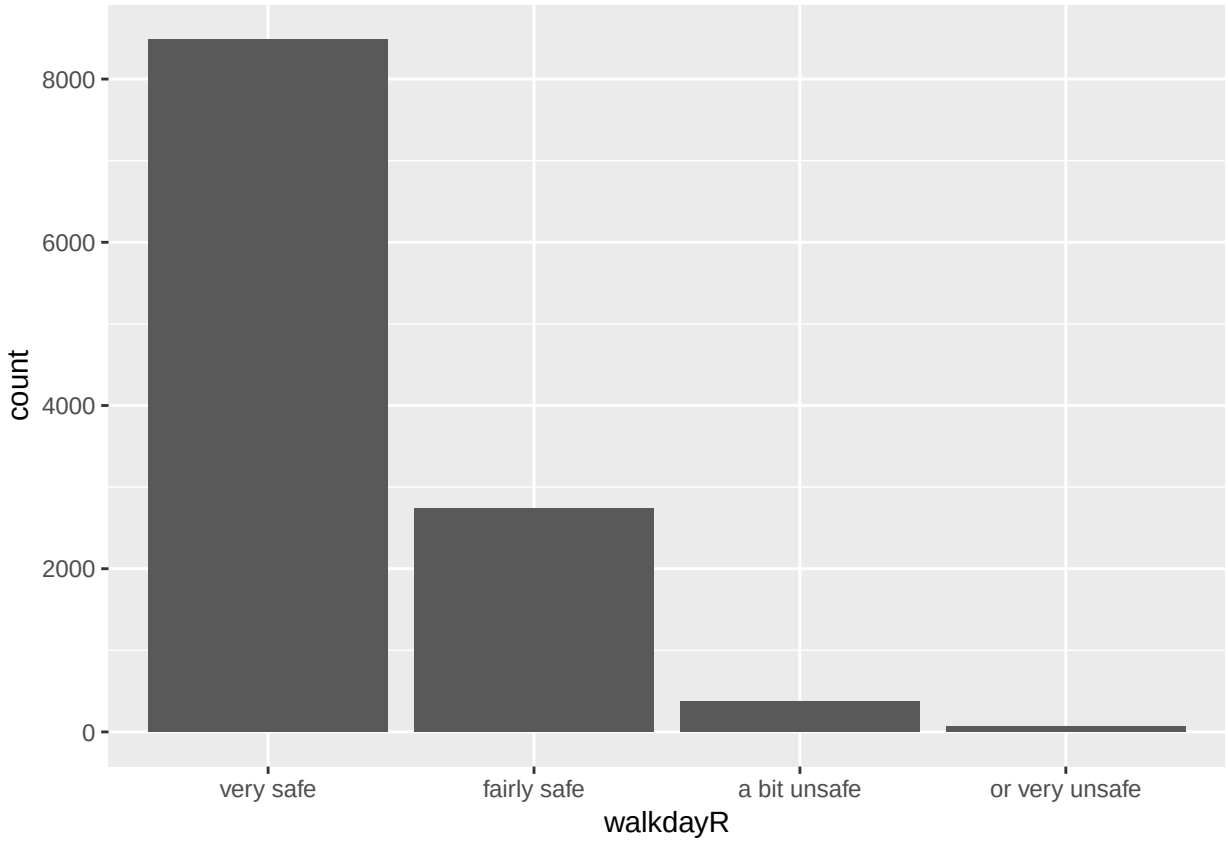


Unfortunately, the levels in this factor are ordered by alphabetical order, which is confusing. We can modify this by reordering the factors levels first -click here for more details. You could do this within the ggplot function (just for the visualisation), but in real life you would want to sort out 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))
```

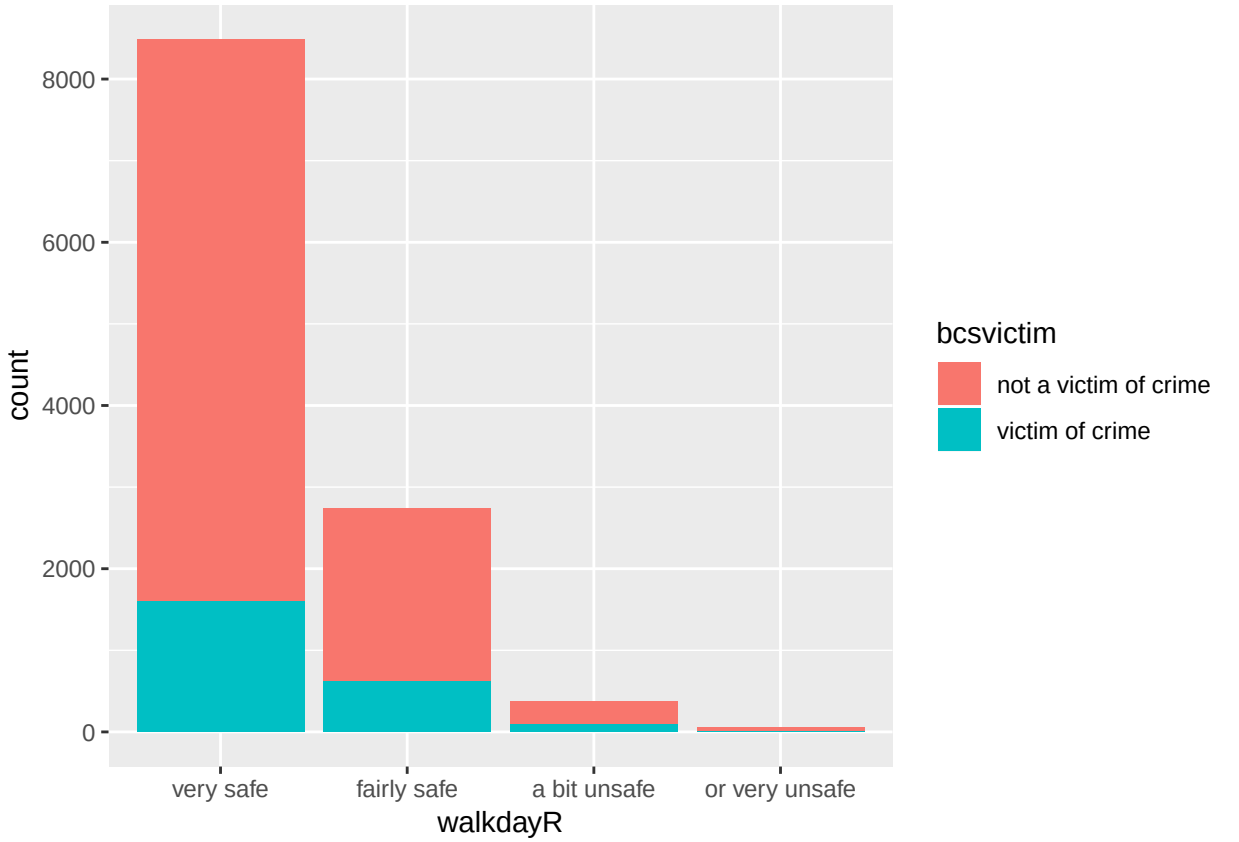
```
## [1] "a bit unsafe"   "fairly safe"    "or very unsafe" "very safe"
```

```
#Reordering the factor levels from very safe to very unsafe (rather than by alphabet). Notice that I am
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()
```



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

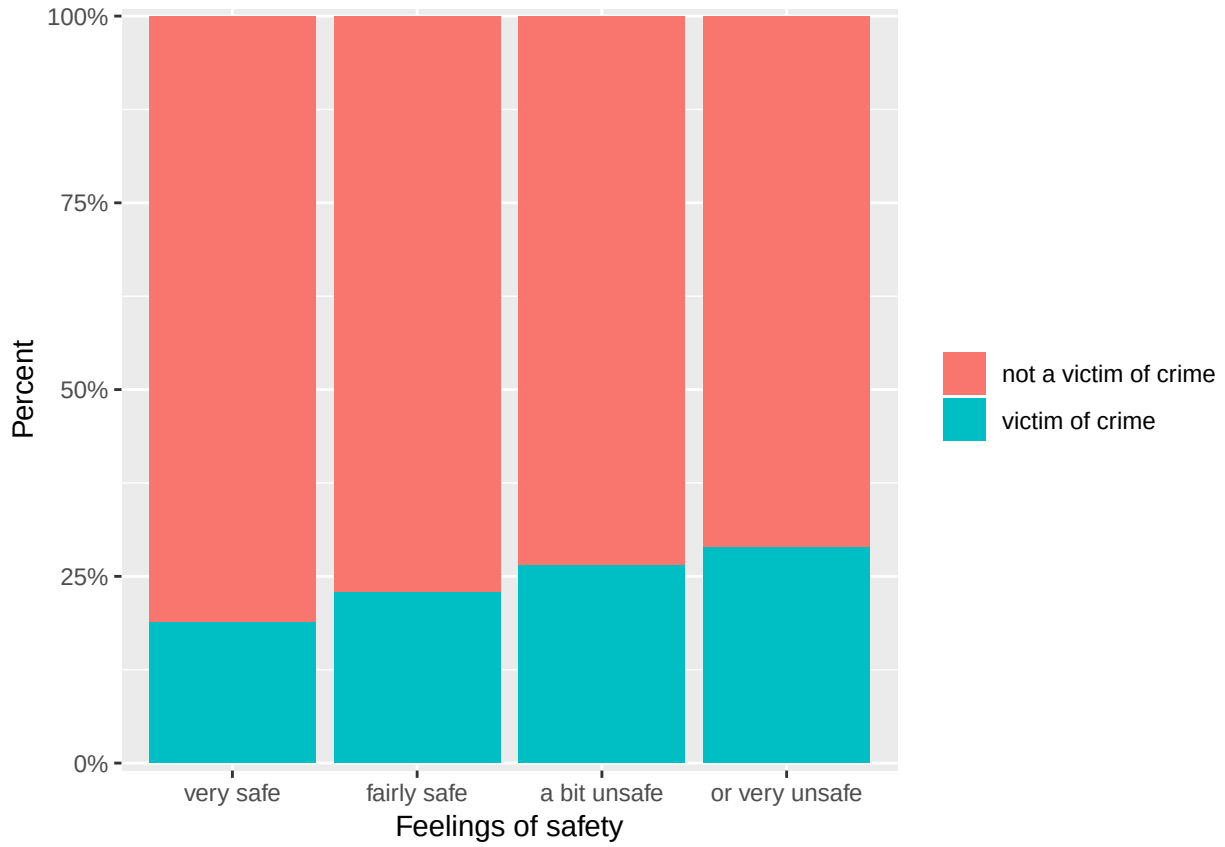
```
ggplot(subset(BCS0708, !is.na(walkdayR)), aes(x=walkdayR, fill=bcsvictim)) +
  geom_bar()
```



These sort of stacked bar charts are not terribly helpful if you are interested in understanding the relationship between these two variables. Instead what you want is a **proportional stacked bar chart**, that gives you the proportion of your "explanatory variable" (here victimisation in the past 12 months) within each of the levels of your "response variable" (here feelings of safety).

First we need to scale the data to 100% within each stack and then plot. The code here is also more complex. Again, I'm not expecting you to fully understand all of it. In your early days of using R, you will find yourself cutting and pasting chunks of code that you will only fully understand with practice.

```
#First we create a subset with the relevant data
.df1 <- data.frame(x = BCS0708$walkdayR, z = BCS0708$bcsvictim)
#Then we compute the proportions within the stacks
.df1 <- as.data.frame(with(.df1, prop.table(table(x, z), margin = NULL)))
#Finally we plot
stbcf <- ggplot(.df1, aes(x = x, y = Freq, fill = z)) +
  geom_bar(position = "fill", stat = "identity") +
  scale_y_continuous(expand = c(0.01, 0), labels = scales::percent_format()) + #Adapts the scale of the
  xlab("Feelings of safety") + #Another way of changing the label for the x axis
  ylab("Percent") + #Change the label for the Y axis
  guides(fill = guide_legend(title = NULL)) #This removes the title of the legent (bcsvictim), since th
stbcf #autoprint the plot, notice how earlier rather than printing directly I stored the plot in the ob
```




```
rm(.df1) #to remove the data frame we created
```

Here we can see that the those that express less feelings of safety are more likely to have experienced a victimisation in the past 12 months.

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 specification wit
stbcf + coord_flip()
```