Human Geography through Merseyside - Quantitative Block: Seeing the world through numbers

Zi Ye and Ron Mahabir

2026-02-04

Table of contents

Welcome 4
Contact . . . 4

Overview 5
Aim and Learning Objectives . . . 5
Module Structure . . . 5
Software and Data . . . 6

Assessment 8

1 Lab: Getting Started in RStudio - Knowing Merseyside 9
1.1 **Overview** . . . 9
1.2 **Getting set up with RStudio** . . . 9
1.2.1 **Install R and RStudio (if necessary)** . . . 9
1.2.2 File management . . . 10
1.2.3 **Open RStudio** . . . 10
1.2.4 Ways of working . . . 11
1.2.5 Your first R code . . . 12
1.3 Knowing Merseyside . . . 26
1.3.1 Merseyside districts . . . 26
1.3.2 Merseyside neighbourhoods . . . 42
1.4 Summary . . . 52
1.4.1 References . . . 52
1.5 Formative Tasks . . . 52

2 Lab: Exploratory Data Analysis - UK Election 56
2.1 **Overview** . . . 56
2.2 Clear the decks . . . 56
2.3 Open libraries . . . 57
2.4 Parliamentary Constituency Data . . . 57
2.4.1 Load the dataset . . . 57
2.4.2 Familiar with the dataset and variable types . . . 58
2.4.3 Exploratory Data Analysis (EDA) . . . 59
2.5 Make your own map for the election result . . . 77
2.5.1 Read in Parliamentary Constituency Boundaries . . . 77
2.5.2 Inspect the spatial dataset . . . 78

2.5.3 Link boundaries to pc_data 79
2.5.4 Map the election result 79
2.6 Formative tasks 81

Welcome

This is the website for "Human Geography through Merseyside - Quantitative Block: Seeing the world through numbers" (module **ENVS162**) at the University of Liverpool. This block of the module is designed and delivered by Dr. Zi Ye and Dr. Ron Mahabir from the Geographic Data Science Lab at the University of Liverpool. The module seeks to provide hands-on experience and training in introductory statistics for human geographers.

 A compilation of this web course is hosted as a GitHub repository that you can access:

- As an html website.
- As a GitHub repository.

Contact

Zi Ye - zi.ye [at] liverpool.ac.uk Lecturer in Geographic Information Science Office 107, Roxby Building, University of Liverpool - 74 Bedford St S, Liverpool, L69 7ZT, United Kingdom.

Ron Mahabir - Ron.Mahabir [at] liverpool.ac.uk Lecturer in Geographic Data Science Office 4xx, Roxby Building, University of Liverpool - 74 Bedford St S, Liverpool, L69 7ZT, United Kingdom.

Overview

Aim and Learning Objectives

This sub-module aims to provide training and skills on a set of basic quantitative skills for data collection, analysis, and interpretation and to enable you to link conceptual ideas with real world examples. **This block serves as the foundation for Year 2 BA field class and, optionally, for Year 3 dissertation.**

Background

Data and research are key pillars of the global economy and society today. We need rigorous approaches to collecting and analysing both the statistics that can tell us 'how much' and if there are observable relationships between phenomena; and the information gives us a nuanced understanding of cultural contexts and human dynamics. Quantitative skills enable us to explore and measure socio-economic activities and processes at large scales, while qualitative skills enable understanding of social, cultural, and political contexts and diverse lived experiences. Rather than being in opposition, qualitative and quantitative research can complement one another in the investigation of today's pressing research questions.

To these ends, this block will help you develop your quantitative skills, as critical tools. This course will help you understand what quantitative statistical researchers use and develop a set of research techniques that can be used in your field classes and dissertations.

Learning objectives:

- Understand how to explore a dataset, containing a number of observations described by a set of variables.
- Demonstrate an understanding in the application and interpretation of commonly used quantitative research methods.
- Ability to work with quantitative data to understand real-world social phenomenian and patterns.

Module Structure

Staff: Dr Zi Ye and Dr Ron Mahabir

Where and When

Week 1 - 5 Lecture: Tuesday (12am – 1pm) @ Mathematical Sciences, Proudman Lecture Theatre

Week 1 - 6 Practical PC session: Friday (9 – 11 am) @ Central Teaching Lab: PC Teaching Centre

Lectures will introduce and explain the fundamentals of quantitative methods, with the opportunity to apply the method introduced in the labs later in the week.

The computer practical sessions, will give you the chance to use and apply quantitative methods to real-world data. These are primarily self-directed sessions, but with support on hand if you get stuck. Support and training in R will be provided through these sessions. Weekly sessions will be driven by empirical research questions.

Week	Topic	Format	Staff
1	Introduction	Lecture	ZY/RM
	Getting Started in RStudio: Knowing Merseyside	Computer Lab Practical	
2	Exploratory Data Analysis: UK Election	Lecture and Computer Lab Practical	ZY
3	Sampling and data manipulation: Happiness around the world	Lecture and Computer Lab Practical	ZY
4	Correlation, data reliability and the issue of scale: Health	Lecture and Computer Lab Practical	RM
5	How robust are my findings	Lecture and Computer Lab Practical	RM
6	Online Assessment	Computer Lab	RM/ZY

Software and Data

For quantitative training sessions, ensure you have installed and/or have access to **RStudio**. To run the analysis and reproduce the code in R, you need the following software installed on your machine:

- R-4.2.2 (or later)
- RStudio 2022.12.0-353 (or later)

To install and update:

- R, download the appropriate version from The Comprehensive R Archive Network (CRAN).
- RStudio, download the appropriate version from here.

This software is already installed on University Machines. But you will need it to run the analysis on your personal devices.

Data

Example datasets could be accessed through ENVS 162 Canvas module every week.

Assessment

Week 6 Computer-based 'open book' multiple-choice exam

- The online assessment will be released at **4pm on Thursday 5th March** and should be c**ompleted by 4pm on Friday 6th March.**
- **Also available 06/03/2025 9:00 – 11:00 CTL PC Teaching Centre (1st Floor CTL)**
- Should take less **90 minute**s; c. 20 questions; 24 hours to complete
- Questions and answers randomised for each student (anti-cheating measure)
- Some questions of factual recall, more requiring data analysis to find answers

Preparation for assessment

- Weekly lecture & weekly computer practical 'clinic sessions'
- Weekly holding hands formative tasks at the last 20 mins of the practical session
- Week 5 mock online test

1 Lab: Getting Started in RStudio - Knowing Merseyside

1.1 Overview

This practical intend to prepare students who have limited experiences with R and RStudio. The content are adapted based on

- Brunsdon, Chris, and Lex Comber. 2018. *An Introduction to r for Spatial Analysis and Mapping (2e)*. Sage.
- Comber, Lex, and Chris Brunsdon. 2021. *Geographical Data Science and Spatial Data Analysis: An Introduction in r*. Sage.

1.2 Getting set up with RStudio

1.2.1 Install R and RStudio (if necessary)

R is a free, open-source programming language used for statistical analysis, data visualization, and data science

RStudio is a free front-end to R, designed to make using R easier

All of the PCs in the University PC Teaching Centre used for this class come with R and RStudio pre-installed, as do the PCs in many other University PC Teaching Centres.

However, you may wish to install R and RStudio on your own computer, or on a University PC that lacks them.

University computers: Use the *Install University Applications* app on the computer to install the latest version of RStudio (this will also install the latest version of R)

Your own computer: R and RStudio can be downloaded from the CRAN website and installed your own computer - see below for details. **A key point is that you should install R before you install RStudio.**

The simplest way to get R installed on your computer is to go the download pages on the R website - a quick search for 'download R' should take you there, but if not you could try:

- Windows: https://cran.r-project.org/bin/windows/base/
- Mac: https://cran.r-project.org/bin/macosx/
- Linux: http://cran.r-project.org/bin/linux/

The Windows and Mac version come with installer packages and are easy to install whilst the Linux binaries require use of a command terminal.

RStudio can be downloaded from https://www.rstudio.com/products/rstudio/download/ and the free version of RStudio Desktop is more than sufficient for this module and all the other things you will to do at degree level.

If you experience any problems installing R or RStudio on your own computer, bring it to one of the class lab sessions where we will be able to provide advice.

1.2.2 File management

Before you start installing software or downloading data, create a folder on your M-Drive (if working on a University networked machine) or locally if working on your own device – name this 'ENVS162' and within this create a sub-folder for each practical session. For this session, create a sub-folder called `Week1` in your `ENVS162` folder on your M-Drive. Take care to ensure you do not delete any work you do complete in the practical sessions. It is imperative that you practice good file management!

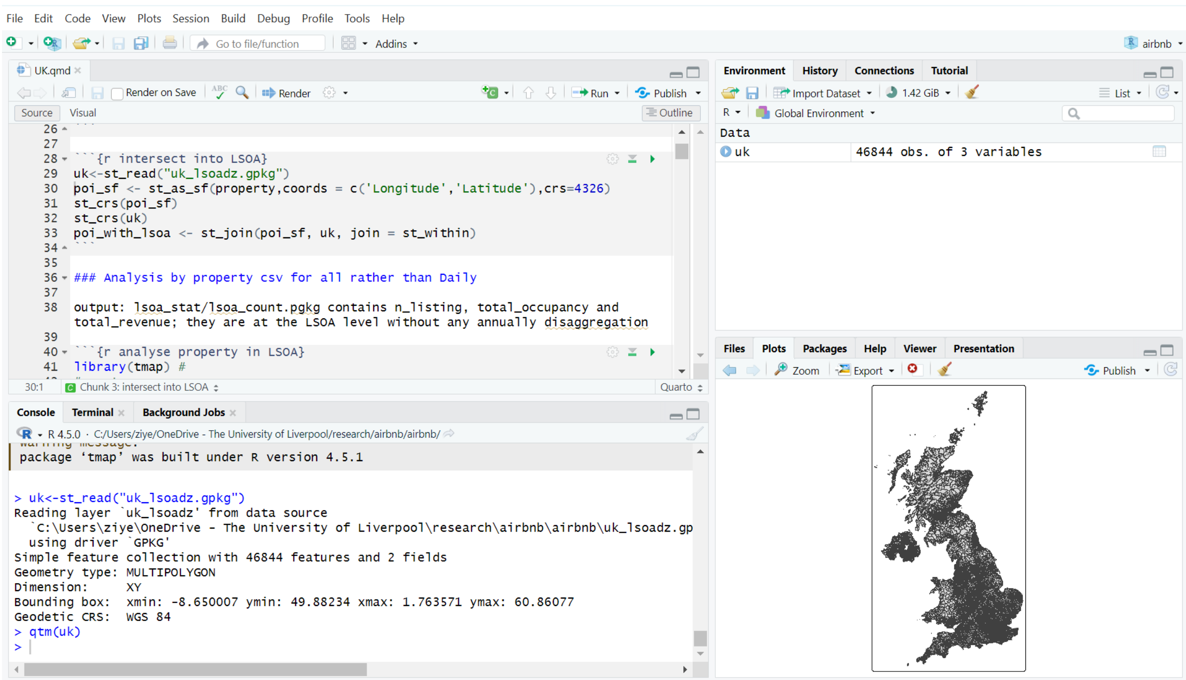
1.2.3 Open RStudio

RStudio provides an interface to the different things that R can do via the 4 panes: the Console where code is entered (bottom left), a Source pane with R scripts (top left), the variables in the working Environment (top right), Files, Plots, Help etc (bottom right) - see the RStudio environment in Figure below.

In the figure above of the RStudio interface, a new script has been opened, a line of code had been written and then run in the console. The code assigns a value of 100 to `x`. The file has been saved into the current working environment. You are expected to define a similar set up for each practical as you work through the code. Note that **in the script**, anything that follows a `#` is a comment and ignored by R.

Users can set up their personal preferences for how they like their RStudio interface. Similar to straight R, there are very few pull-down menus in R, and therefore you will type lines of code into your script and run these in what is termed a *command line interface* (the console). Like all command line interfaces, the learning curve is steep but the interaction with the software is more detailed which allows greater flexibility and precision in the specification of commands.

Beyond this there are further choices to be made. Commands can be entered in two forms: directly into the *R console* window or as a series of commands into a script window. We strongly advise that all code should be **written in a script** - (a `.R` file) and then **run from the script**. - To run code in a script, place the cursor on the line of code and then run by pressing the 'Run' icon at the top left of the script pane, or by pressing **Ctrl Enter** (PC) (or **Cmd Enter** on a Mac).



1.2.4 Ways of working

The first set of consideration relate to *how* you should work in R/RStudio. The key things to remember are:

- R is a learning curve if you have never done anything like this before. It can be scary. It can be intimidating. But once you have a bit of familiarity with how things work, it is incredibly powerful.
- You will be working from practical worksheets which will have all the code you need. Your job is to try to **understand** what the code is doing and **not** to remember the code. Comments in your code really help.
- To help you do this, the very strong suggestion is use the R scripts that are provided, and that you add your own comments to help you understand what is going on when you return to them. Comments are prefaced by a hash (`#`) that is ignored by R. Then

you can save your code (with comments), run it and return to it later and modify at your leisure.

The module places a strong emphasis placed on learning by doing, which means that you encouraged to unpick the code that you are given, adapt it and play with it. It is not about remembering or being able to recall each function used but about understanding what is being done. If you can remember what you did previously (i.e. the operations you undertook) and understand what you did, you will be able to return to your code the next time you want to do something similar. To help you with this you should:

1. Always run your code from an R script… **always**! These are provided for each practical;
2. Annotate you scripts with comments. These are prefixed by a hash (`#`) in the code;
3. Save your R script to your folder.

In Summary:

- You should always use a script (a text file containing code) for your code which can be saved and then re-run at a later date.
- You can write your own code into a script, copy and paste code into it or use an existing script (for example as provided for each of the R/RStudio practicals in this module).
- To open a new R script go to File > New File > R Script to open a new R file, and save it with a sensible name.
- To load an existing script file go to File > Open File and then navigate to your file. Or, if you have recently opened the file, go to File > Recent Files >.
- It is good practice to set the working directory at the beginning of your R session. This can be done via the menu in RStudio Session > Set Working Directory > …. This points the R session to the folder you choose and will ensure that any files you wish to read, write or save are placed in this directory.
- To run code in a script, place the cursor on the line of code and then run by pressing the 'Run' icon at the top left of the script pane, or by pressing Ctrl Enter (PC) or Cmd Enter (Mac).

1.2.5 Your first R code

In this section you will undertake a few generic operations. You will:

- undertake **assignment**: the allocation of values to an R object.
- use assignment to create a **vector** of elements and a **matrix** of elements.
- undertake **operations** on R objects.

- apply some **functions** to R objects (functions nearly always return a value).
- access some of R in-built data to examine a data table (or `data.frame` which is like an Excel spreadsheet).
- do some basic **plotting**, including scatter plots and histograms.
- create data summaries.

On the way you will also be introduced to **indexing**.

First, you should **create a new R script** (see above) and save it as `week1.R` in the working directory you are using for this practical. This should be the `Week1` sub-directory you created in the `ENVS162` folder. Note that you should create a separate folder for each week's practical.

1.2.5.1 Assignment

The command line prompt in the Console window, the `>`, is an invitation to start typing in your commands.

Write the following into your script: `3+5` and run it. Recall that code is run done by either by pressing the Run icon at the top left of the script pane, or by pressing **Ctrl Enter** (PC) or **Cmd Enter** (Mac).

```
3+5
```

```
[1] 8
```

Here the result is 8. The `[1]` that precedes the output it formally indicates, *first requested element will follow*. In this case there is just one element. The `>` indicates that R is ready for another command.

Now type the following in to your script and run it:

```
y <- 3+5
y
```

```
[1] 8
```

Here the value of the `3+5` has been ***assigned*** to `y`. The syntax `y <- 3+5` can be read as `y` ***gets*** `3+5`. When `y` is invoked its value is returned (8).

For the purposes of this module, in R the equals sign (`=`) is the same as `<-`, a left diamond bracket `<` followed by a minus sign `-`. This too is interpreted by R as ***is assigned to*** or ***gets*** when the code is read **right to left**.

Now copy and paste the following into your R script and run both lines:

```
x <- matrix(c(1,2,3,4,5,6,7,8), nrow = 4)
y = matrix(1:8, nrow = 4, byrow = T)
```

You should see the `x` appear with the `y` in the Environment pane. `y` has now been overwritten with a new assignment. If you click on the icon next to them, you will get a 'spreadsheet' view of the data you have created.

Of course you can also enter the following in the console and see what is returned:

```
x
```

```
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

```
y
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
```

Note In the code snippets above you have used `parentheses` - round brackets. Different kinds of brackets are used in different ways in R. Parentheses are used with **functions**, and contain the **arguments** that are passed to the function, separated by commas (`,`).

In this case the functions are `c()` and `matrix()`. The function `c()` combines or concatenates elements into a vector, and `matrix()` creates a matrix of elements in a tabular format.

In the line of code `x = matrix(c(1,2,3,4,5,6,7,8), nrow = 4)`, the arguments passed to the `matrix()` function are the vector of values `c(1,2,3,4,5,6,7,8)` and `nrow = 4`. Other kinds of brackets are used in different ways as you will see later.

One final thing to note is that the matrix is `y` is has the numbers 1 to 8, but this is specified by `1:8`. Try entering `3:65`, `19:11`, and `1.5:5` to see how the colon (`:`) works in this context.

1.2.5.2 Operations

Now you can undertake *operations* on R objects and apply *functions* to them. Write the following code into your script and then run it:

```
# x is a matrix
x
```

```
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

```
# multiplication
x*2
```

```
     [,1] [,2]
[1,]    2   10
[2,]    4   12
[3,]    6   14
[4,]    8   16
```

```
# sum of x
sum(x)
```

```
[1] 36
```

```
# mean of x
mean(x)
```

```
[1] 4.5
```

Operations can be undertaken directly using mathematical notation like `*` for multiplication or using functions like `max` to find the maximum value in an R object.

1.2.5.3 Functions

Functions are always followed by parenthesis (round brackets) `( )`. These are different from square and curly brackets `[ ]` and `{ }`. Functions always return something, a result if you like, and have the generic form:

```
# don't run this or write this into your script!
result = function(value or R object, other parameters)
```

Do not run or enter this code in your script - it is an example!

1.2.5.4 Data Tables

Here we will load a data table in `data.frame` (like a spreadsheet) in R/RStudio. R has number of in-built datasets that we can use the code below loads one of these:

```
data(mtcars)
class(mtcars)
```

```
[1] "data.frame"
```

Have a look at what is loaded by listing the objects in the current R session

```
ls()
```

```
[1] "mtcars" "x"      "y"
```

You should see the `mtcars` object. You can examine this data in a number of ways

```
# the structure of mtcars
str(mtcars)
```

```
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
```

```
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

```r
# the first six rows (or head) of mtcars
head(mtcars)
```

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

The `mtcars` object is a `data.frame`, a kind of data table, and it has a number of attributes which are all numeric. The code below prints it all out to the console:

```r
mtcars
```

```
                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
```

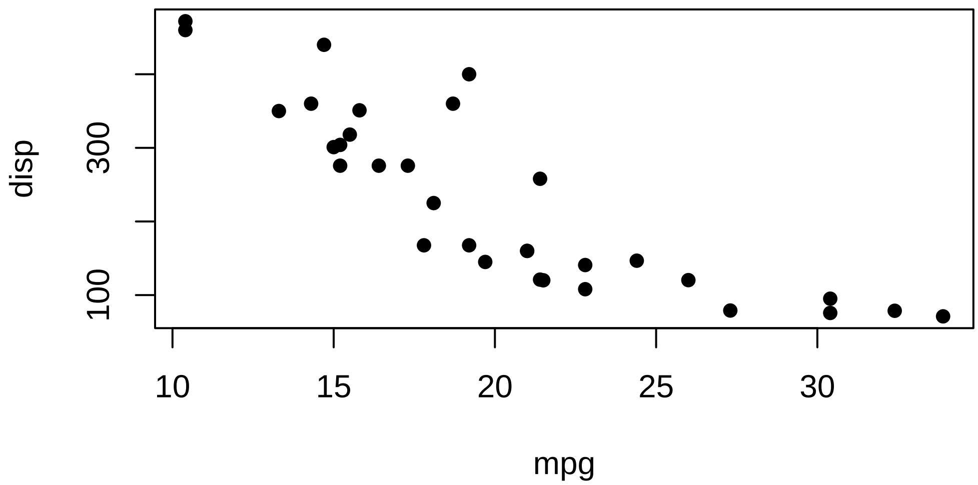
```
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Data frames are 'flat' in that they typically have a rectangular layout like a spreadsheet, with rows typically relating to observations (individuals, areas, people, houses, etc) and columns relating to their properties or attributes (height, age, etc). The columns in data frames can be of different types: vectors of numbers, factors (classes) or text strings. In matrices all of the columns have to be off the same type. Data frames are central to what we will do in R.

1.2.5.5 Plotting the data: 'Hello World!'

The code below creates a plot of 2 variables counts in the data: `mpg` and `disp`.

```
plot(disp ~ mpg,  data = mtcars, pch=16)
```



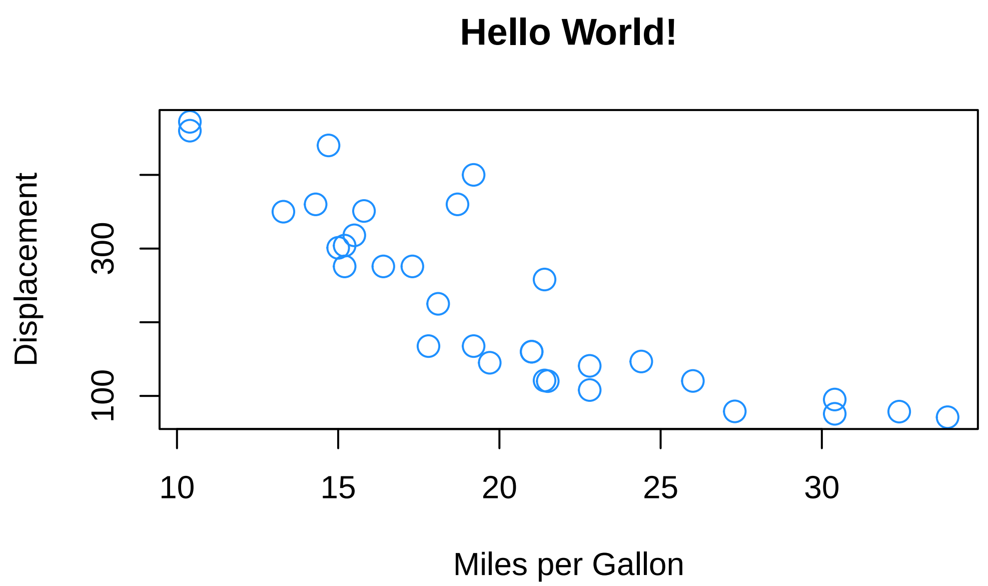
The option `pch=16` sets the plotting character to a solid black dot. More plot characters are available - examine the help for `points()` to see these (For any command, if you are the first time use it, you can always ask R to explain to you by using ? as help)

```
?points
```

This plot can be improved greatly. We can specify more informative axis labels, change size of the text and of the plotting symbol, and so on.

We can also specify the same plot by passing named variables to the `plot` function directly as well as other parameters, as in the figure. Notice how the syntax for this is different to the `plot` function above, and the different **parameters** that are passed to the `plot()` function:

```
plot(x = mtcars$mpg, y = mtcars$disp,   pch = 1, col = "dodgerblue",
     cex = 1.5, xlab = "Miles per Gallon", ylab = "Displacement",
     main = "Hello World!")
```



Notice how the dollar sign (`$`) is used to access variables in the `mtcars` data table compared to the first plot command, which specified `data = mtcars`.

1.2.5.6 Data summaries and indexing

We may for example require information on variables in `mtcars`. The `summary` function is very useful:

```
summary(mtcars)
```

```
      mpg             cyl             disp             hp       
 Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
 1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
 Median :19.20   Median :6.000   Median :196.3   Median :123.0  
 Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
 Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
      drat             wt             qsec             vs        
 Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000  
 1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000  
 Median :3.695   Median :3.325   Median :17.71   Median :0.0000  
 Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375  
```

```
 3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
 Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
       am              gear            carb
 Min.   :0.0000   Min.   :3.000   Min.   :1.000
 1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
 Median :0.0000   Median :4.000   Median :2.000
 Mean   :0.4062   Mean   :3.688   Mean   :2.812
 3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
 Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

This shows different summaries of the individual attributes in `mtcars`.

The main R graphics function is `plot()`. In addition to `plot()` there are functions for adding points and lines to existing graphs, for placing text at specified positions, for specifying tick marks and tick labels, for labelling axes, and so on.

There are various other alternative helpful forms of graphical summary. A helpful graphical summary for the `mtcars` data frame is the scatterplot matrix.

```r
# return the names of the mtcars variables
names(mtcars)
```

```
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"
```

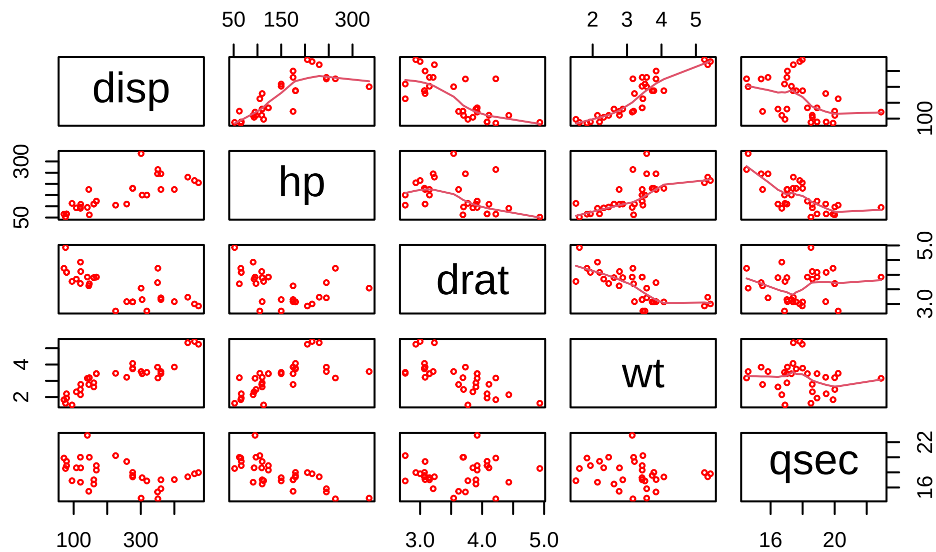
```r
# return the 3rd to 7th names
names(mtcars)[c(3:7)]
```

```
[1] "disp" "hp"   "drat" "wt"   "qsec"
```

```r
# check what this does
c(3:7)
```

```
[1] 3 4 5 6 7
```

```r
# plot the 3rd to 7th variables in mtcars
plot(mtcars[, c(3:7)], cex = 0.5,
     col = "red", upper.panel=panel.smooth)
```



The results show the correlations between the variables in the `mtcars` data frame, and the trend of their relationship is included with the `upper.panel=panel.smooth` parameter passed to `plot`.

There are number of things to notice here (as well as the figure). In particular note the use of the vector `c(2:7)` to subset the columns of `mtcars`:

- In the second line, this is was used to subset the vector of column names created by `names(mtcars)`.
- In the third line, it was printed out. Notice how `3:7` printed out all the number between 3 and 7 - very useful.
- For the plot, the vector was passed to the second argument, after the comma, in the square brackets `[,]` to indicate which columns were to be plotted.

The referencing in this way (or *indexing*) is **very important**: the individual rows and columns of 2 dimensional data structures like data frames, matrices, tibbles etc can be accessed by passing references to them in the square brackets.

```
# 1st row
mtcars[1,]
```

```
          mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4  21   6  160 110  3.9 2.62 16.46  0  1    4    4
```

```r
# 3rd column
mtcars[,3]
```

```
 [1] 160.0 160.0 108.0 258.0 360.0 225.0 360.0 146.7 140.8 167.6 167.6 275.8
[13] 275.8 275.8 472.0 460.0 440.0  78.7  75.7  71.1 120.1 318.0 304.0 350.0
[25] 400.0  79.0 120.3  95.1 351.0 145.0 301.0 121.0
```

```r
# a selection of rows
mtcars[c(3:5,8),]
```

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
```

Such indexing could of course have been assigned to a R object and used to do the subsetting:

```r
x = c(3:7)
names(mtcars)[x]
```

```
[1] "disp" "hp"   "drat" "wt"   "qsec"
```

Thus indexing allows specific rows and columns to be extracted from the data as required.

Note You have encountered a second type of brackets, square brackets `[ ]`. These are used to reference or **index** positions in a vector or a data table.

Consider the object `x` above. It contains a vector of values, `3,4,5,6,7`. Entering `x[1]` would extract the first element of `x`, in this case 3. Similarly `x[4]` would return the 4th element and `x[c(1,4)]` would return the 1st and 4th elements of `x`.

However, in the examples above that index the 2-dimensional `mtcars` object, the square brackets are used to index **row** and **column** positions. The syntax for this is `[rows, columns]`. We will be using such indexing throughout this module.

You can ask R to return you specific rows and columns by different ways:

```r
mtcars[c(2,9), 3:7]
```

```
              disp  hp drat    wt  qsec
Mazda RX4 Wag 160.0 110 3.90 2.875 17.02
Merc 230      140.8  95 3.92 3.150 22.90
```

```r
mtcars[3:6, c("disp","hp","qsec")]
```

```
                  disp  hp  qsec
Datsun 710         108  93 18.61
Hornet 4 Drive     258 110 19.44
Hornet Sportabout  360 175 17.02
Valiant            225 105 20.22
```

```r
mtcars [, c("wt","gear","cyl")]
```

```
                       wt gear cyl
Mazda RX4           2.620    4   6
Mazda RX4 Wag       2.875    4   6
Datsun 710          2.320    4   4
Hornet 4 Drive      3.215    3   6
Hornet Sportabout   3.440    3   8
Valiant             3.460    3   6
Duster 360          3.570    3   8
Merc 240D           3.190    4   4
Merc 230            3.150    4   4
Merc 280            3.440    4   6
Merc 280C           3.440    4   6
Merc 450SE          4.070    3   8
Merc 450SL          3.730    3   8
Merc 450SLC         3.780    3   8
Cadillac Fleetwood  5.250    3   8
Lincoln Continental 5.424    3   8
Chrysler Imperial   5.345    3   8
Fiat 128            2.200    4   4
Honda Civic         1.615    4   4
Toyota Corolla      1.835    4   4
Toyota Corona       2.465    3   4
Dodge Challenger    3.520    3   8
AMC Javelin         3.435    3   8
Camaro Z28          3.840    3   8
Pontiac Firebird    3.845    3   8
Fiat X1-9           1.935    4   4
```

```
Porsche 914-2       2.140    5   4
Lotus Europa        1.513    5   4
Ford Pantera L      3.170    5   8
Ferrari Dino        2.770    5   6
Maserati Bora       3.570    5   8
Volvo 142E          2.780    4   4
```

1.2.5.7 Packages

The `base` installation of R includes many functions and commands. However, more often we are interested in using some particular functionality, encoded into **packages** contributed by the R developer community. Installing packages for the first time can be done at the command line in the R console using the `install.packages` command as in the example below to install the `tmap` library or via the RStudio menu via **Tools > Install Packages**.

When you install these packages it is strongly suggested you also install the *dependencies*. These are other packages that are required by the package that is being installed. This can be done by selecting check the box in the menu or including `dep=TRUE` in the command line as below (don't run this yet!):

```
# don't run this!
install.packages("tidyverse", dep = TRUE)
```

You may have to set a **mirror** site from which the packages will be downloaded to your computer. Generally you should pick one that is nearby to you.

Further descriptions of packages, their installation and their data structures will be given as needed in the practicals. There are literally 1000s of packages that have been contributed to the R project by various researchers and organisations. These can be located by name at http://cran.r-project.org/web/packages/available_packages_by_name.html if you know the package you wish to use. It is also possible to search the CRAN website to find packages to perform particular tasks at http://www.r-project.org/search.html. Additionally many packages include user guides and vignettes as well as a PDF document describing the package and listed at the top of the index page of the help files for the package.

As well as `tidyverse` you should install the `sf` package and dependencies. So we have 2 packages to install:

- `sf` for spatial data and spatial objects
- `tidyverse` for lots of lovely data science things - see https://www.tidyverse.org

You could do this in one go and this will take a bit of time:

```r
install.packages(c("sf", "tidyverse"), dep = TRUE)
```

Remember: you will only have to install a package once!! So when the above code has run in your script you should comment it out. For example you might want to include something like the below in your R script.

```r
# packages only need to be loaded once
# install.packages(c("sf", "tidyverse"), dep = TRUE)
```

Once the package has been installed on your computer then the package can be called using the `library()` function into each of your R sessions as below.

```r
library(tidyverse)
library(sf)
```

1.3 Knowing Merseyside

1.3.1 Merseyside districts

Now we use these basic R command and newly installed packages to start our initial exploration by using some existing secondary dataset from the Census 2021.

In R we normally read in tabular dataset from .csv format. In your ENVS162 Canvas page find Week 1 -> Practical 1 Dataset, download the four datasets to your current working folder on your M drive (ENVS162 - Week 1). You may first identify one `.csv` dataset: **merseyside.csv**. You can open them in excel to have a look, but here we are using R instead of Excel to load and examine them.

1.3.1.1 Loading tabular data

The survey data can be loaded into RStudio using the `read.csv` function.

However, you will need to tell R where to get the data from. The easiest way to do this is to use the menu if the R script file is open. Go to **Session > Set Working Directory > To Source File Location** to set the working directory to the location where your `week1.R` script is saved. When you do this you will see line of code print out in the Console (bottom left pane) similar to `setwd("SomeFilePath")`. You can copy this line of code to your script and paste into the line above the line calling the `read.csv` function.

```
# use read.csv to load a CSV file
# this is assignment to an object called `df`
df = read.csv(file = "merseyside.csv", stringsAsFactors = TRUE)
```

The `stringsAsFactors = TRUE` parameter tells R to read any character or text variables as classes or categories and not as just text.

You could inspect the help for the `read.csv` function to see the different parameters and their default values:

```
help(read.csv)
```

```
starting httpd help server ... done
```

```
# or
?read.csv
```

Functions always return something and in this case `read.csv()` function has returned a tabular R object with 5 records and 12 fields. This has been *assigned to* `df`.

Finally in this section, lets have a look at the data. This can be done in a number of ways.

- you could look at the `df` object by entering `df` in the Console. However this is not particular helpful as it simply prints out everything that is in `df` to the Console.
- you could click on the `df` object in the Environment pane and this shows the structure of the attributes in different fields.
- you could click on the little grid-like icon next `df` in the Environment pane to get a `View` of the data and remember to close the tab that opens!.
- or you could use some code as in the examples below.

First, let's have a look at the internal structure of the data using the `str` function:

```
str(df)
```

```
'data.frame':	5 obs. of  12 variables:
 $ LAD21CD           : Factor w/ 5 levels "E08000011","E08000012",..: 1 2 3 4 5
 $ District          : Factor w/ 5 levels "Knowsley","Liverpool",..: 1 2 4 3 5
 $ Population        : int  154519 486089 183248 279234 320196
 $ Households        : int  66073 207491 81011 123075 143253
 $ Working_population: int  69495 205749 82622 124596 139500
```

```
 $ Students          : int  7050 59628 7582 12636 14642
 $ Unemployed        : int  3852 13894 4076 6143 6542
 $ Age_over_65       : int  26242 74322 37642 64763 70391
 $ Disability        : int  34990 105962 40829 61134 73088
 $ No_central_heating: int  1020 4822 1003 1965 2125
 $ Overcrowding      : int  1892 7352 1888 2700 2355
 $ Working_from_home : int  14880 53721 18973 34750 37299
```

There is other ways to get info about the number of rows and columns:

```r
nrow(df)
```

```
[1] 5
```

```r
ncol(df)
```

```
[1] 12
```

```r
#or both row and col
dim(df)
```

```
[1]  5 12
```

The `head` function does this by printing out the first six records of the data table and you may need to scroll up and down in the Console pane to see all of what is returned.

```r
head(df)
```

```
    LAD21CD   District Population Households Working_population Students
1 E08000011   Knowsley     154519      66073              69495     7050
2 E08000012  Liverpool     486089     207491             205749    59628
3 E08000013 St. Helens     183248      81011              82622     7582
4 E08000014     Sefton     279234     123075             124596    12636
5 E08000015     Wirral     320196     143253             139500    14642
  Unemployed Age_over_65 Disability No_central_heating Overcrowding
1       3852       26242      34990               1020         1892
2      13894       74322     105962               4822         7352
3       4076       37642      40829               1003         1888
4       6143       64763      61134               1965         2700
```

```
5       6542       70391      73088                2125          2355
  Working_from_home
1             14880
2             53721
3             18973
4             34750
5             37299
```

Another way to explore the data is through the `summary` function:

```
summary(df)
```

```
     LAD21CD          District      Population        Households
 E08000011:1   Knowsley  :1   Min.   :154519   Min.   : 66073
 E08000012:1   Liverpool :1   1st Qu.:183248   1st Qu.: 81011
 E08000013:1   Sefton    :1   Median :279234   Median :123075
 E08000014:1   St. Helens:1   Mean   :284657   Mean   :124181
 E08000015:1   Wirral    :1   3rd Qu.:320196   3rd Qu.:143253
                              Max.   :486089   Max.   :207491
 Working_population    Students       Unemployed     Age_over_65
 Min.   : 69495     Min.   : 7050   Min.   : 3852   Min.   :26242
 1st Qu.: 82622     1st Qu.: 7582   1st Qu.: 4076   1st Qu.:37642
 Median :124596     Median :12636   Median : 6143   Median :64763
 Mean   :124392     Mean   :20308   Mean   : 6901   Mean   :54672
 3rd Qu.:139500     3rd Qu.:14642   3rd Qu.: 6542   3rd Qu.:70391
 Max.   :205749     Max.   :59628   Max.   :13894   Max.   :74322
   Disability     No_central_heating  Overcrowding  Working_from_home
 Min.   : 34990   Min.   :1003       Min.   :1888   Min.   :14880
 1st Qu.: 40829   1st Qu.:1020       1st Qu.:1892   1st Qu.:18973
 Median : 61134   Median :1965       Median :2355   Median :34750
 Mean   : 63201   Mean   :2187       Mean   :3237   Mean   :31925
 3rd Qu.: 73088   3rd Qu.:2125       3rd Qu.:2700   3rd Qu.:37299
 Max.   :105962   Max.   :4822       Max.   :7352   Max.   :53721
```

Finally in this section, we come back to the dollar sign (`$`). This is used to refer to or *extract* an individual named field or variable in an R object, like `df`.

The code below prints out the Population attribute and generates a summary of its values:

```
# extract an individual variable
df$Population
```

```
[1] 154519 486089 183248 279234 320196
```

```
# generate a summary of an individual variable
summary(df$Population)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
 154519  183248  279234  284657  320196  486089 
```

And of course we can use such operations to *assign* the result to new R objects. The code below extracts three variables from `df`, assigns them to `x`, `y` and `z`, and then uses the `data.frame` function to convert these into a new `data.frame` object called `my_df`

```
# extract three variables, assigning them to temporary R objects
x = df$District
y = df$Working_population
z = df$Students
# create a data.frame from these, naming the new variables
my_df = data.frame(district = x,worker = y,student = z)
```

You should have a look at what you have created:

```
head(my_df)
```

```
    district worker student
1   Knowsley  69495    7050
2  Liverpool 205749   59628
3 St. Helens  82622    7582
4     Sefton 124596   12636
5     Wirral 139500   14642
```

```
summary(my_df)
```

```
       district      worker          student     
 Knowsley   :1   Min.   : 69495   Min.   : 7050  
 Liverpool :1    1st Qu.: 82622   1st Qu.: 7582  
 Sefton     :1   Median :124596   Median :12636  
 St. Helens:1    Mean   :124392   Mean   :20308  
 Wirral     :1   3rd Qu.:139500   3rd Qu.:14642  
                 Max.   :205749   Max.   :59628  
```

The temporary R objects can be removed from the Environment using the `rm` function and a *combine* vector function, `c()` that you encountered in Week 19, that takes a vector of object names (hence they are in quotes) as its arguments.

```
rm(list = c("x","y","z"))
```

1.3.1.2 Basic data manipulation

Now we can do some basic data manipulation to know Merseyside more from the data perspective.

What is the total population in Merseyside?

```
sum(df$Population)
```

```
[1] 1423286
```

What is the total number of full-time students in Merseyside?

```
sum(df$Students)
```

```
[1] 101538
```

Then, we can calculate the total number of workers that working from home:

```
sum(df$Working_from_home)
```

```
[1] 159623
```

What is the proportion of working population actually work from home in Merseyside? Yes, we need to use a division calculation of the total number of working from home vs. all the working population. R can do it by:

```
sum(df$Working_from_home) / sum(df$Working_population)
```

```
[1] 0.2566443
```

So the answer is 25.7% for the whole Merseyside - but which district has the highest proportion and which as the lowest? You may have your own guessing. But let R do the calculation:

```r
df$Prop.WFH = df$Working_from_home / df$Working_population * 100 #add a new column called Pro
df #print out the df
```

```
    LAD21CD    District Population Households Working_population Students
1 E08000011    Knowsley     154519      66073              69495     7050
2 E08000012   Liverpool     486089     207491             205749    59628
3 E08000013 St. Helens     183248      81011              82622     7582
4 E08000014      Sefton     279234     123075             124596    12636
5 E08000015      Wirral     320196     143253             139500    14642
  Unemployed Age_over_65 Disability No_central_heating Overcrowding
1       3852       26242      34990               1020         1892
2      13894       74322     105962               4822         7352
3       4076       37642      40829               1003         1888
4       6143       64763      61134               1965         2700
5       6542       70391      73088               2125         2355
  Working_from_home Prop.WFH
1             14880 21.41161
2             53721 26.10997
3             18973 22.96362
4             34750 27.89014
5             37299 26.73763
```

Here we ask R to add a new column named `Prop.WFH` which is the working from home proportion that calculated by the number of working from home people in each district divided by the total working population in that district. The `* 100` convert the rate in the percentage number. R will automatically do it row-by-row. We then print out the `df`, you may find at the very right end of the tabular, there is a new column called `Prop.WFH`.

For a very small dataframe like this, we can also using View() to open a new tab to review the data, where each column can be sorted from largest to smallest or vice versa. Try viewing it and find the newly created column `Prop.WFH`. Click on the column name, you should see it is sorted from highest to lowest, and click again, the ranking is reversed.

```r
View(df)
```

1.3.1.3 Your first map for Merseyside

Now let's try to do our first map in R and allow yourself know more about Merseyside.

We will use the library sf and tmap to help us at here. Run the install codes if you haven't install them. Remember: you will only have to install a package once!!

```r
install.packages("tmap",dep =TRUE)
```

Check the package version of tmap, as here we need to use tmap over 4.0 version.

```r
packageVersion("tmap") # the version should over 4.0
```

```
[1] '4.1'
```

When they have been installed, we can start using them

```r
library(tidyverse)
library(sf)
library(tmap)
```

You may find in Week 1 data, we have another file named *merseyside_districts.gpkg*. A GeoPackage (GPKG) is a file-based format designed for storing geographic data. It supports the efficient storage and exchange of spatial datasets and can be readily used across GIS software such as QGIS and ArcGIS, as well as in programming environments including R and Python.

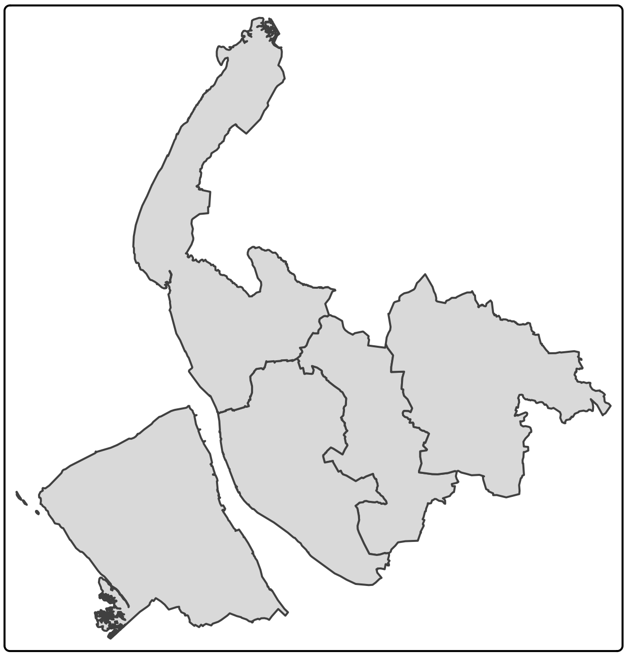
We first read it in by using the `st_read()` command in library sf.

```r
sf <- st_read("merseyside_districts.gpkg")
```

```
Reading layer `lad_may_2025_uk_bgc_v2_4306843991635065087__lad_may_2025_uk_bgc_v2' from data
  using driver `GPKG'
Simple feature collection with 5 features and 8 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 318351.7 ymin: 377515.4 xmax: 361796.3 ymax: 422866.5
Projected CRS: OSGB36 / British National Grid
```

The fastest way to map it is the `qtm()` function.

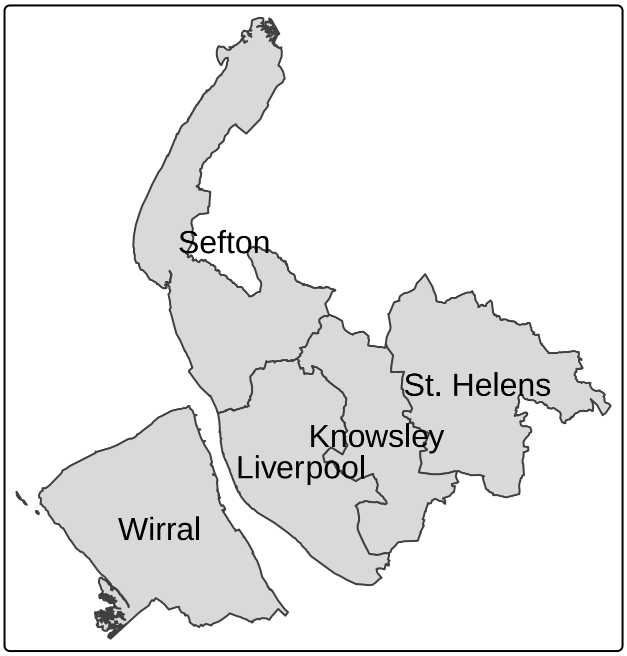
```r
qtm(sf)
```



You can also add the district names on the map - which column in the sf contains district name? Use `names(sf)` to check for it.

Yes, the column should be `LAD25NM`. Now let's ask `qtm()` to also show the district names.

```
qtm(sf,text="LAD25NM")
```



But what if we want to make some meaningful maps, rather than just the boundaries of these five districts of Merseyside?

1.3.1.4 Link tabular data to geographical boundaries

Recall that in our `df`, we have 14 columns, containing different information about the districts. We can get all their names by using `names()`.

```
names(df)
```

```
 [1] "LAD21CD"            "District"           "Population"
 [4] "Households"         "Working_population" "Students"
 [7] "Unemployed"         "Age_over_65"        "Disability"
[10] "No_central_heating" "Overcrowding"       "Working_from_home"
[13] "Prop.WFH"
```

We can do the same thing for our geographical dataset `sf` to see what it includes:

```
names(sf)
```

```
[1] "LAD25CD"  "LAD25NM"  "LAD25NMW" "BNG_E"    "BNG_N"    "LONG"     "LAT"
[8] "GlobalID" "geom"
```

We can also show the whole `sf` as

```
sf
```

```
Simple feature collection with 5 features and 8 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 318351.7 ymin: 377515.4 xmax: 361796.3 ymax: 422866.5
Projected CRS: OSGB36 / British National Grid
    LAD25CD    LAD25NM LAD25NMW  BNG_E  BNG_N      LONG      LAT
1 E08000011   Knowsley          344762 393778 -2.832979 53.43789
2 E08000012  Liverpool          339359 390556 -2.913680 53.40833
3 E08000013 St. Helens          353413 395992 -2.703093 53.45862
4 E08000014     Sefton          334282 398835 -2.991771 53.48213
5 E08000015     Wirral          329109 386965 -3.067034 53.37478
                                GlobalID                           geom
1 {B4196BFE-EE90-4C31-ABD5-C7E743AE2F9B} MULTIPOLYGON (((341447.1 40...
2 {4FB47E7A-EF4E-4B9E-BF75-D4FC059CDE61} MULTIPOLYGON (((338860.9 39...
3 {943F0C6B-EB30-4C00-A42B-F6B3AEC3EFEE} MULTIPOLYGON (((349111.4 40...
4 {C6FD073B-CBEB-4E78-934A-A8FD11A20F0A} MULTIPOLYGON (((336374.5 42...
5 {88E9328B-371C-469C-91F1-3479C77D6950} MULTIPOLYGON (((331364.9 39...
```

Now we see that sf includes also the five districts, but also other geographical information. You may notice that although different column names, the first two columns of both df and sf are the district code and district name. This means what potentially we can link this two dataset together - appendix the df to sf to enrich the attributes of our geographical dataset.

```
merseyside <- left_join(sf, df,by=c("LAD25NM"="District"))
```

let's check out the new `sf2` by `View()` it:

```
View(merseyside)
```

In the open tab, we see all the `df` columns are now also attached to the `sf`, linking by the district names.

1.3.1.5 Choropleth map of Merseyside districts

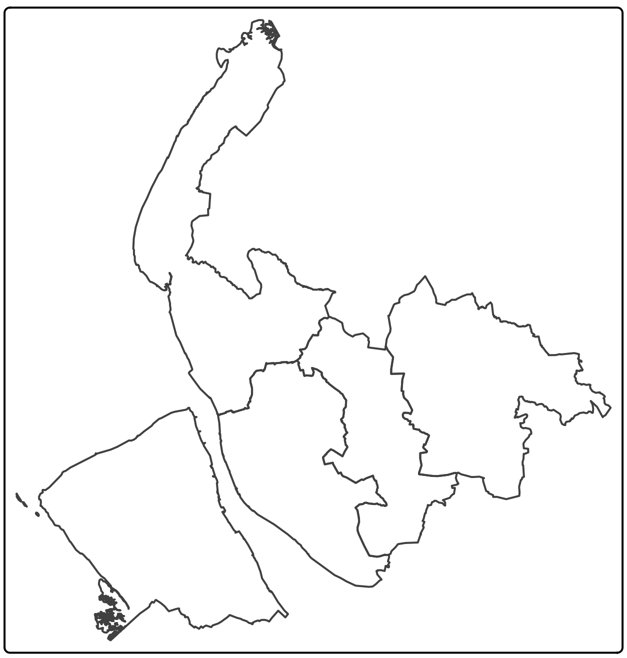
Now, we can use those new columns we attached from `df` to `sf2` to make some meaningful choropleth maps! Here we make use of the mapping functions in tmap (Remember to run `library(tmap)` if you haven't) to do the work for us.

`tmap` has a basic syntax (again, do not run this code - its is simply showing the syntax of `tmap`):

```
# don't run this or write this into your script!
tm_shape(data = <data>)+
  tm_<function>(<variable to be mapped>)
```

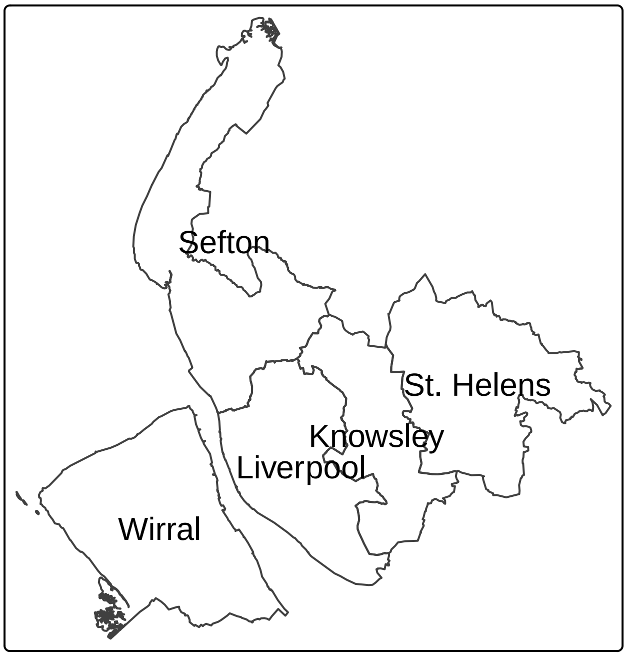
For example, to map the boundaries of `merseyside`:

```
tm_shape(merseyside) +
  tm_borders()
```



To add label of district:

```
tm_shape(merseyside) +
  tm_borders() +
  tm_text("LAD25NM")
```

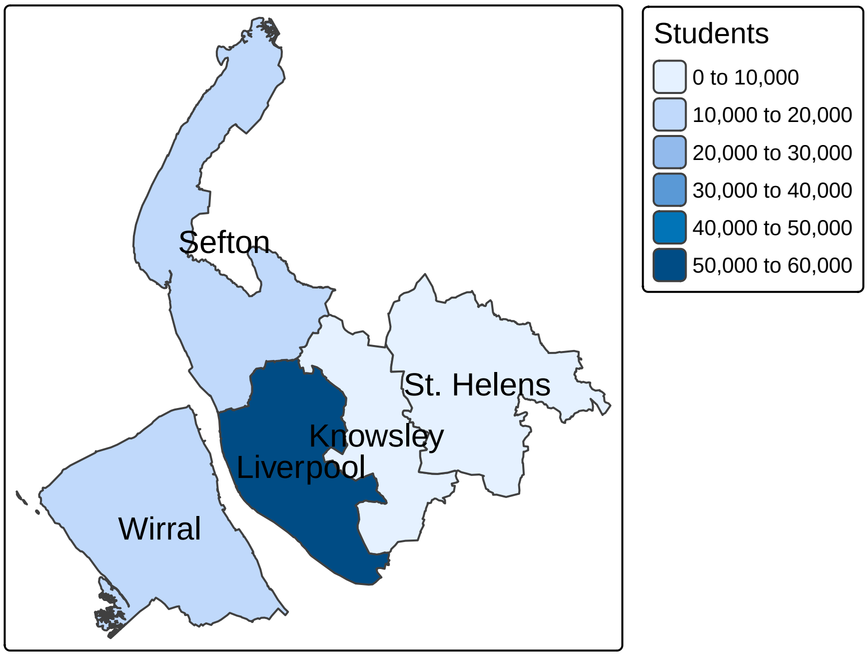


You might assume the quick mapping function `qtm()` can achieve the same result, but `tmap` provides far more flexibility when it comes to aesthetic customization. The easiest way to illustrate `tmap` is through some examples.

Let's start with a simple choropleth map, by using `tmap` to show the distribution of a continuous variable in different elements of the spatial data (here are the data Merseyside districts are polygons).

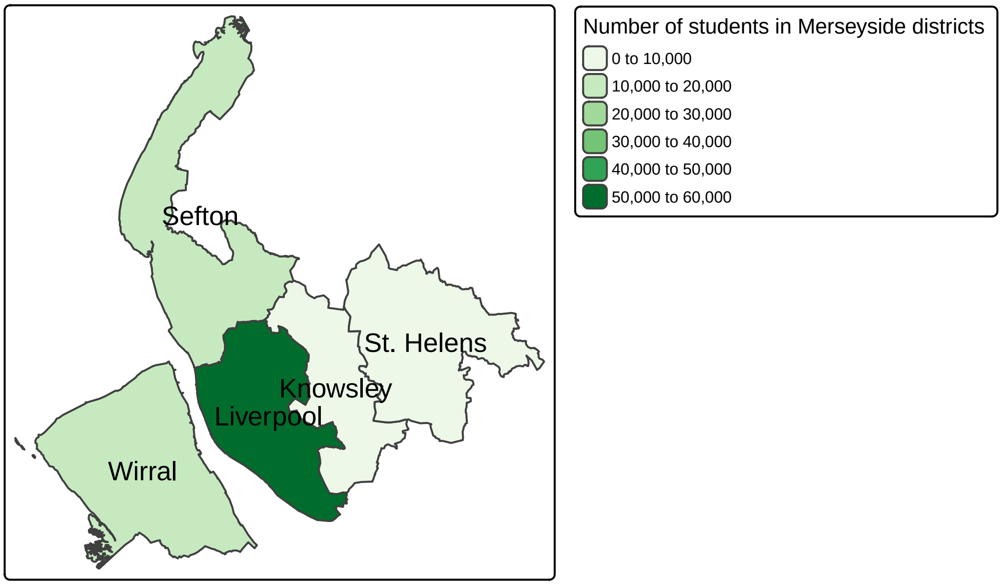
The code below maps '`Students`' as in the Merseyside districts, and shows the district names of each polygon from '`LAD25NM`' columns. The map below indicates that Liverpool has the highest number of full-time students while Knowsley and St.Helens have the least.

```
tm_shape(merseyside) +
  tm_polygons(fill = "Students") +    # Variable to map
  tm_text("LAD25NM")                  # Variable to label
```



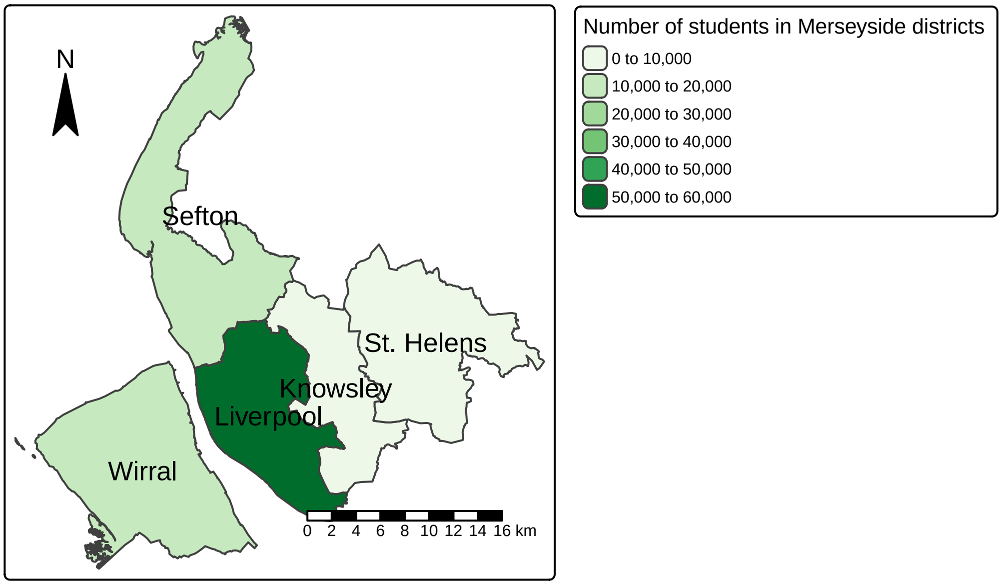
By default tmap picks a shading scheme, the class breaks and places a legend somewhere. All of these can be changed. The code below allocates the tmap plot to `map1` (Map 1), change the legend title as “Number of students in Merseyside districts”, and then prints it:

```
map1  = tm_shape(merseyside) +
  tm_polygons(fill="Students",
              fill.scale = tm_scale(values = "Greens"), # change palette to greens
              fill.legend = tm_legend(title = "Number of students in Merseyside districts")
              ) +# Legend title
  tm_text("LAD25NM",size=0.8) #size down the label slightly
map1
```



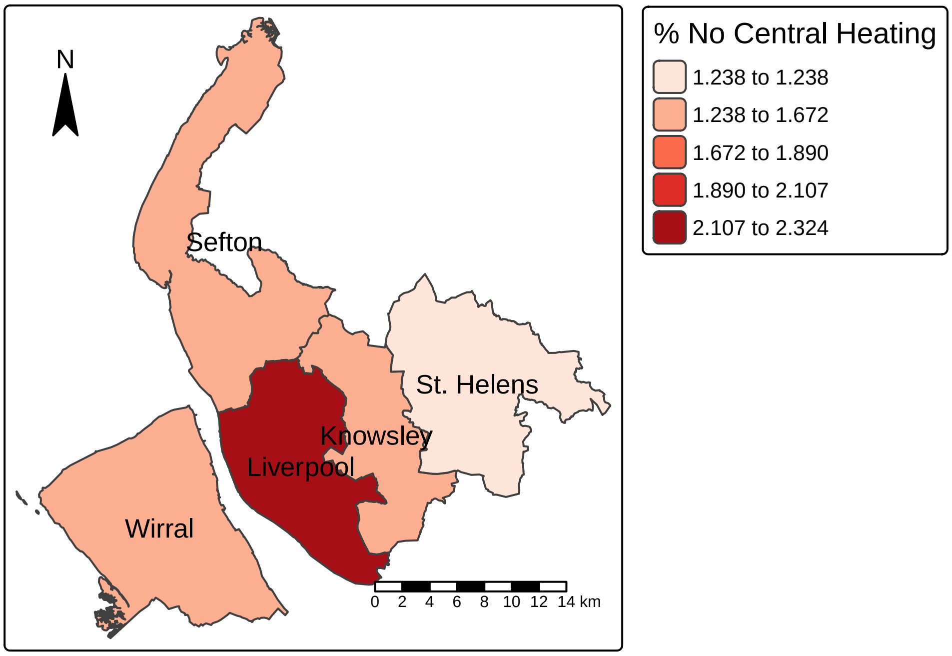
And of course many other elements included either by running the code snippet defining `map1` above with additional lines or by simply adding them as in the code below:

```
map1 +
  tm_scalebar(position = c("right", "bottom")) +
  tm_compass(position = c("left", "top"))  # Use "top", "center", or "bottom"
```



We can also create new variable to the dataset and then map it. The below code chunk first creates a new column, "`NoCentralHeating_rate`", by dividing the number of households without access to central heating by the total number of households in each district; it then uses `tmap` to make a map of the proportion of households without central heating across districts in Merseyside:

```
merseyside$NoCentralHeating_rate = merseyside$No_central_heating / merseyside$Households * 10

map2 = tm_shape(merseyside) +
  tm_polygons(fill="NoCentralHeating_rate",
              fill.scale = tm_scale(values = "Reds", style = "jenks"), #use jenks classifica
              fill.legend = tm_legend(title = "% No Central Heating")) +
  tm_text("LAD25NM",size=0.8) +
  tm_scalebar(position = c("right", "bottom")) +  # Add a scale bar at the top-right corner
  tm_compass(position = c("left", "top"))  # Add a compass rose at the top-right corner
map2
```



1.3.2 Merseyside neighbourhoods

Now let's read in the neighbourhood-level datasets, which include a `.csv` file of local statistics and the corresponding geographical boundaries.

```
lsoa_df <- read.csv("merseyside_lsoa.csv")
lsoa_sf <- st_read("LSOA_boundaries.gpkg")
```

```
Reading layer `merseyside_LSOA' from data source
  `C:\Users\ziye\Documents\GitHub\quant\labs\LSOA_boundaries.gpkg'
  using driver `GPKG'
Simple feature collection with 923 features and 4 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: -3.200368 ymin: 53.2963 xmax: -2.576743 ymax: 53.6982
Geodetic CRS:  WGS 84
```

First, we take a look at the `.csv` dataset, which as been read into R as `lsoa_df`:

```
View(lsoa_df)
```

or check the structure of the dataset:

```
str(lsoa_df)
```

```
'data.frame':	923 obs. of  11 variables:
 $ LSOA21CD          : chr  "E01006416" "E01006418" "E01006434" "E01006435" ...
 $ Population        : int  1520 1315 1519 1524 1150 1654 1450 1581 1421 1373 ...
 $ Households        : int  678 567 652 663 490 695 592 622 809 618 ...
 $ Working_population: int  588 547 660 581 546 766 558 570 524 481 ...
 $ Students          : int  59 64 69 82 51 66 65 89 52 72 ...
 $ Unemployed        : num  3.57 2.74 5.11 3.43 1.62 ...
 $ Age_over_65       : num  14.8 17.5 11.7 19.9 26.3 ...
 $ Disability        : num  27.1 30 23.4 29 20.5 ...
 $ No_central_heating: int  16 14 16 7 3 8 9 9 12 9 ...
 $ Overcrowding      : int  24 24 35 28 13 21 29 23 31 22 ...
 $ Working_from_home : int  91 84 102 96 165 171 101 64 66 48 ...
```

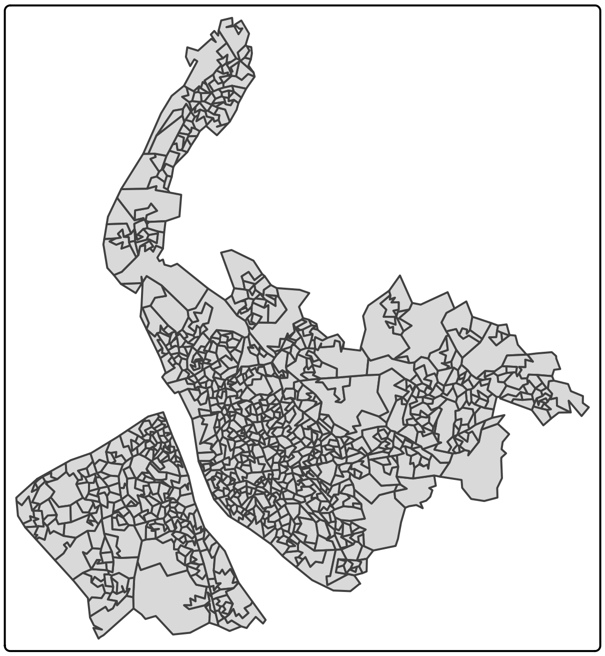
So now, you know how many LSOAs in Merseyside? Yes, there are 923 LSOAs. As we introduced in the Week 1 lecture, LSOA means Super Output Area Lower Area and is commonly used in the Census statistics. Each LSOA represents 1,000 to 3,000 people or 400 to 1,200 households in England and Wales.

```
dim(lsoa_df)
```

```
[1] 923  11
```

Use the quick mapping function `qtm()` to quickly inspect the geographical boundary dataset `lsoa_sf` .

```
qtm(lsoa_sf)
```



check how many LSOAs in the boundary dataset - there should also be 923.

```r
nrow(lsoa_sf)
```

```
[1] 923
```

To familiarise yourself with the structures of both datasets, we can use the **names()** command

```r
names(lsoa_df)
```

```
 [1] "LSOA21CD"           "Population"         "Households"
 [4] "Working_population" "Students"           "Unemployed"
 [7] "Age_over_65"        "Disability"         "No_central_heating"
[10] "Overcrowding"       "Working_from_home"
```

```r
names(lsoa_sf)
```

```
[1] "LSOA21CD" "LSOA21NM" "LAD23CD"  "LAD23NM"  "geom"
```

You may find that both dataset are recorded at the LSOA level, with `LSOA21CD` as the key column. As we did with the district-level dataset, we can use `left_join()` to join these two dataset by their sharing field - `LSOA21CD`:

```
lsoa <- left_join(lsoa_sf,lsoa_df,by="LSOA21CD")
```

Now let's check the columns of new dataframe `lsoa`:

```
names(lsoa)
```

```
 [1] "LSOA21CD"           "LSOA21NM"           "LAD23CD"
 [4] "LAD23NM"            "Population"         "Households"
 [7] "Working_population" "Students"           "Unemployed"
[10] "Age_over_65"        "Disability"         "No_central_heating"
[13] "Overcrowding"       "Working_from_home"  "geom"
```

Or open a new tab to view the newly created dataset `lsoa` by

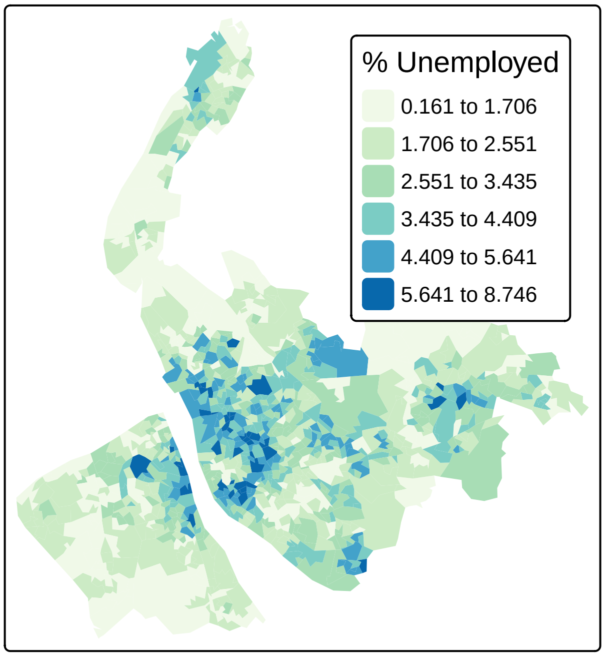
```
View(lsoa)
```

We can see that some columns contain counts, such as the number of residential population, number of households, number of working population, and number of students. Other columns are expressed as percentages, including unemployment, population aged 65 and over, disability, households without central heating, overcrowded households, and people working from home.

1.3.2.1 Making maps across LSOAs in Merseyside

Using the `Unemployed` column, we can create a map of the unemployment rate across neighbourhoods in Merseyside. Instead of using the default equal-interval breaks, this time we will use a jenks classification with six categories.

```
map3 = tm_shape(lsoa) +
  tm_fill(
    fill = "Unemployed",
    fill.scale = tm_scale(values = "GnBu",
                          style = "jenks",
                          n = 6), #use jenks classification of 6 categories
    fill.legend = tm_legend(title = "% Unemployed")
    ) +
  tm_layout(legend.position = c("right", "top"))
map3
```



The above code uses `tm_layout(legend.position = c("right", "top"))` to move the legend inside the map frame, positioning it at the right-top corner.

Replace `tm_fill()` to `tm_polygons()` to see how the map changes?

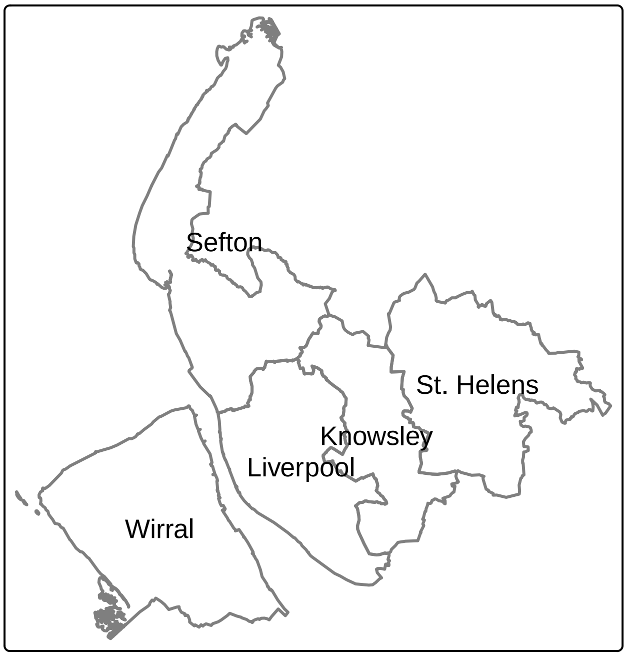
`tm_polygons()` is a condense version of `tm_fill() + tm_border()`. Here if you want show all the LSOA borders, use `tm_polygons()` instead of `tm_fill()`.

1.3.2.2 Overlapping tmap objects

`tmap` also supports adding or overlaying other data, such as boundaries. Because these are additional spatial data layers, they needs to be added with `tm_shape()` followed by the usual function.

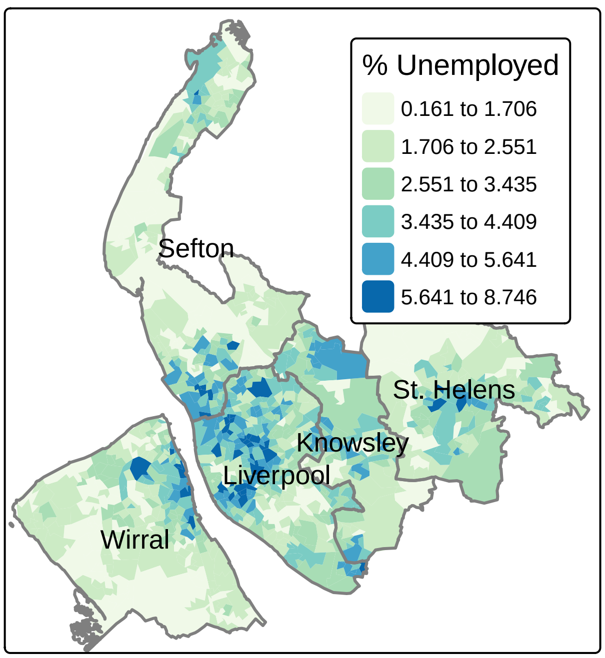
Remember we use the code chunk to make the district boundaries of Merseyside? This time let's change the aesthetic by using grey color as the border color and increase the line width, then we save it also as a `tmap` object called map_district:

```
map_district = tm_shape(merseyside) +
  tm_borders(col = "grey50",lwd=1.5) + #border color as grey, line width as 1.5
  tm_text("LAD25NM",size = 0.8)
map_district
```



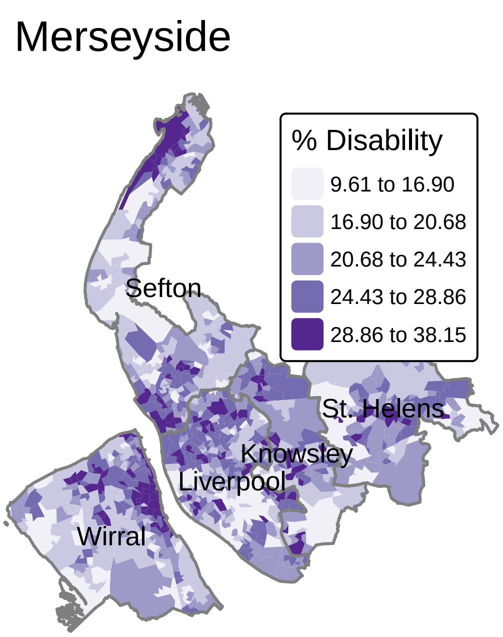
To display both **tmap** layers together, we can proceed as follows:

```
map3 + map_district
```



Now, let's move to making some more maps - this time showing the proportion of disability across neighbourhoods in Merseyside. Referring back to the columns in `lsoa`, this time we use the `Disability` variable. The code below applies a jenks classification and use a different color palette `Purples`. Also the map frame is removed by `frame = FALSE`.

```
tm_shape(lsoa) +
  tm_fill(fill = "Disability",
          fill.scale = tm_scale(values="Purples",
                                style = "jenks",
                                n=5),
          fill.legend = tm_legend(title = "% Disability")
          ) +
  tm_layout(main.title = "Merseyside",#add a main title
            legend.position = c("right", "top"),
            frame = FALSE)+
  map_district
```



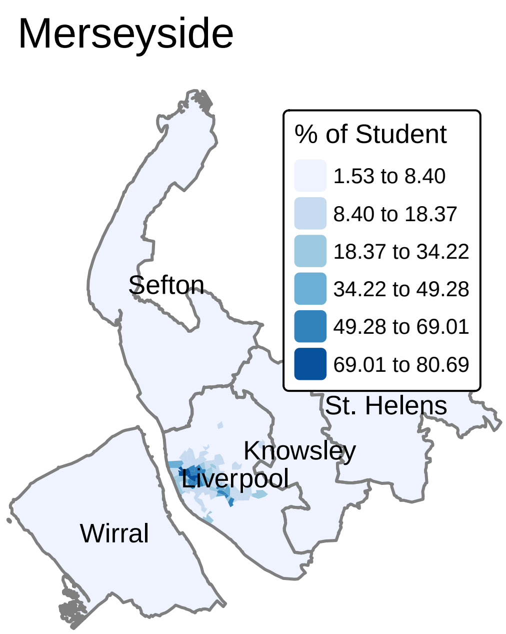
1.3.2.3 Create new variables to make maps

Similarly, we can make maps from new columns we made ourselves. For example, we can calculate the percentage of student by adding a new column to the dataframe:

```
lsoa$student.perc = lsoa$Students / lsoa$Population * 100
```

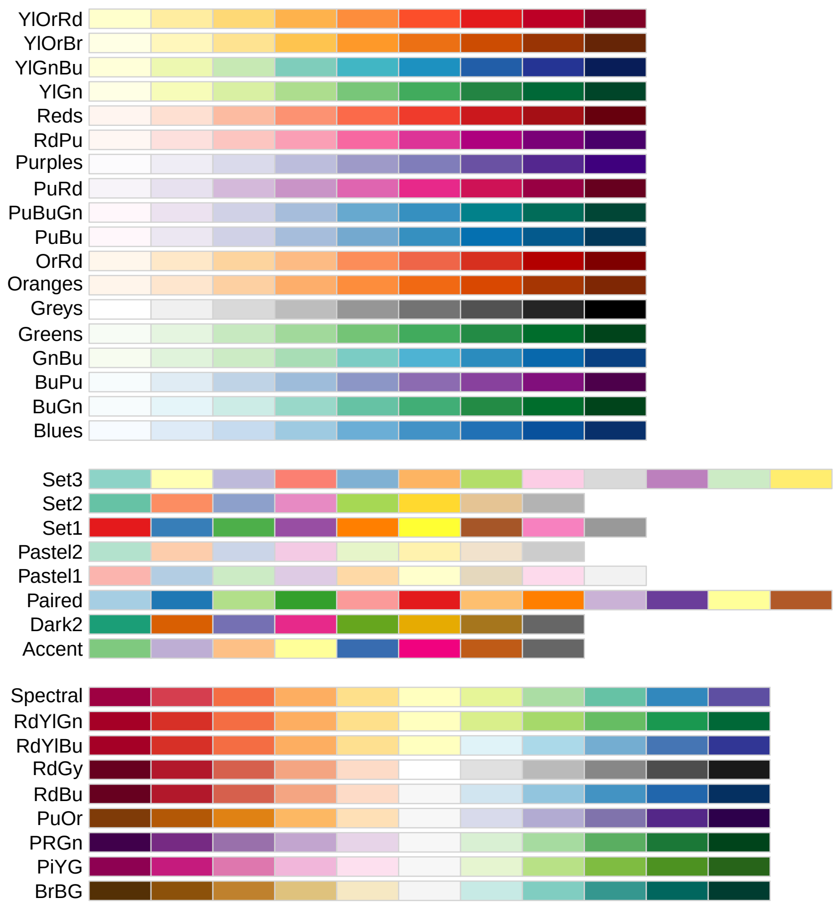
To make a map to visualisation the spatial distribution of student percentage. The code below uses `n=6` to increase the classification categories to 6 rather than default 5.

```
tm_shape(lsoa) +
  tm_fill(fill = "student.perc",
          fill.scale = tm_scale(values="Blues",
                                style = "jenks",
                                n=6),
          fill.legend = tm_legend(title = "% of Student",
                                  title.size = 0.8) #legend title change smaller font
          ) +
  tm_layout(main.title = "Merseyside",
            frame = FALSE,
            legend.position = c("right", "top"))+
  map_district
```



To change the palette, RColorBrewer provides different palette choices:

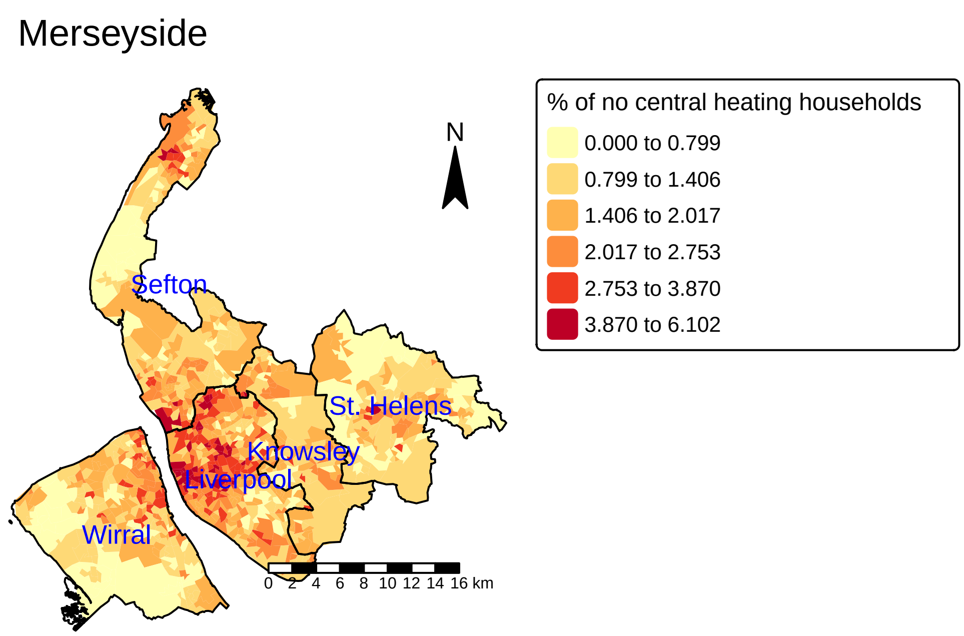
```
RColorBrewer::display.brewer.all()
```



1.3.2.4 In a nutshell

If we want to make a map to show the rate of no central heating households in all the neighbourhoods in Merseyside, we need to create a new variable `no.central.heating.perc` as the result of dividing households without central heating by total households in each LSOA. The code below combines all the cartographic elements together:

```
lsoa$no.central.heating.perc = lsoa$No_central_heating / lsoa$Households * 100

tm_shape(lsoa) +
  tm_fill(fill = "no.central.heating.perc",
          fill.scale = tm_scale(values="YlOrRd",
                                style = "jenks",n=6),
          fill.legend = tm_legend(title = "% of no central heating households",
                                  title.size = 0.8)
        ) +
  tm_layout(main.title = "Merseyside",
            main.title.size=1.2,
            frame = FALSE) +
  tm_compass(position = c("right", "top")) +
  tm_scalebar(position = c("right", "bottom")) +
  tm_shape(merseyside) +              # Add another spatial layer (Merseyside boundary)
  tm_borders(col = "black", lwd = 1) +  # Draw the boundaries with black lines of width 1
  tm_text("LAD25NM",col = "blue",size = 0.8)
```



1.4 Summary

The aim of this session has been to familiarise you with the R environment if you have not used R before. If you have but not for a while, then hopefully this has acted as a refresher. Some key things to take away are:

- R is a learning curve, and like driving the more your practice the better you become.
- Your job is to try to **understand** what the code is doing and **not** to remember the code.
- To help with this, you should add your own comments to the script to help you understand what is going on when you return to them. Comments are prefaced by a hash (`#`) that is ignored by R.
- Always set your working directory to the sub-folder containing your R script.
- Always run your code from an R script… **always**!

1.4.1 References

Brunsdon, Chris, and Lex Comber. 2018. *An Introduction to r for Spatial Analysis and Mapping (2e)*. Sage.

Comber, Lex, and Chris Brunsdon. 2021. *Geographical Data Science and Spatial Data Analysis: An Introduction in r*. Sage.

Harris, Richard. 2016. *Quantitative Geography: The Basics*. Sage.

Other good on-line *get started in R* guides include:

- The Owen guide (only up to page 28) : https://cran.r-project.org/doc/contrib/Owen-TheRGuide.pdf
- An Introduction to R - https://cran.r-project.org/doc/contrib/Lam-IntroductionToR_LHL.pdf
- R for beginners https://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf

1.5 Formative Tasks

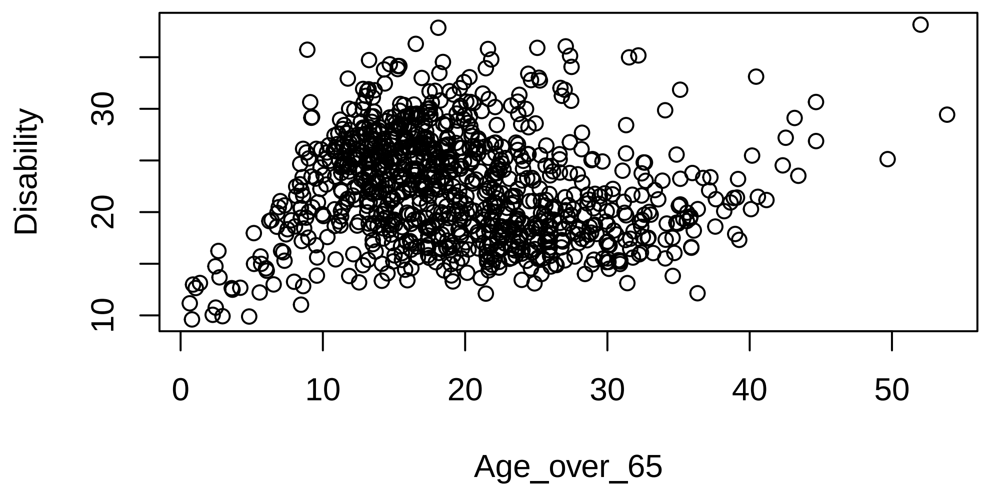
Task 1 From the district level dataset "merseyside.csv", extract household information for the Liverool and Wirral districts. The variables to be included are "Households", "No_central_heating" and "Overcrowding".

```r
df <- read.csv("merseyside.csv")
df[c(2,5),c("District","Households","No_central_heating","Overcrowding")]
```

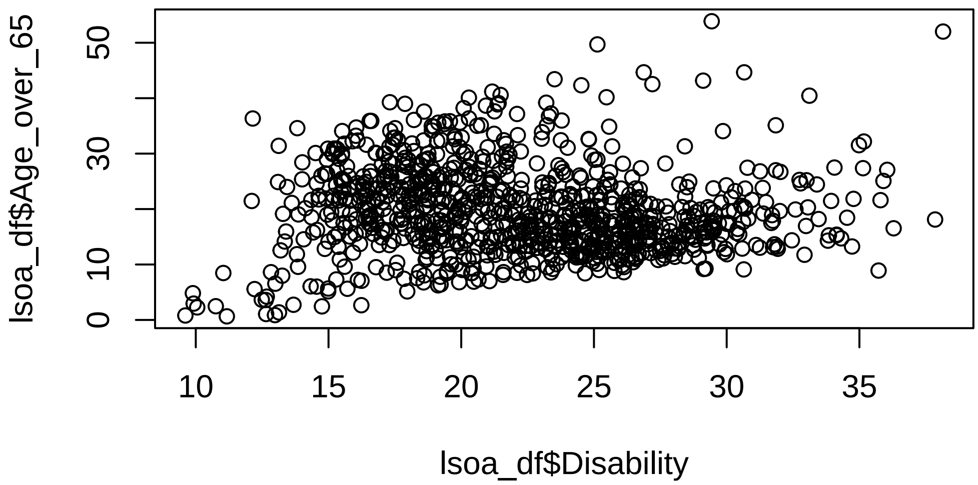
```
   District Households No_central_heating Overcrowding
2 Liverpool     207491               4822         7352
5    Wirral     143253               2125         2355
```

Task 2 Use the dataset "merseyside_lsoa.csv", plot the `Disability` against `Age_over_65` from the data frame.

```r
lsoa_df <- read.csv("merseyside_lsoa.csv")
plot(Disability~Age_over_65, data = lsoa_df)
```




```r
# or
plot(lsoa_df$Disability, lsoa_df$Age_over_65)
```



Task 3 Use the district level dataset, how many households in total in Merseyside?

```
df <- read.csv("merseyside.csv")
sum(df$Households)
```

```
[1] 620903
```

Task 4 Use the district level dataset, what is the overall proportion of the ageing population (age over 65) in Merseyside?

```
sum(df$Age_over_65)/sum(df$Population)
```

```
[1] 0.1920626
```

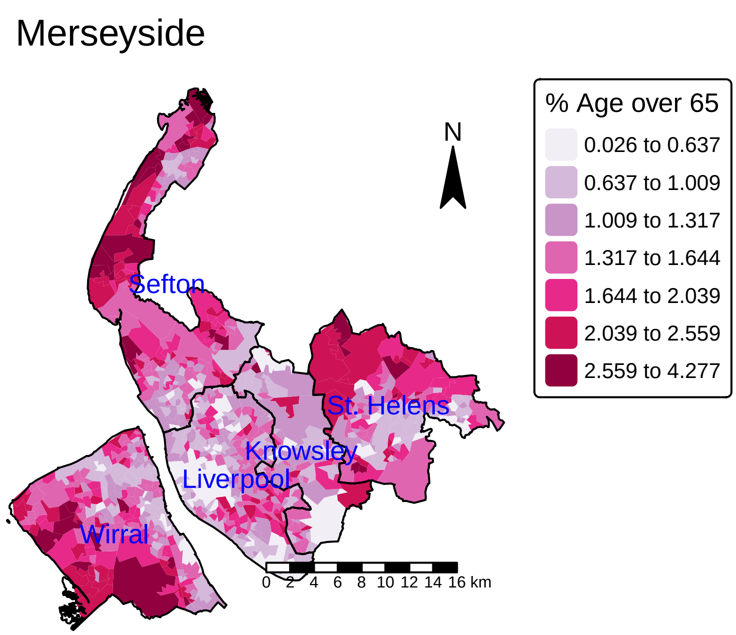
Task 5 Use the LSOA level dataset, what is the average proportion of the ageing population (age over 65) across all the neighbourhoods of Merseyside?

```
df$ageing_rate = df$Age_over_65 / df$Population * 100
mean(df$ageing_rate)
```

```
[1] 19.59824
```

Task 6 Create a map showing the spatial distribution of the proportion of ageing population (age over 65) over LSOAs in Merseyside? (use Jenks classification of 7 categories).

```
lsoa$ageing_rate = lsoa$Age_over_65 / lsoa$Population * 100

tm_shape(lsoa) +
  tm_fill(fill = "ageing_rate",
          fill.scale = tm_scale(values="PuRd",style = "jenks",n=7),
          fill.legend = tm_legend(title = "% Age over 65", title.size = 0.8)
        ) +
  tm_layout(main.title = "Merseyside",
            main.title.size=1.2,
            frame = FALSE) +
  tm_compass(position = c("right", "top")) +
  tm_scalebar(position = c("right", "bottom")) +
  tm_shape(merseyside) +                # Add another spatial layer (Merseyside boundary)
  tm_borders(col = "black", lwd = 1) +  # Draw the boundaries with black lines of width 1
  tm_text("LAD25NM",col = "blue",size = 0.8)
```



2 Lab: Exploratory Data Analysis - UK Election

2.1 Overview

This week's practical session will draw upon the UK 2024 constituency election dataset. We will revisit some libraries and functions we have used last week, but also learn to do our first exploratory data analysis by foundmental R coding. We will do:

- Loading and examining tabular data (like spreadsheets) and geographical data into R (same as Week 1)
- Exploratory Data Analysis (or EDA) of numeric variables and categorical variables
- Using histogram, boxplot, barplot to understand the distribution of variables
- Variable interactions, particularly cross-tabulation and between-group comparison

You may wish to recap this week's lecture: Lecture 02.pptx

2.2 Clear the decks

- For this Week 2 session, create a sub-folder called `Week2` in your `ENVS162` folder on your M-Drive.
- Open RStudio
- Open a new R Script for your Week 2 work, rename it as Week2.R and save it in your newly creately Week 2 folder, under M drive -> ENVS162 folder. This is exactly the step we did in Week 1, and we will do this every week to Week 5.
- Check whether there is any previous left dataframe in your RStudio in the upper-right side Environment pane. You can always use the  to clear all the dataframes in your environment and make it all clean. For the same aim, you can run the below code:

```
rm(list = ls())
```

This command will clear RStudio's memory, removing any data objects that you have previously created.

2.3 Open libraries

In Week 1 we have installed essential R package **tidyverse**, **sf**, and **tmap**. Remember if any package has been installed, then we don't need to re-install them. Instead, we use library() command to import and use them.

As ever, when you start a new session in RStudio, you need to load the packages you wish to use into memory. In addition to the **tidyverse**, **gdslStats** and **sf** packages we used last week.

```
library(tidyverse)
library(sf)
library(tmap)
```

If R returns Error: there is no package called '***'. Then it means that the package '***' has not been installed in the PC you current use. Therefore you need to install them first. Switch back to Week 1 instruction - Getting set up with RStudio - Your first R code - Package part to refresh yourself how to do this.

2.4 Parliamentary Constituency Data

2.4.1 Load the dataset

In 2024 the UK held a general election. Download the file ***uk_constituencies_2024.csv*** from our Canvas module page Week 2. Read in the dataset exactly in the same way as we did in Week 1 - insert the below code line and run it:

```
pc_data <- read.csv("uk_constituencies_2024.csv",stringsAsFactors = TRUE)
```

The datasheet captures a range of information relating to this election, and to the nature of each parliamentary constituency. By using `read.csv()` command, we use R to store the dataset in a dataframe called *pc_data* (short for Parliamentary Constituency data).

The *pc_data* dataset should appear in your RStudio Environment Pane on the upper right part, indicating that it has now been loaded into memory, and is available for analysis.

For your information only, the *pc_data* dataset has been assembled by combining information from the following sources.

- *HoC-GE2024-results-by-constituency.xlsx*

  Source: Cracknell et al (2024) General election 2024 results, *Research Briefing*, House of Commons Library. https://commonslibrary.parliament.uk/research-briefings/cbp-10009/

- *Demographic-data-for-new-parliamentary-constituencies-May-2024.xlsx*

  Source: House of Commons Demographic data for Constituencies https://commonslibrary.parliament.uk/d for-new-parliamentary-constituencies/

- *NatCen Constituency Data_20 June 2024.xlsx*

  Source: National Centre for Social Research (2024) Parliamentary constituency look-up, https://natcen.ac.uk/constituency-look-up (date accessed: 07-02-2025)

- *nomis_2025_01_14_101749.xlsx*

  Source: Income data from Annual Survey of Hours and Earnings (ASHE), collecgted by the Office for National Statistics. https://www.nomisweb.co.uk/datasets/asher

2.4.2 Familiar with the dataset and variable types

In the *pc_data* dataset each row represents a different UK Parliamentary Constituency. Use the `View()` command to familiarise yourself with the variables contained in the dataset.

```
View(pc_data)
```

Use the `nrow( )` or `dim()` command to find out how many Parliamentary Constituencies (and therefore MPs) there are in the UK.

```
nrow(pc_data)

dim(pc_data)
```

Whenever we try to understand the variables in one dataset, the first question to ask ourselves is: "are they *continuous* or *categorical* ?"

Continuous variables are numeric measures of some quantity, such as a count or percentage or a precise value. E.g. number of valid votes; % of persons unemployed etc.

In contrast, categorical variables simply group observations into categories or ranges. E.g. name of the winning party; age group etc.

Explore the structure of the data table using the `str` function and have examined it by `head` functions.

```
str(pc_data)

head(pc_data)
```

We can see that we have numeric data in integers (`int`) form (these are counts or whole numbers) and continuous (`num`) form, and the character variables (text) have been converted to `factors`. For each of these data types we can generate numeric and visual summaries and we can also see how they interact with each other.

The *pc_data* dataset contains three basic sets of information about each Parliamentary Constituency:

- **constituency identifiers** - *gss_code* and *pc_name*
- **population information** for each constituency, ranging from the total population and number of households through to the % in various categories to information about local house prices, salaries and crime rates
- **2024 election results** ranging from the winning MP and party through to the size of the electorate and vote turnout, and the share of votes received by each party

2.4.3 Exploratory Data Analysis (EDA)

2.4.3.1 Numeric variable

You can use the *pc_data* dataset to extract some headline results from the 2024 General Election. Starting with the simplest case, the distribution of a single numeric variable whether continuous or count based can be examined numerically using the `summary` function.

```
#valid votes in constituency
summary(pc_data$valid_votes)
```

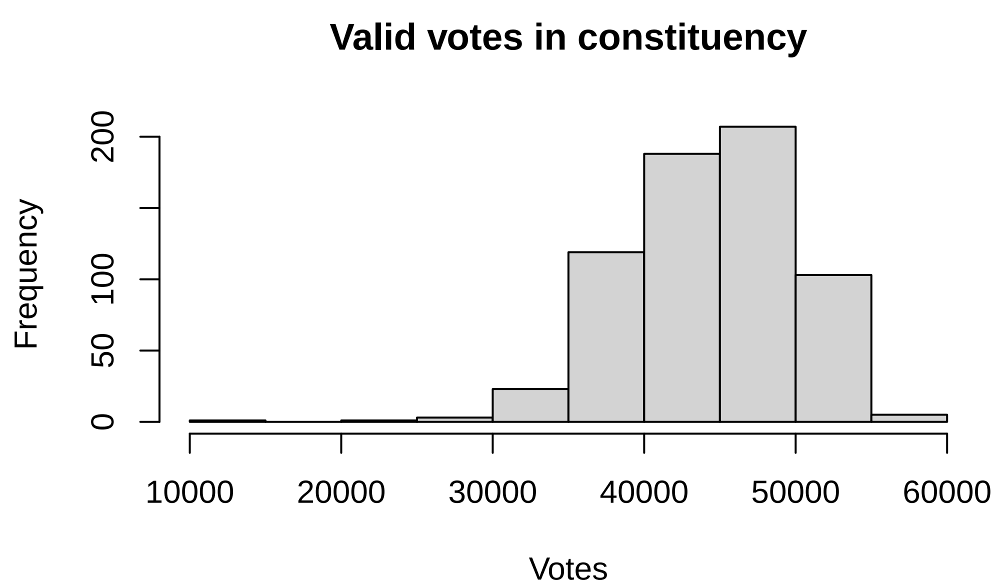
```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  13528   40397   44628   44322   48607   57744 
```

```
# percentage of White British in constituency
summary(pc_data$pct_White_British)
```

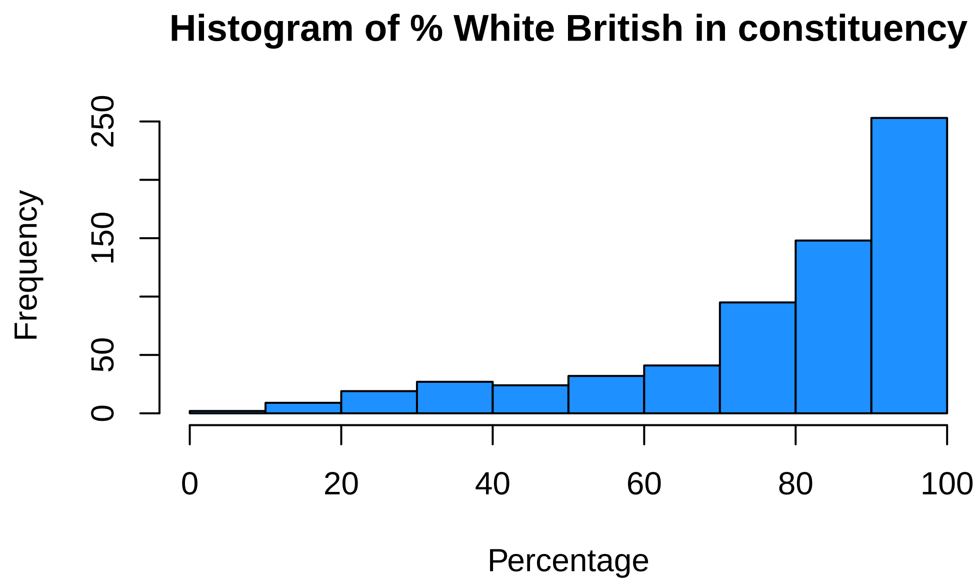
```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  8.937  71.534  86.208  78.006  92.843  98.664 
```

A visual approach is more intuitive. The code below plots histograms of the two variables, using the `hist` function. The logic under-pinning a histogram is that a continuous variable (in this case *valid_votes* and *pct_White_British*) is temporarily regrouped into categories, using an equal-interval approach. The number of observations (in this case, constituencies) that fall into each equal-interval category then determine the height of each column in the histogram.The comments after each line shall inform you what the `main` and `xlab` in the function mean:

```
# histograms
hist(pc_data$valid_votes,
     main = "Valid votes in constituency", #change chart title
     xlab = "Votes")     #change x axis label
```



```r
hist(pc_data$pct_White_British,
     main = "Histogram of % White British in constituency",
     xlab = "Percentage",
     col = "dodgerblue")   #change bar color to dodgerblue
```



You may noticed that `valid_votes` has a relatively normal, bell-shaped distribution whereas the `pct_White_British` variable is left skewed (negatively) distribution.

We can examine the how these distributions relate to central tendencies (mean, median) and spread, using standard deviations for means and the IQR (Inter-Quartile Range) for medians.

From the numeric summary above, the mean of `pc_data$valid_votes` is 44,322. We can determine the spread around this value by calculating the standard deviation for our sample , as is returned by the `sd` function in R:

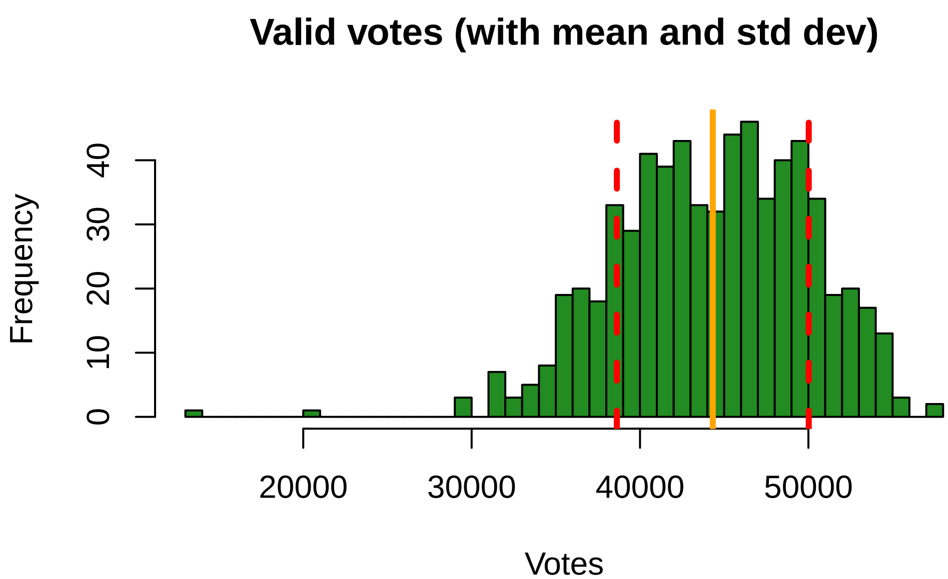
```r
sd(pc_data$valid_votes)
```

```
[1] 5697.665
```

For a normal distribution, about 68% of observations lie within 1 standard deviation of the mean, and about 95% lie within 2 standard deviations of the mean. So this suggest that 68% of the valid votes are within 5,698 votes of the mean of 44,322 votes, i.e. 38,624 (44322-5698) and 50020 (44322+5698).

We can augment the histogram of the `valid_votes` variable with this information, requesting R to increase the intervals to 50, and using the `abline` function to add lines to create the figure below:

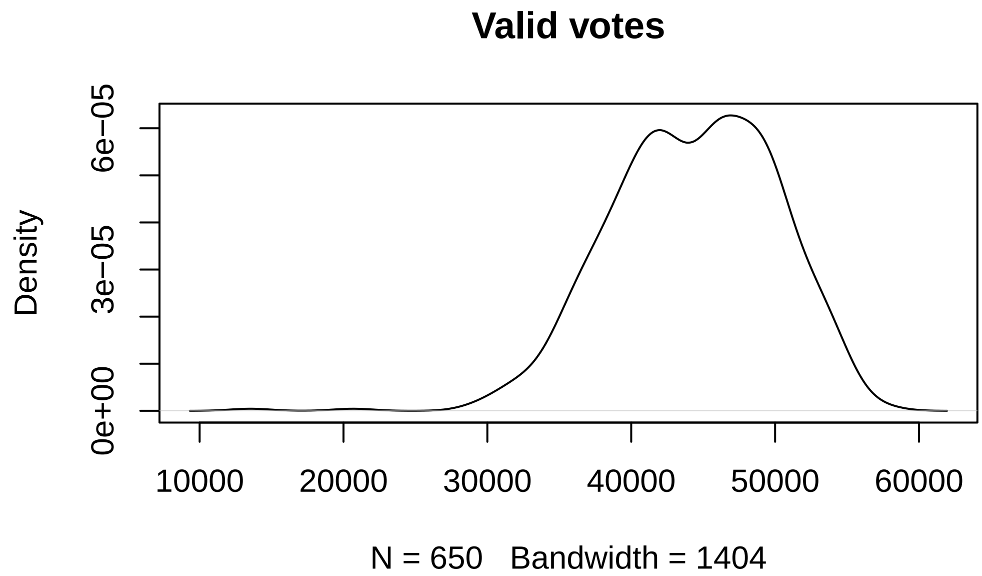
```
# histogram
hist(pc_data$valid_votes, col="forestgreen", main="Valid votes (with mean and std dev)",
     breaks = 50, xlab="Votes")
# calculate and add the mean
mean_val = mean(pc_data$valid_votes)
abline(v = mean_val, col = "orange", lwd = 3)
# calculate and add the standard deviation lines around the mean
sdev = sd(pc_data$valid_votes)
# minus 1 sd
abline(v = mean_val-sdev, col = "red", lwd = 3, lty = 2)
# plus 1 sd
abline(v = mean_val+sdev, col = "red", lwd = 3, lty = 2)
```



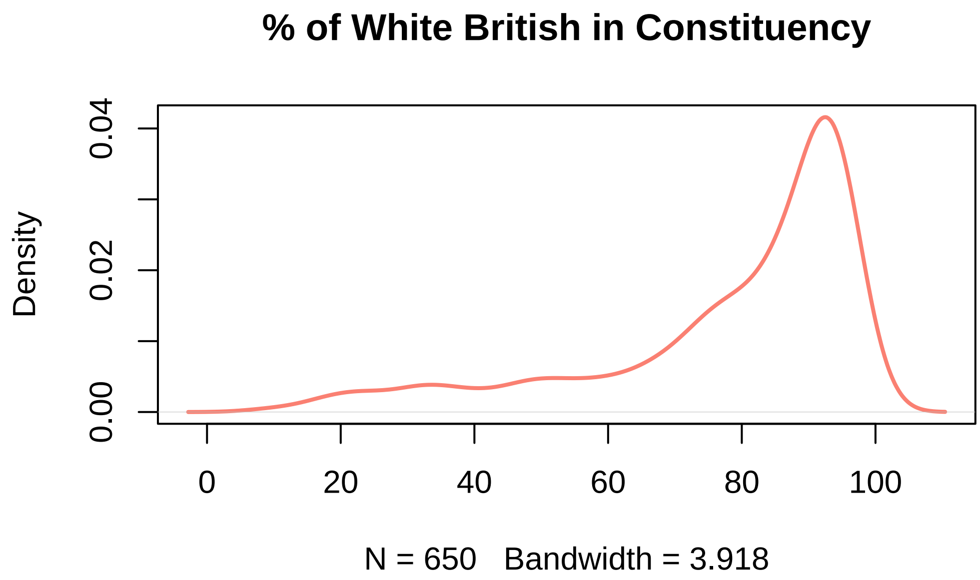
This histogram show the variable `valid_votes` with the orange solid line as the mean, and dashed red line as the standard deviation. Note that in the call to `abline` above, note the specification of different line types (lty) and line widths (`lwd`). Later on, you could explore these as described here.

We can also use density curve to present the variable distribution:

```r
plot(density(pc_data$valid_votes),
     main = "Valid votes")
```

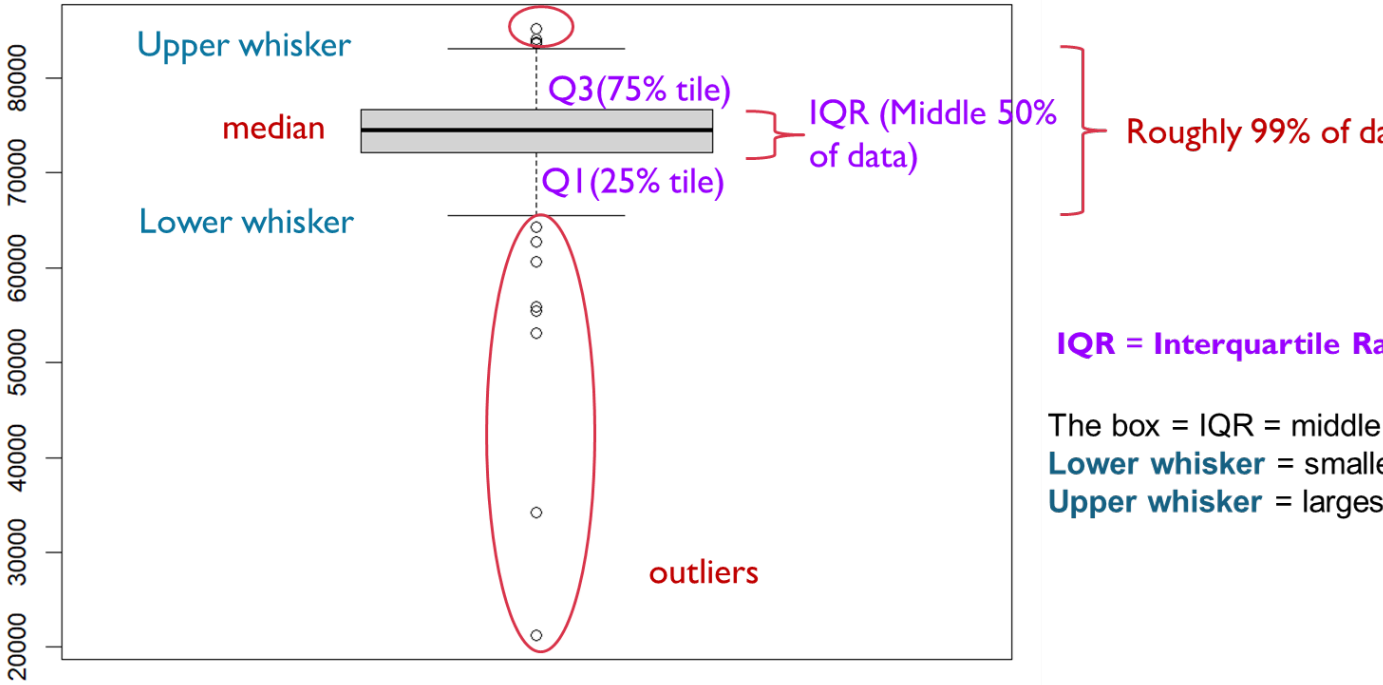



```r
plot(density(pc_data$pct_White_British),
     main = "% of White British in Constituency",
     col="salmon",
     lwd=2)
```



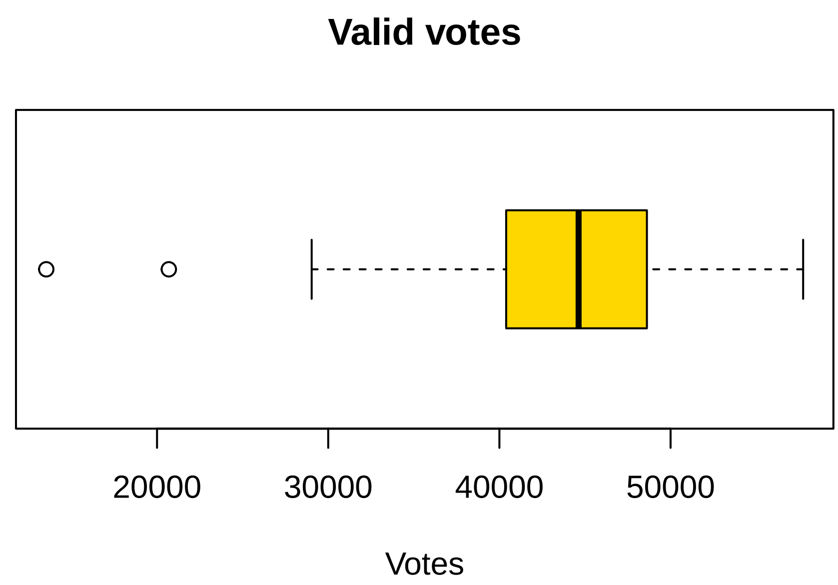
Boxplots show the same information but here we can see a bit more of the nature of the distribution.

Recap Week 2 lecture, the dark line shows the median, the box represents the interquartile range (from the 1st to the 3rd quartile), the whiskers extend to the most extreme non-outlying values, and the dots indicate outliers.

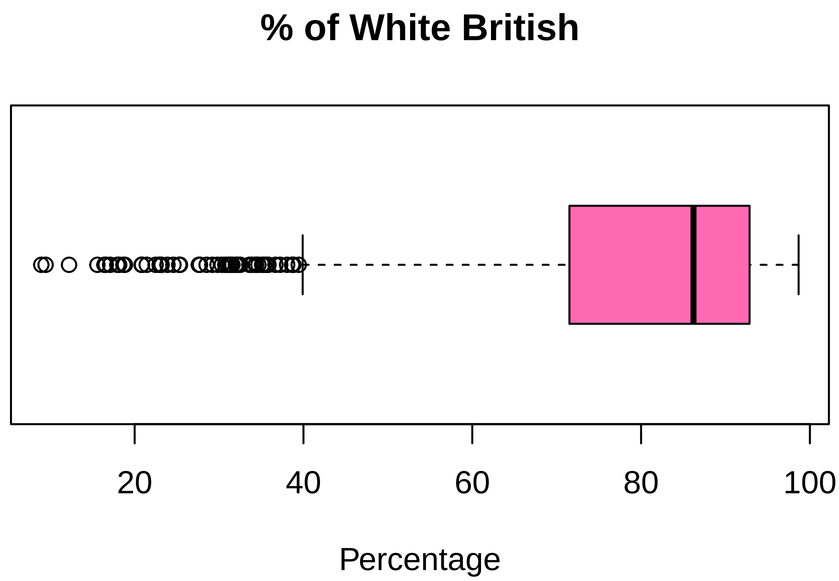


Now let's check out the boxplots of these two variables:

```
boxplot(pc_data$valid_votes,
        horizontal=TRUE,
        main = "Valid votes",
        xlab='Votes',
        col = "gold")
```




```r
boxplot(pc_data$pct_White_British,
        horizontal=TRUE,
        main = "% of White British",
        xlab='Percentage',
        col = "hotpink")
```



1. Is the median line in the centre of the box? If the median line is not in the middle of the box, it means the data are skewed, with greater spread on one side of the median.
    - Median closer to the bottom of the box -> right-skewed (positively skewed) -> some big values pulling up
    - Median closer to the top of the box -> left-skewed (negatively skewed) -> some small values dragging down
2. Are the whiskers the same length? This indicates skewness in the distribution. The data have a longer tail on the side of the longer whisker, meaning values are more spread out in that direction.
3. Are there any outliers? Who has more? Outliers clustered on one side indicate that extreme observations occur predominantly in one tail of the distribution. More outliers indicates the variable hs more extreme or unusual values, possibly a heavier-tailed distribution. But this doesn't mean the variable is bad or more variable overall.
4. The size of the boxes? The box represents the interquartile range (IQR), which is the middle 50% of the data as we call them typical. A larger box indicates greater variability in the middle 50% of the variable; a smaller box suggests that values are more tightly clustered around the median.

Now, you should be able to compare how the two kinds of distribution are shown in the the boxplots with a trained eye: The `pct_White_British` variable is left-skewed, indicating that more values are spread towards lower percentages of White British in the constituency. The box is also larger than that of `valid_votes`, which suggests greater variability in the central 50% of the data. In addition, the longer lower whisker and the presence of more outliers on the lower end further reinforce the left-skewed distribution.

In summary, numeric variable distributions, of counts and continuous data, should be investigated as an initial step in any data analysis. There are a number of metrics and graphical functions (tools) for doing this including `summary()`, `hist()` `plot(density())` and `boxplot()`.

2.4.3.2 Categorical variable

Some of the character variables could be considered as **categorical**, representing a grouping or classification of some kind, as described above. In these cases we are interested in the count or frequency of each class in the classification, which we can examine numerically or graphically.

The simplest way to examine classes is to put them into a table of counts. The `table` function is very useful and in the code below it is applied to one of the categorical variables in the survey data:

So firstly we use the `table()` command to find the number of MPs elected to each Party. [Hint: use the *first_party* variable].

```
table(pc_data$first_party)
```

```

       Alliance    Conservative             DUP           Green     Independent 
              1             121               5               4               6 
         Labour        Lib Dems     Plaid Cymru          Reform            SDLP 
            411              72               4               5               2 
      Sinn Fein             SNP         Speaker Ulster Unionist  Unionist Voice 
              7               9               1               1               1 
```

These can be made a bit more tabular in format with the `data.frame` function, which takes the `table` operation as its input:

```
data.frame(table(pc_data$first_party))
```

```
              Var1 Freq
1         Alliance    1
2     Conservative  121
3              DUP    5
```

```
4             Green    4
5       Independent    6
6            Labour  411
7          Lib Dems   72
8       Plaid Cymru    4
9            Reform    5
10             SDLP    2
11        Sinn Fein    7
12              SNP    9
13          Speaker    1
14 Ulster Unionist     1
15  Unionist Voice     1
```

However, if we not only care about the count of MPs but also the proportion? Then we can make a good use of our tidyverse library to run the following code line.

```
pc_data %>%
  count(first_party) %>%
  mutate(pct = round(n / sum(n) * 100,1))
```

```
       first_party   n  pct
1         Alliance   1  0.2
2     Conservative 121 18.6
3              DUP   5  0.8
4            Green   4  0.6
5      Independent   6  0.9
6           Labour 411 63.2
7         Lib Dems  72 11.1
8      Plaid Cymru   4  0.6
9           Reform   5  0.8
10            SDLP   2  0.3
11       Sinn Fein   7  1.1
12             SNP   9  1.4
13         Speaker   1  0.2
14 Ulster Unionist   1  0.2
15  Unionist Voice   1  0.2
```

Here you are using two very useful functions in the library `tidyverse`.

First, the `count()` calculate the frequency of different categories in the `first_party` and use a new column `n` to store the frequencies - it actually do the same thing as above code, but better in presenting as a table;

Second, the `mutate()` function to assist use create a new column `pct`and fill in the value by the calculation `pct = n / sum(n) * 100`.

The `%>%` is used to link these two commands: `count()` and `mutate()`. You can insert `%>%` in your R script by using `ctrl + shift + M` for Windows and `Cmd + Shift + M` on Mac.

Therefore, when you run the code, you will see a table showing as three column: first part name, a new column automatically named as `n` by R after the `count()` function, and count of the party MPs, and a new column called `pct` and with values calculated by `n / sum(n) * 100`. We use `round()` function to keep only 1 digits for the `pct`.

We can also improve the code to make a better table presentation. You may find the comment text after each code line would be useful to understand what R has done to the *pc_data*.

```
#Calculate the frequency and percentage of different categories for "first_party"
pc_data %>%
  count(first_party) %>%
  mutate(pct = round(n / sum(n) * 100,1)) %>%
  arrange(desc(n)) %>%   #sort the table by number of MPs from more to less
  setNames(c("First Party", "Number of MPs", "% of MPs"))  #rename table column names
```

```
       First Party Number of MPs % of MPs
1           Labour           411     63.2
2     Conservative           121     18.6
3         Lib Dems            72     11.1
4              SNP             9      1.4
5        Sinn Fein             7      1.1
6      Independent             6      0.9
7              DUP             5      0.8
8           Reform             5      0.8
9            Green             4      0.6
10     Plaid Cymru             4      0.6
11            SDLP             2      0.3
12        Alliance             1      0.2
13         Speaker             1      0.2
14 Ulster Unionist             1      0.2
15  Unionist Voice             1      0.2
```

In this code chuck, we requested four command to the dataframe pc_data, and linked them with `%>%` :

1. `count()` function to summarises the data by counting the number of observations in each group;

2. `mutate()` function to create a new column named "pct" as we did above;
3. `arrange()` function sorts the rows, `desc(n)` function decending the norder of `n`, together they order the category with the highest counts first;
4. `setNames()` function renames the results.

Similarly, we can use the categorical variable `crime_rate` in the `pc_data` dataset to understand the crime status of all the constituencies:

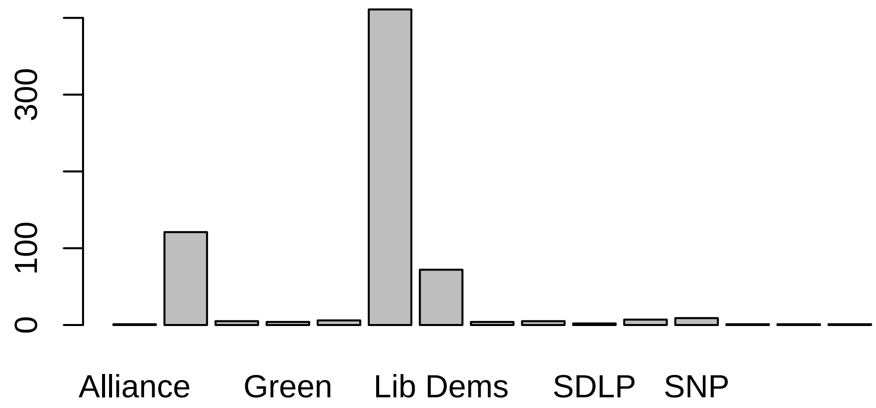
```
#Calculate the frequency and percentage of different categories for "crime_rate"
pc_data %>%
  count(crime_rate) %>%
  mutate(pct = round(n / sum(n)*100, 1))  %>%
  arrange(desc(n)) %>%
  setNames(c("Crime rate", "Number of Constituency", "% of Constituency"))
```

```
   Crime rate Number of Constituency % of Constituency
1 Much higher                    118              18.2
2      Higher                    115              17.7
3     Average                    114              17.5
4       Lower                    114              17.5
5  Much lower                    114              17.5
6        <NA>                     75              11.5
```

What you have learnt from the result table? It seems that there is a quite equally distribution across the five categories of crime rate, although there are 11.5% of the constituency are missing their crime rates.

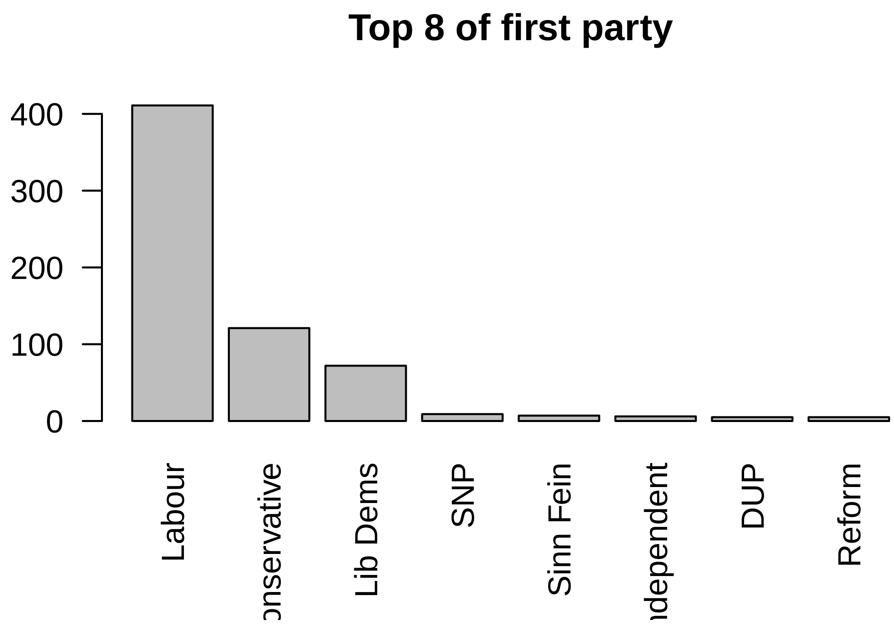
Categorical data can be visualised using bar plots of the tabularised data. The code below does this by creating a table, changing the names of the table and passing that to the `barplot` function:

```
#calculate the frequency of first party and save the result in table 'tab', present tab as a
tab = table(pc_data$first_party)
barplot(tab)
```



It is very simple to get the barplot from the result of table(). But it may need some improvement. As you may noticed, there are many bars have rarely no values, and too crowd x axis labels makes some bar label can't able to show. Therefore, we probably don't need to show all the parties, but only the top 8 in terms of their winning constituencies.

```
#sorted tab by the frequency from highest to lowest, present the top 8 of the sorted tab
tab_sorted <- sort(tab,decreasing = TRUE)
barplot(head(tab_sorted,8),las = 2, main = "Top 8 of first party")
```



In this code chunk, we first use sort() to reorder the tab ranking the parties based on the counts from highest to lowest. You may ask R to show how tab_sorted looks like:

```
#show tab_sorted
tab_sorted
```

```
         Labour    Conservative        Lib Dems             SNP       Sinn Fein
            411             121              72               9               7
    Independent             DUP          Reform           Green    Plaid Cymru
              6               5               5               4               4
           SDLP        Alliance         Speaker Ulster Unionist Unionist Voice
              2               1               1               1               1
```

Then, using `barplot()`, we plot the top 8 rows of `tab_sorted`, with `las = 2` used to rotate the axis labels so they are displayed vertically and remain readable.

2.4.3.3 Categorical to categorical: cross-tabulation

In EDA, it is also important to understand the relationship between variables. Now it is the time to do the variable interactions. Let's first start with interact one categorical variable to another categorical variable. In many situations, it can be called cross-tabulation.

The relationship between two sets of classes or categories can be explored using correspondence tables created by the `table` function. Here we can cross tabulate the two categorical variables that we have already familiar with: `first_party` and `crime_rate`:

If we want to examine how the distribution of crime-rate categories varies for each first party. The question can be: for each political party, how are its wins distributed across different crime-rate levels?

```
# cross-tabulation first_party vs.crime_rate

table(pc_data$first_party, pc_data$crime_rate)
```

```
                  Average Higher Lower Much higher Much lower
  Alliance              0      0     0           0          0
  Conservative         25      6    44           0         41
  DUP                   0      0     0           0          0
  Green                 1      0     0           1          2
  Independent           0      3     0           2          0
  Labour               79    101    47         115         32
  Lib Dems              6      2    20           0         38
  Plaid Cymru           0      0     3           0          1
  Reform                2      3     0           0          0
  SDLP                  0      0     0           0          0
  Sinn Fein             0      0     0           0          0
  SNP                   0      0     0           0          0
  Speaker               1      0     0           0          0
  Ulster Unionist       0      0     0           0          0
  Unionist Voice        0      0     0           0          0
```

Conversely, we can also examine how the distribution of first parties varies across different crime-rate categories.The question this time is: for each crime-rate category, how are different parties distributed?

```
# cross-tabulation crime_rate vs first_party

table(pc_data$crime_rate, pc_data$first_party)
```

```
              Alliance Conservative DUP Green Independent Labour Lib Dems
  Average            0           25   0     1           0     79        6
  Higher             0            6   0     0           3    101        2
```

```
Lower                0           44   0     0           0     47        20
Much higher          0            0   0     1           2    115         0
Much lower           0           41   0     2           0     32        38

            Plaid Cymru Reform SDLP Sinn Fein SNP Speaker Ulster Unionist
Average               0      2    0         0   0       1               0
Higher                0      3    0         0   0       0               0
Lower                 3      0    0         0   0       0               0
Much higher           0      0    0         0   0       0               0
Much lower            1      0    0         0   0       0               0

            Unionist Voice
Average                  0
Higher                   0
Lower                    0
Much higher              0
Much lower               0
```

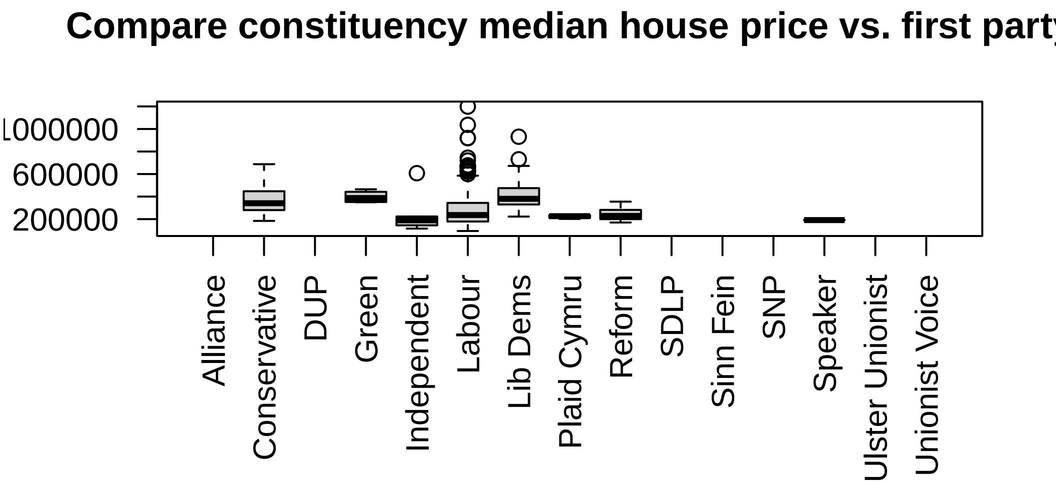
What insights now you can learn from these cross-tabulation? We will examine methods for determining whether the cross-tabulated counts and their differences are significant (i.e. would not be expected by chance) in later weeks.

2.4.3.4 Continuous to categorical: compare between groups

We may also be interested in the exploring differences and similarities in the continuous variables associated with for each categorical class. This can be done using multiple boxplots.

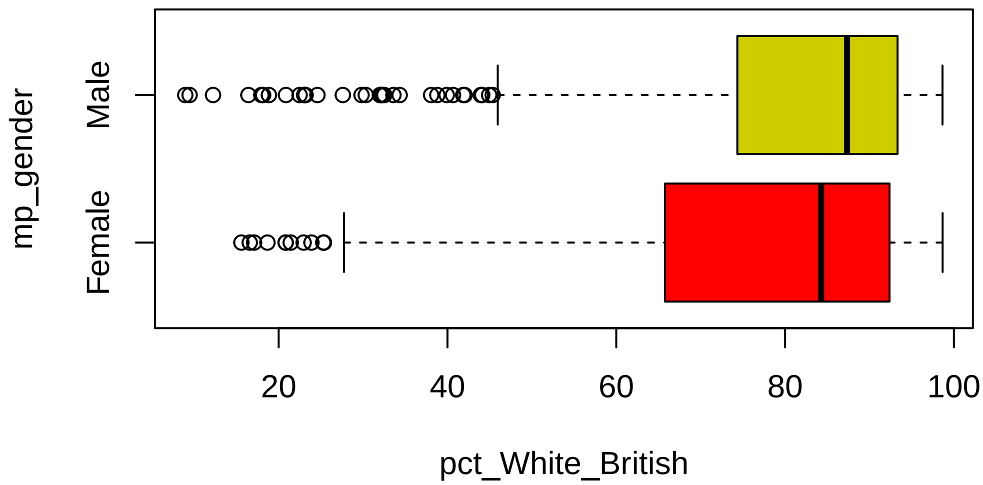
If we want to make boxplots of each party by the value median house price of the constituencies who vote for this party as their first party:

```
#create boxplots by median_house_price for each first_party

par(mar = c(10, 4, 4, 2))  # increase the margin of the chart for each side: bottom, left, to

boxplot(median_house_price ~ first_party,
        data = pc_data,
        las = 2, #vertical present item label
        xlab="", #no x axis label
        ylab="", #no y axis label,
        main="Compare constituency median house price vs. first party")
```



If we want to explore the % of White British in the constituencies and compare the distribution between different MP genders. The code below shows that, compared with constituencies represented by male MPs, those with female MPs generally have a lower proportion of White British population. This distribution also exhibits clear skewness towards lower percentages of White British residents.

```
boxplot(pct_White_British ~ mp_gender,
        data = pc_data,col=c("red", "yellow3"), #specify bar colors
        horizontal = TRUE) #horizonal the boxplot
```



We can do this numerically as well, but it is a bit more convoluted using the `with` and `aggregate` functions:

```
with(pc_data, aggregate(pct_White_British, by=list(mp_gender) , FUN=summary))
```

```
  Group.1     x.Min. x.1st Qu.  x.Median    x.Mean x.3rd Qu.    x.Max.
1  Female 15.587153 65.775762 84.276710 75.207471 92.374361 98.663686
2    Male  8.936846 74.343213 87.344901 79.907108 93.326348 98.652616
```

2.5 Make your own map for the election result

Having established how many MPs and votes each party got, it is time to look at the geography of the election outcome. To do this we need to link a set of digital boundaries for Parliamentary Constituencies with our pc_data dataset.

2.5.1 Read in Parliamentary Constituency Boundaries

Digital boundaries for the Parliamentary Constituencies used in the 2024 General Election can be found in the files *uk_constituencies_2024.geojson*. Both of these boundary datasets were sourced from https://automaticknowledge.org/wpc-hex/, and them simplified to make them quicker to plot.

Download the two digital boundary files from the Week 3 folder in CANVAS and save them to the same folder as your Week 3 Quarto document.

```
pc_map <- st_read("uk_constituencies_2024.gpkg") #read the boundaries as a spatial dataset by
```

```
Reading layer `uk_constituencies_2024' from data source
  `C:\Users\ziye\Documents\GitHub\quant\labs\uk_constituencies_2024.gpkg'
  using driver `GPKG'
Simple feature collection with 650 features and 9 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 191.9359 ymin: 7423.9 xmax: 655599.6 ymax: 1218591
Projected CRS: OSGB36 / British National Grid
```

2.5.2 Inspect the spatial dataset

Use the `names()` or `str()` to know the contents, or as above use `View()` to open and check. Check by yourself the number of rows and columns of the map data:

```
str(pc_map)
```

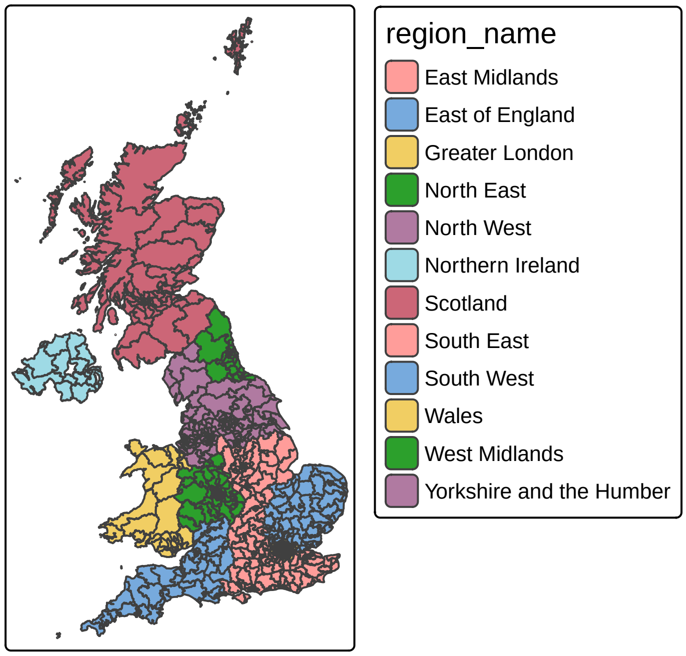
```
View(pc_map)
```

The *pc_map* dataset contains the standard set of Parliamentary Constituency boundaries:

```
#make a map of constituency
tm_shape(pc_map) + #map a spatial data
  tm_polygons() #map it as polygons
```

or we can make a colorful map by using different color for different regions:

```
#make a map of constituency and color each polygon based on "region_name"

tm_shape(pc_map) + #map a spatial data
  tm_polygons("region_name")  #map it as polygons, use different colors by region
```



2.5.3 Link boundaries to pc_data

In order to map the election results contained in the `pc_data` dataset, we need to join it to a set of digital boundaries using the `left_join( )` command - you should have already familiar with this from Week 1.

In your inspection of the `pc_data` and `pc_map` datasets, you may have noticed that they all have two variables in common. The first is a unique identifier for each Parliamentary Constituency: `gss_code`. The second is the name of the constituency: `pc_name`.

We can use these two variables to first link the `pc_data` dataset to the standard map:

```
#left join pc_data to pc_map, joining when gss_code from pc_map equals to pc_name in pc_data

pc_map_new <- left_join(pc_map, pc_data, by = c("gss_code", "pc_name"))
```

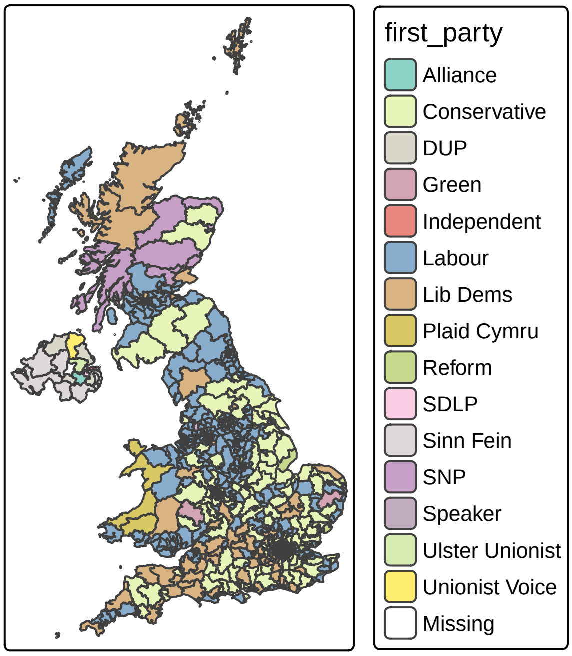
As ever, having created a new dataset, use the `dim( )`, `str( )`, `names( )` and `View( )` commands to check its contents are as you would expect.

2.5.4 Map the election result

Having joined the `pc_data` dataset with a set of digital boundaries, it becomes a simple matter to map the election results using the mapping skills covered in Week 1:

```
#make a map for the pc_map_new, fill the colors for each polygon based on "first_party", usi

tm_shape(pc_map_new) + #map a spatial data
  tm_polygons(fill = "first_party",
              palette="Set3")  #map it as polygons, use different colors by first_party
```

```
-- tmap v3 code detected -------------------------------------------------------

[v3->v4] `tm_tm_polygons()`: migrate the argument(s) related to the scale of
the visual variable `fill` namely 'palette' (rename to 'values') to fill.scale
= tm_scale(<HERE>).
[cols4all] color palettes: use palettes from the R package cols4all. Run
`cols4all::c4a_gui()` to explore them. The old palette name "Set3" is named
"brewer.set3"
Multiple palettes called "set3" found: "brewer.set3", "hcl.set3". The first one, "brewer.set3
```



This time, let's change the mode of tmap by using `tmap_mode()` from default "plot" to "view" for an interactive map:

```
# make the map interactive
tmap_mode("view")
tm_shape(pc_map_new) +
  tm_polygons("first_party")
```

This time, in your right-bottom pane, the map should be plotted in Viewer tab as an interactive map. You can zoom in/out to explore your map, for a better view, you can click the to open a webpage.

You may need to switch `tmap_mode()` back to "plot" for a static map making:

```
#switch back to plot static mode for later use

tmap_mode("plot")
```

```
i tmap mode set to "plot".
```

Now what pattern you can observe from the interactive map you just made?

2.6 Formative tasks

Task 1 How many columns and rows in the UK constituency boundary dataset?

```
Reading layer `uk_constituencies_2024' from data source
  `C:\Users\ziye\Documents\GitHub\quant\labs\uk_constituencies_2024.gpkg'
  using driver `GPKG'
Simple feature collection with 650 features and 9 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 191.9359 ymin: 7423.9 xmax: 655599.6 ymax: 1218591
Projected CRS: OSGB36 / British National Grid
```

```
[1] 650  10
```

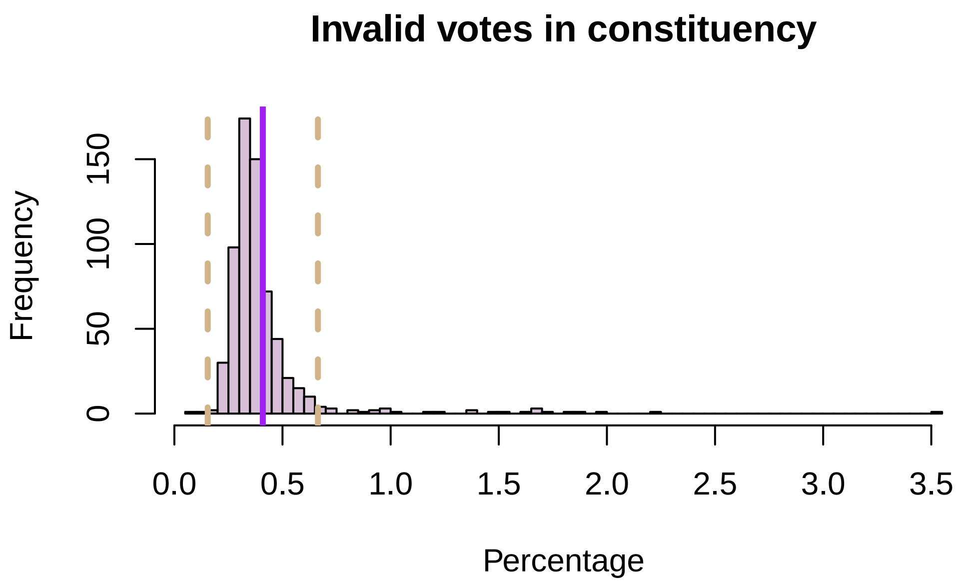
```
[1] 10
```

```
[1] 650
```

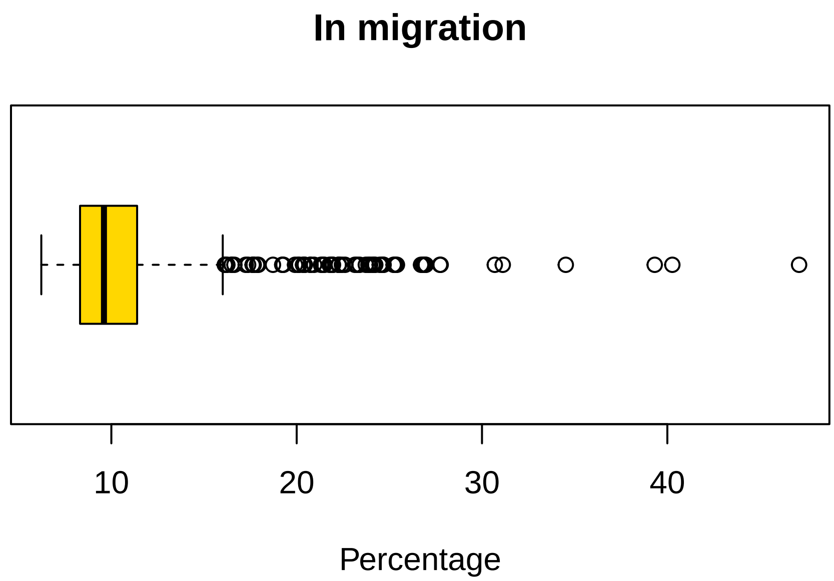
Task 2 Using the UK constituency boundary dataset, descriptive summary the area of all the constituencies:

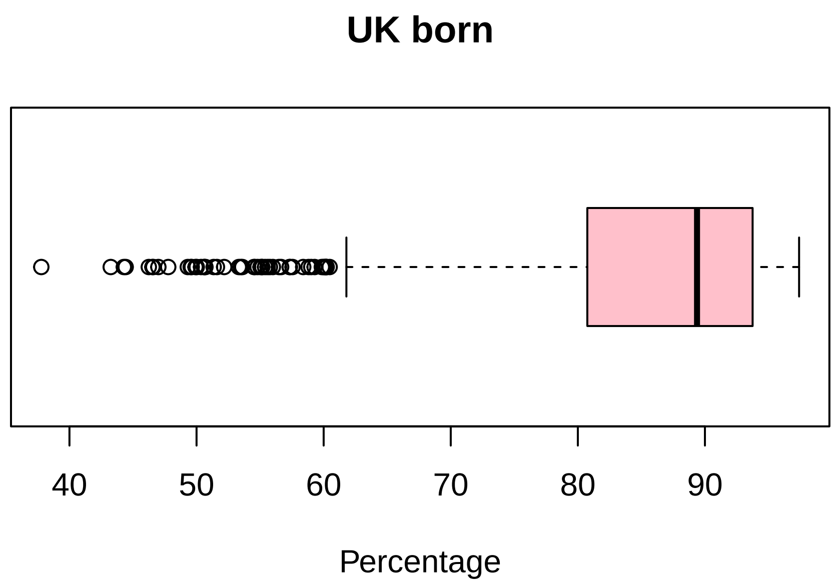
```
Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
6.80    33.65   106.45   375.01   351.98 11634.40
```

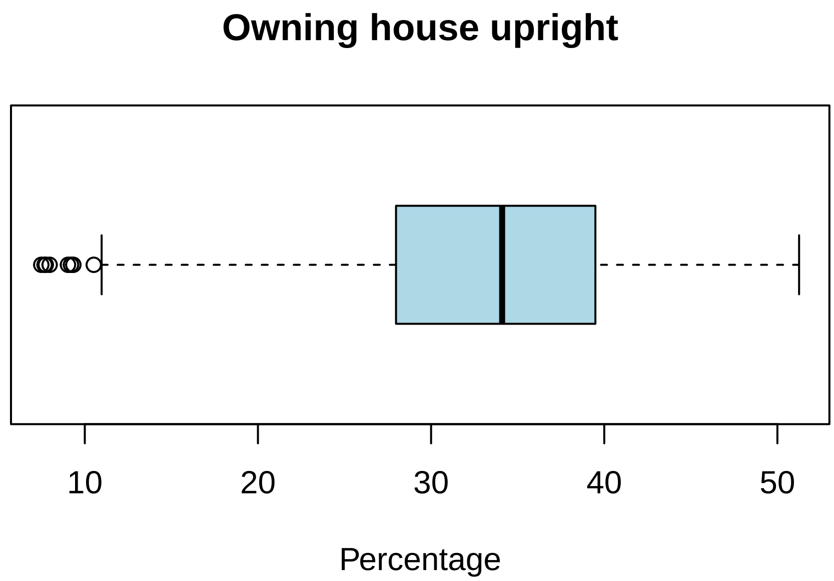
Task 3 Write some code to plot a histogram of the `pct_invalid_votes` variable in the constituency election dataset, with lines showing the mean and the standard deviation around the mean. Try add breaks into the function and change breaks from 10, 20 to 50 and see how the histogram changed.



Task 4 Write codes to create boxplots for `pct_in_migration`, `pct_UK_born` and `pct_owned` (owning upright household):

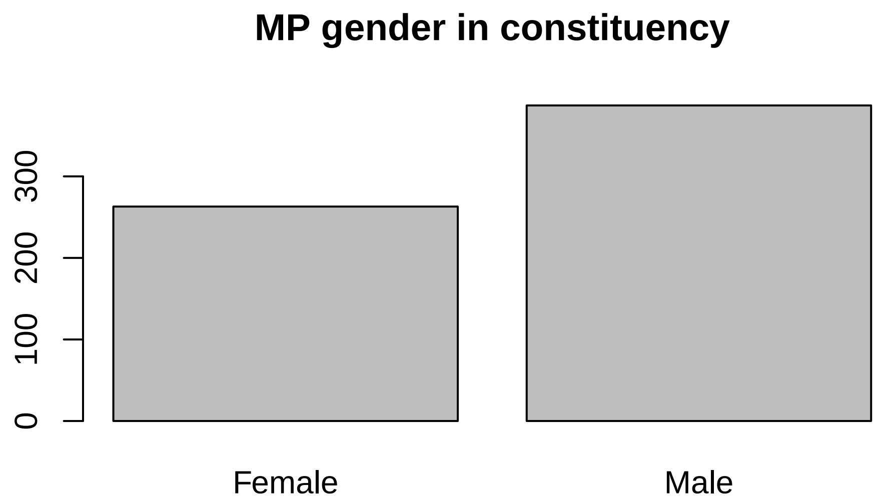

In migration
10
20
30
40
Percentage


UK born
40
50
60
70
80
90
Percentage



Task 4 Write codes to summary the counts of MPs in each gender (this is in the `mp_gender`variable), presenting the table with percentage and showing the barplot.

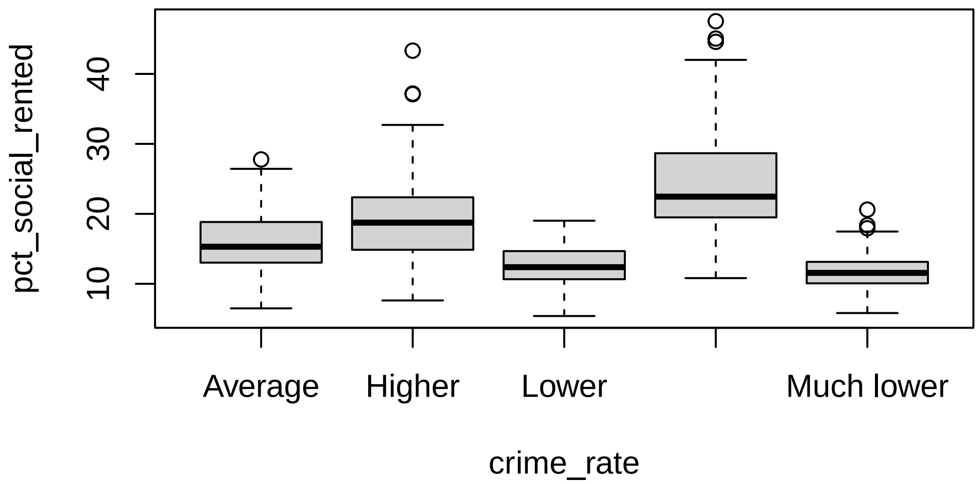
```
  mp_gender   n  pct
1    Female 263 40.5
2      Male 387 59.5
```



Task 5 Cross tabulation `region_name` to `mp_gender` by using the newly created `pc_map_new` dataset, which joined by the constituency boundary and election dataset.

```
                         Female Male
East Midlands                20   27
East of England              16   45
Greater London               38   37
North East                   12   15
North West                   31   42
Northern Ireland              5   13
Scotland                     20   37
South East                   39   52
South West                   19   39
Wales                        15   17
West Midlands                25   32
Yorkshire and the Humber     23   31
```

Task 6 Compare between `crime_rate` with the `pct_social_rented` in constituency election dataset. What pattern you can learn from your boxplot?


pct_social_rented
10
20
30
40
Average
Higher
Lower
Much lower
crime_rate