Modelling Criminological Data CRIM20452

2026-02-02

Contents

Preface

This study material is designed to introduce Criminology students at the University of Manchester to the use of data science in crime research and practice. This is an improved version of the material originally developed by Juanjo Medina and Reka Solymosi, and is currently being updated and maintained by Ana Nicoriciu - anamaria.nicoriciu@manchester.ac.uk and Sam Langton - samuel.langton@manchester.ac.uk.

This lab note is a work in process. If you have any suggestions or find any errors, please don't hesitate to contact us by submitting an issue to the GitHub repository. We appreciate your feedback and will use it to improve the material for future students.

Note for students * While this material briefly covers some concepts, students are expected to engage in weekly reading, attend lab sessions, and participate in lectures for a comprehensive course experience. These notes are not intended to be a stand-alone reference or textbook but rather a set of exercises to gain hands-on practice with the concepts introduced during the course. * This material is designed for Criminology students at the University of Manchester. They are meant to introduce students to the concept of descriptive statistics and the key concepts required to build an understanding of quantitative data analysis in crime research. * The handouts utilise various datasets, including data from the UK data service, such as the Crime Survey for England and Wales, which is available under an Open Government Licence. These datasets are designed to be a learning resource and should not be used for research purposes or the production of summary statistics.

Chapter 1

A first lesson about R

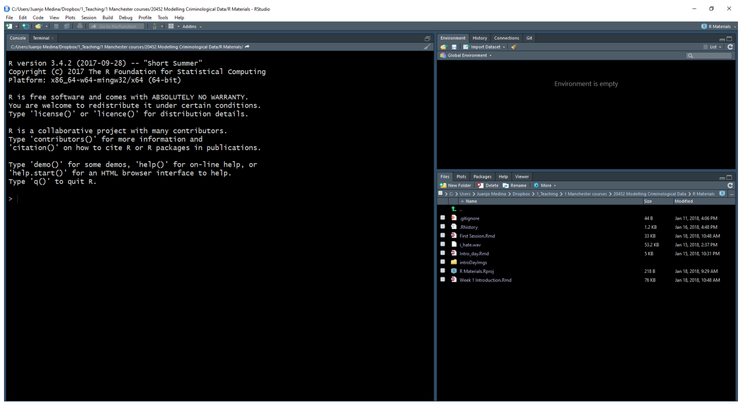
1.1 Install R & RStudio

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

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

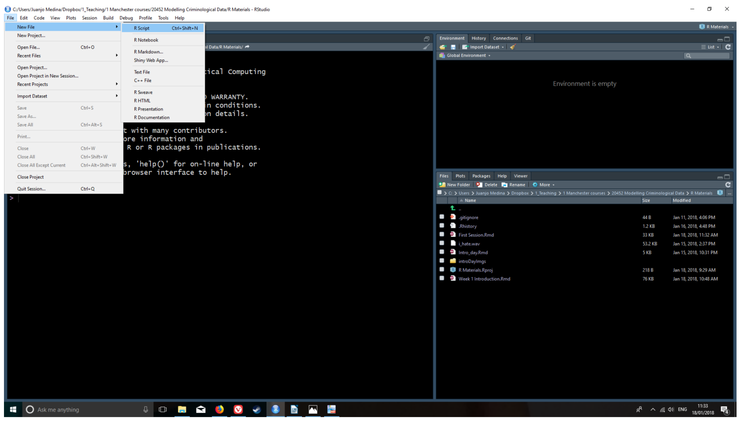
1.2 Open up and explore RStudio

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

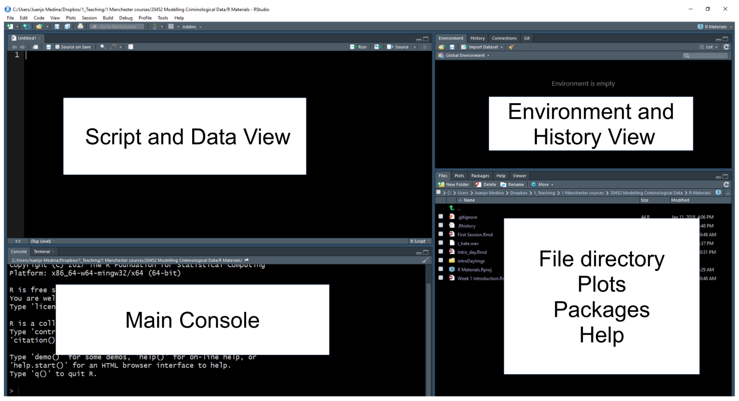

C:/Users/Juanjo Medina/Dropbox/1_Teaching/1 Manchester courses/20452 Modelling Criminological Data/R Materials - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console
Terminal
Environment History Connections Git
Import Dataset
Global Environment
Environment is empty
Files Plots Packages Help Viewer
New Folder Delete Rename More

```
R version 3.4.2 (2017-09-28) -- "Short Summer"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

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



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



```
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free s
You are wel
Type 'licen

R is a coll
Type 'contr
'citation()

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

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

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

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

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

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

One of the key advantages of doing data analysis this way is that you are producing a written record of every step you take in the analysis. The challenge, though, is that you need to learn this language in order to be able to use it.

That will be the main focus of this course, teaching you to write R code for data analysis purposes.

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

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

1.3 Customising the RStudio look

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

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

1.4 Getting organised: R Projects

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

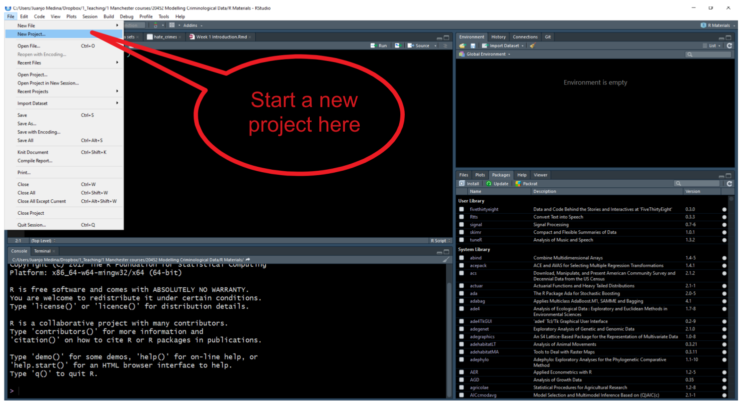
Windows users

MAC users

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

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

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



C:/Users/Juanjo Medina/Dropbox/1_Teaching/1 Manchester courses/20452 Modelling Criminological Data/R Materials/

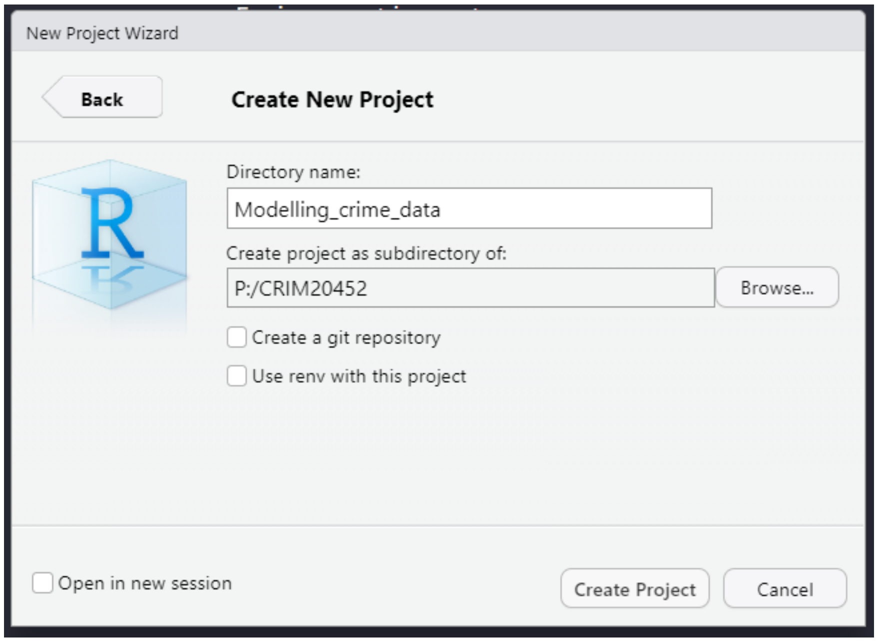
```
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Name	Description	Version
User Library		
fivethirtyeight	Data and Code Behind the Stories and Interactives at 'FiveThirtyEight'	0.3.0
Rtts	Convert Text into Speech	0.3.3
signal	Signal Processing	0.7-6
skimr	Compact and Flexible Summaries of Data	1.0.1
tuneR	Analysis of Music and Speech	1.3.2
System Library		
abind	Combine Multidimensional Arrays	1.4-5
acepack	ACE and AVAS for Selecting Multiple Regression Transformations	1.4.1
acs	Download, Manipulate, and Present American Community Survey and Decennial Data from the US Census	2.1.2
actuar	Actuarial Functions and Heavy Tailed Distributions	2.1-1
ada	The R Package Ada for Stochastic Boosting	2.0-5
adabag	Applies Multiclass AdaBoost.M1, SAMME and Bagging	4.1
ade4	Analysis of Ecological Data : Exploratory and Euclidean Methods in Environmental Sciences	1.7-8
ade4TkGUI	'ade4' Tcl/Tk Graphical User Interface	0.2-9
adegenet	Exploratory Analysis of Genetic and Genomic Data	2.1.0
adegraphics	An S4 Lattice-Based Package for the Representation of Multivariate Data	1.0-8
adehabitatLT	Analysis of Animal Movements	0.3.21
adehabitatMA	Tools to Deal with Raster Maps	0.3.11
adephylo	Adephylo: Exploratory Analyses for the Phylogenetic Comparative Method	1.1-10
AER	Applied Econometrics with R	1.2-5
AGD	Analysis of Growth Data	0.35
agricolae	Statistical Procedures for Agricultural Research	1.2-8
AICcmodavg	Model Selection and Multimodel Inference Based on (Q)AIC(c)	2.1-1

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

Now, you get another dialogue box where you have to specify what kind of project you want to create. Select the first option *New Project*.

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



Directory name: Modelling_crime_data

Create project as subdirectory of: P:/CRIM20452

☐ Create a git repository

☐ Use renv with this project

☐ Open in new session

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

- *Scripts and code*: Here, I put all the text files with my analytic code, including Rmarkdown files, which is something we will introduce much later in the semester.
- *Source data*: Here, I put the original data. I tend not to touch this once I have obtained the original data.
- *Documentation*: This is the subdirectory where I place all the data documentation (e.g., codebooks, questionnaires, etc.)
- *Modified data*: All analyses involve transformations and changes to the original data files. You don't want to mess up the original data files, so you should create new data files as soon as you start changing your source data. I go so far as to place them in a different subdirectory.
- *Literature*: Analysis is all about answering research questions. There is always a literature about these questions. I place the relevant literature for the analytic project I am conducting in this subdirectory.

- *Reports and write-up*: Here is where I file all the reports and data visualisations that are associated with my analysis.

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

1.5 Functions: Talk to your computer

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

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

```r
print("I love stats")
```

```
## [1] "I love stats"
```

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

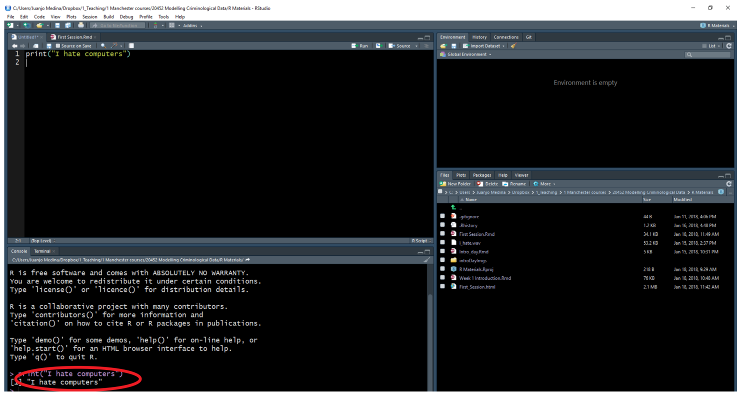
In these handouts, you will see grey boxes with bits of code. You can cut and paste this code into your script window and run the code from it to reproduce our results. As we go along, we will be covering new bits of code.

Sometimes, in these lab notes, you will see the results of running the code, which is what you see printed in your console or in your plot viewer. The results will appear enclosed in a box as above.

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

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

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



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

```
C:/Users/Juanjo Medina/Dropbox/1_Teaching/1 Manchester courses/20452 Modelling Criminological Data/R Materials/
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> print("I hate computers")
[1] "I hate computers"
```

Environment is empty

C: > Users > Juanjo Medina > Dropbox > 1_Teaching > 1 Manchester courses > 20452 Modelling Criminological Data > R Materials

Name	Size	Modified
..		
.gitignore	44 B	Jan 11, 2018, 4:06 PM
.Rhistory	1.2 KB	Jan 16, 2018, 4:48 PM
First Session.Rmd	34.1 KB	Jan 18, 2018, 11:49 AM
i_hate.wav	53.2 KB	Jan 15, 2018, 2:37 PM
Intro_day.Rmd	5 KB	Jan 15, 2018, 10:31 PM
introDayImgs		
R Materials.Rproj	218 B	Jan 18, 2018, 9:29 AM
Week 1 Introduction.Rmd	76 KB	Jan 18, 2018, 10:48 AM
First_Session.html	2.1 MB	Jan 18, 2018, 11:42 AM

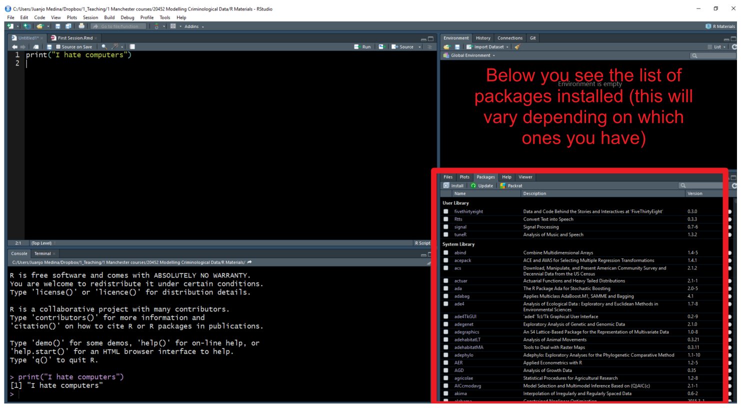
As indicated above, the window in the bottom left corner is the main **console**. You will see that the words "I love stats" appear printed there. If, rather than using R Studio, you were working directly from R, that's all you would get: the main console where you can write code interactively (rather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

1.6 More on packages

Before, we described packages as elements that add the functionality of R. Most packages introduce new functions that allow you to ask R to do different things.

Anybody can write a package, so consequently, R packages vary in quality and complexity. You can find packages in different places, as well, from official repositories (which means they have passed a minimum of quality control), something called GitHub (a webpage where software developers post work in progress), to personal webpages (danger, danger!). In early 2017 we passed the 10,000 packages mark just in the main official package repository, so the number of things that can be done with R grows exponentially every day as people keep adding new packages.

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


Below you see the list of packages installed (this will vary depending on which ones you have)
Environment is empty
Files
Plots
Packages
Help
Viewer
Install
Update
Packrat
Console
Terminal
R Script

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

```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> print("I hate computers")
[1] "I hate computers"
>
```

Name	Description	Version
User Library		
fivethirtyeight	Data and Code Behind the Stories and Interactives at 'FiveThirtyEight'	0.3.0
Rtts	Convert Text into Speech	0.3.3
signal	Signal Processing	0.7-6
tuneR	Analysis of Music and Speech	1.3.2
System Library		
abind	Combine Multidimensional Arrays	1.4-5
acepack	ACE and AVAS for Selecting Multiple Regression Transformations	1.4.1
acs	Download, Manipulate, and Present American Community Survey and Decennial Data from the US Census	2.1.2
actuar	Actuarial Functions and Heavy Tailed Distributions	2.1-1
ada	The R Package Ada for Stochastic Boosting	2.0-5
adabag	Applies Multiclass AdaBoost.M1, SAMME and Bagging	4.1
ade4	Analysis of Ecological Data : Exploratory and Euclidean Methods in Environmental Sciences	1.7-8
ade4TkGUI	'ade4' Tcl/Tk Graphical User Interface	0.2-9
adegenet	Exploratory Analysis of Genetic and Genomic Data	2.1.0
adegraphics	An S4 Lattice-Based Package for the Representation of Multivariate Data	1.0-8
adehabitatLT	Analysis of Animal Movements	0.3.21
adehabitatMA	Tools to Deal with Raster Maps	0.3.11
adephylo	Adephylo: Exploratory Analyses for the Phylogenetic Comparative Method	1.1-10
AER	Applied Econometrics with R	1.2-5
AGD	Analysis of Growth Data	0.35
agricolae	Statistical Procedures for Agricultural Research	1.2-8
AICcmodavg	Model Selection and Multimodel Inference Based on (Q)AIC(c)	2.1-1
akima	Interpolation of Irregularly and Regularly Spaced Data	0.6-2

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

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

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

Here, we are introducing a new function, "install.packages" and what we have passed as an argument is the name of the package that we want to install. This is how we install a package *that is available in the official CRAN repository*. Given that you are connecting to an online repository, you will need an internet connection every time you want to install a package. CRAN is an official repository that has a collection of R packages that meet a minimum set of quality criteria. It's a fairly safe place to get packages from. If we wanted to install a package from somewhere else, we would have to adapt the code. Later this semester, you will see how we install packages from GitHub.

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

How do you find out what a package does? You look at the relevant documentation. In the Packages window, scroll down until you find the new package we installed. Here, you will see the name of the package (cowsay), the source of the package (i.e., where got it from, CRAN). and the package version. The version

I have for cowsay is 1.2.2. Yours may be older or newer. It doesn't matter much at this point.

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

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

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

```r
say("I love stats")
```

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

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

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

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

```r
search()
```

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

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

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

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

```r
say("I love stats")
```

```
## 
##  -------------- 
## < I love stats >
##  -------------- 
##       \
##        \
## 
##         ^__^ 
##         (oo)\ ________ 
##         (__)\         )\ /\ 
##              ||------w|
##              ||      ||
```

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

```r
say("I love stats", "cow")
```

```
## 
##  -------------- 
## < I love stats >
##  -------------- 
##       \
##        \
## 
##         ^__^ 
##         (oo)\ ________ 
##         (__)\         )\ /\ 
##              ||------w|
##              ||      ||
```

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

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

1.7 Objects: creating an object

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

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

```
my_text <- "I love stats."
```

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

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

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

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

```
##
##  ---------------
## < I love stats. >
##  ---------------
##       \
##        \
##
##         ^__^
##         (oo)\ ________
##         (__)\         )\ /\
##              ||------w|
##              ||      ||
```

1.8 More on objects

Now that we have covered some of the preliminaries, we can move on to the data. In Excel, you are used to seeing your data in spreadsheet format. If you need some recap, you should review some of the materials from previous modules. This chapter will be helpful to have a better understanding of the notion of a data set, levels of measurement, and tidy data.

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

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

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

```r
x <- 5
```

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

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

```
x
```

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

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

```
X
```

```
## Error: object 'X' not found
```

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

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

When you get an error message or implausible results, you want to look back at your code to figure out what the problem is. This process is called **debugging**. There are some proper systematic ways to write code that facilitate debugging, but we won't get into that here. R is very good with automatic error handling at the levels we'll be using it at. Very often, the solution will simply involve correcting the spelling or checking you've used the correct placement and number of brackets or commas (you will find this out later, the hard way).

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

1.9 Naming conventions for objects in R

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

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

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

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

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

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

You get an error!

1.10 R object types: vectors

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

What is a vector? A vector is simply a set of elements *of the same class*. Typically, these classes are character (i.e., text), numeric, integer, or logical (i.e., True or False). Vectors are the basic data structure in R.

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

```
my_1st_vector <- c(0.5, 0.6) #creates a numeric vector with two elements
my_2nd_vector <- c(1L, 2L) #creates an integer vector ("L" suffix specifies an integer
my_3rd_vector <- c(TRUE, FALSE) #creates a logical vector
my_4th_vector <- c(T, F) #creates a logical vector using abbreviations of True and Fal
#but you should avoid this formulation and instead use the full word.
my_5th_vector <- c("a", "b", "c") #creates a character vector
my_6th_vector <- c(1+0i, 2+4i) #creates a complex vector (we won't really use in this
```

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

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

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

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

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

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

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

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

```
my_7th_vector <- my_1st_vector + 1 #Create a new vector that contains
#the elements of my1stvector plus a constant of 1
my_1st_vector + my_7th_vector #Adds the two vectors and Auto-print
```

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

```
#the results (note how the sum was done)
```

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

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

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

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

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

It is also possible to remove all objects at once:

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

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

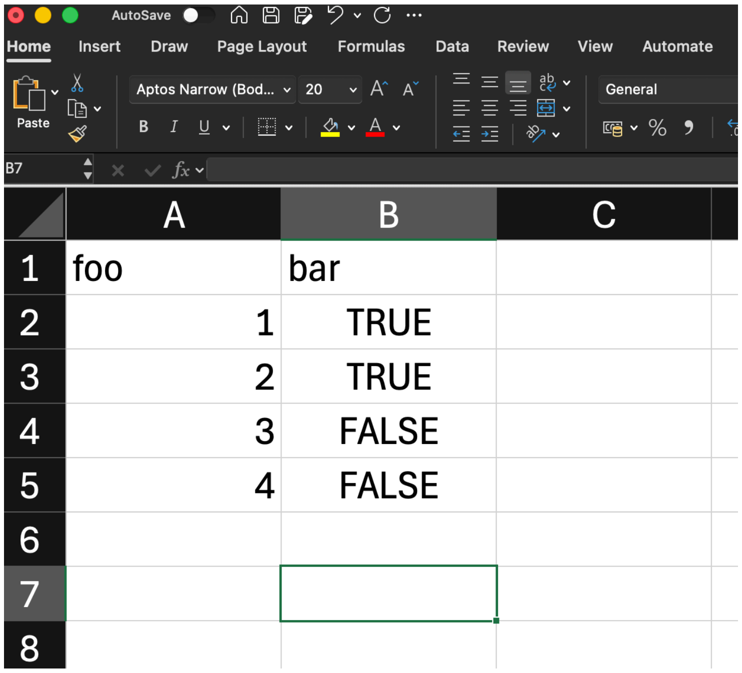
```
my_8th_vector <- c(0.5, "a")
class(my_8th_vector) #The class() function will tell us the class of the vector
```

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

1.11 R object types: Data frame

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

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



	A	B	C
1	foo	bar	
2	1	TRUE	
3	2	TRUE	
4	3	FALSE	
5	4	FALSE	
6			
7			
8			

Let's create a data frame with two variables:

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

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

Or alternatively, for the same result:

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

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

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

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

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

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

```
library(fivethirtyeight)
```

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

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

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

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

This function will search among all the *loaded* packages and locate the "hate_crimes" data set.

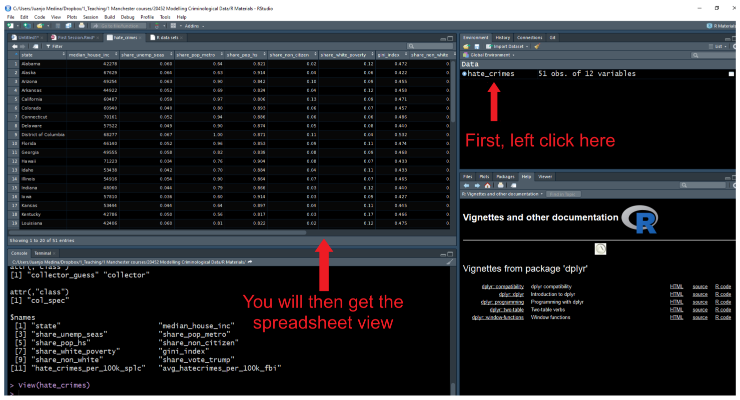
Every object in R can have **attributes**. These are names; dimensions (for matrices and arrays: number of rows and columns) and dimensions names; class of object (numeric, character, etc.); length (for a vector, this will be the number of elements in the vector); and other user-defined. You can access the attributes of an object using the `attributes()` function. Let's query R for the attributes of this data frame.

```
attributes(hate_crimes)
```

```
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
## [51] 51
##
## $class
## [1] "tbl_df"     "tbl"        "data.frame"
##
## $names
##  [1] "state"                       "state_abbrev"
##  [3] "median_house_inc"            "share_unemp_seas"
##  [5] "share_pop_metro"             "share_pop_hs"
##  [7] "share_non_citizen"           "share_white_poverty"
##  [9] "gini_index"                  "share_non_white"
## [11] "share_vote_trump"            "hate_crimes_per_100k_splc"
## [13] "avg_hatecrimes_per_100k_fbi"
```

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

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


C:/Users/Juanjo Medina/Dropbox/1_Teaching/1 Manchester courses/20452 Modelling Criminological Data/R Materials - RStudio
First, left click here
You will then get the spreadsheet view

	state	median_house_inc	share_unemp_seas	share_pop_metro	share_pop_hs	share_non_citizen	share_white_poverty	gini_index	share_non_white
1	Alabama	42278	0.060	0.64	0.821	0.02	0.12	0.472	0.
2	Alaska	67629	0.064	0.63	0.914	0.04	0.06	0.422	0.
3	Arizona	49254	0.063	0.90	0.842	0.10	0.09	0.455	0.
4	Arkansas	44922	0.052	0.69	0.824	0.04	0.12	0.458	0.
5	California	60487	0.059	0.97	0.806	0.13	0.09	0.471	0.
6	Colorado	60940	0.040	0.80	0.893	0.06	0.07	0.457	0.
7	Connecticut	70161	0.052	0.94	0.886	0.06	0.06	0.486	0.
8	Delaware	57522	0.049	0.90	0.874	0.05	0.08	0.440	0.
9	District of Columbia	68277	0.067	1.00	0.871	0.11	0.04	0.532	0.
10	Florida	46140	0.052	0.96	0.853	0.09	0.11	0.474	0.
11	Georgia	49555	0.058	0.82	0.839	0.08	0.09	0.468	0.
12	Hawaii	71223	0.034	0.76	0.904	0.08	0.07	0.433	0.
13	Idaho	53438	0.042	0.70	0.884	0.04	0.11	0.433	0.
14	Illinois	54916	0.054	0.90	0.864	0.07	0.07	0.465	0.
15	Indiana	48060	0.044	0.79	0.866	0.03	0.12	0.440	0.
16	Iowa	57810	0.036	0.60	0.914	0.03	0.09	0.427	0
17	Kansas	53444	0.044	0.64	0.897	0.04	0.11	0.445	0.
18	Kentucky	42786	0.050	0.56	0.817	0.03	0.17	0.466	0.
19	Louisiana	42406	0.060	0.81	0.822	0.02	0.12	0.475	0

Showing 1 to 20 of 51 entries

```
[1] "collector_guess" "collector"

attr(,"class")
[1] "col_spec"

$names
 [1] "state"                    "median_house_inc"
 [3] "share_unemp_seas"         "share_pop_metro"
 [5] "share_pop_hs"             "share_non_citizen"
 [7] "share_white_poverty"      "gini_index"
 [9] "share_non_white"          "share_vote_trump"
[11] "hate_crimes_per_100k_splc"   "avg_hatecrimes_per_100k_fbi"

> View(hate_crimes)
>
```

Data

hate_crimes   51 obs. of 12 variables

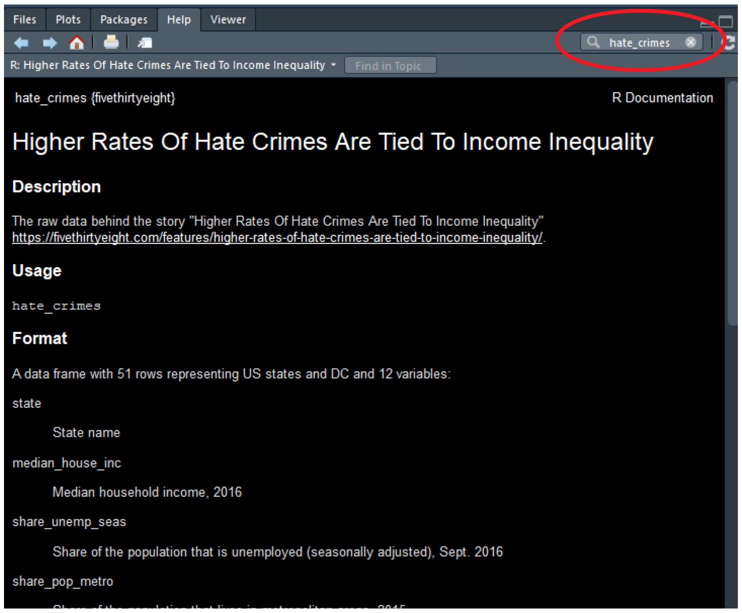
Vignettes and other documentation

Vignettes from package 'dplyr'

dplyr::compatibility	dplyr compatibility	HTML	source	R code
dplyr::dplyr	Introduction to dplyr	HTML	source	R code
dplyr::programming	Programming with dplyr	HTML	source	R code
dplyr::two-table	Two-table verbs	HTML	source	R code
dplyr::window-functions	Window functions	HTML	source	R code

1.12 Exploring data

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


Files
Plots
Packages
Help
Viewer
hate_crimes
R: Higher Rates Of Hate Crimes Are Tied To Income Inequality
Find in Topic

hate_crimes {fivethirtyeight}   R Documentation

Higher Rates Of Hate Crimes Are Tied To Income Inequality

Description

The raw data behind the story "Higher Rates Of Hate Crimes Are Tied To Income Inequality" https://fivethirtyeight.com/features/higher-rates-of-hate-crimes-are-tied-to-income-inequality/.

Usage

`hate_crimes`

Format

A data frame with 51 rows representing US states and DC and 12 variables:

state

    State name

median_house_inc

    Median household income, 2016

share_unemp_seas

    Share of the population that is unemployed (seasonally adjusted), Sept. 2016

share_pop_metro

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

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

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

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

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

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

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

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

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

```
summary(hate_crimes)
```

```
##     state           state_abbrev       median_house_inc share_unemp_seas
##  Length:51          Length:51          Min.   :35521    Min.   :0.02800 
##  Class :character   Class :character   1st Qu.:48657    1st Qu.:0.04200 
##  Mode  :character   Mode  :character   Median :54916    Median :0.05100 
##                                        Mean   :55224    Mean   :0.04957 
##                                        3rd Qu.:60719    3rd Qu.:0.05750 
##                                        Max.   :76165    Max.   :0.07300 
##                                                                         
##  share_pop_metro   share_pop_hs    share_non_citizen share_white_poverty
##  Min.   :0.3100   Min.   :0.7990   Min.   :0.01000   Min.   :0.04000    
##  1st Qu.:0.6300   1st Qu.:0.8405   1st Qu.:0.03000   1st Qu.:0.07500    
##  Median :0.7900   Median :0.8740   Median :0.04500   Median :0.09000
##  Mean   :0.7502   Mean   :0.8691   Mean   :0.05458   Mean   :0.09176
##  3rd Qu.:0.8950   3rd Qu.:0.8980   3rd Qu.:0.08000   3rd Qu.:0.10000
##  Max.   :1.0000   Max.   :0.9180   Max.   :0.13000   Max.   :0.17000
##                                    NA's   :3
##    gini_index     share_non_white  share_vote_trump hate_crimes_per_100k_splc
##  Min.   :0.4190   Min.   :0.0600   Min.   :0.040    Min.   :0.06745
##  1st Qu.:0.4400   1st Qu.:0.1950   1st Qu.:0.415    1st Qu.:0.14271
##  Median :0.4540   Median :0.2800   Median :0.490    Median :0.22620
##  Mean   :0.4538   Mean   :0.3157   Mean   :0.490    Mean   :0.30409
##  3rd Qu.:0.4665   3rd Qu.:0.4200   3rd Qu.:0.575    3rd Qu.:0.35693
##  Max.   :0.5320   Max.   :0.8100   Max.   :0.700    Max.   :1.52230
##                                                     NA's   :4
##  avg_hatecrimes_per_100k_fbi
##  Min.   : 0.2669
##  1st Qu.: 1.2931
##  Median : 1.9871
##  Mean   : 2.3676
##  3rd Qu.: 3.1843
##  Max.   :10.9535
##  NA's   :1
```

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

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

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

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

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

```
skim(hate_crimes)
```

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

Apart from summary statistics, last semester, we discussed a variety of ways to graphically display variables. Week 3 of 'Making Sense of Criminological Data' covered scatterplots, a graphical device to show the relationship between two quantitative variables. I don't know if you remember the number of points and clicks you had to make in Excel to get this done.

1.13 R data types: Factors

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

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

Factors can also be created with the `factor()` function concatenating a series of *character* elements. You will notice that it is printed differently from a simple character vector and that it tells us the *levels* of the factor (i.e., each unique value in the factor).

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

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

Alternatively, for similar results, use the as.factor() function. Here, you will create `the_smiths_char` object and then transform it to a factor variable, `the_smiths_f`.

```
the_smiths_char <- c("Morrisey", "Marr", "Rourke", "Joyce") #create a character vector
the_smiths_f <- as.factor(the_smiths_char) #create a factor using a character vector
the_smiths_f #auto-print factor
```

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

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

```
levels(the_smiths_f)
```

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

Notice that the levels appear printed in alphabetical order (Try `levels(the_smiths_char)` and see what R says. Yes, `the_smiths_char` and `the_smiths_f` are different!). There will be situations when this is not the most convenient order (e.g., *Dec, Jan, Mar (Alphabetical order)* instead of *Jan, Mar, Dec*). Later on, we will discuss in these tutorials how to reorder your factor levels when you need to.

Let's look at one more example here. Let's say we are making data about Hogwarts Houses, which are divided into four houses: Gryffindor, Hufflepuff, Ravenclaw and Slytherin.

```
#We create a data frame called HarryPotter with two variables:
#a character vector called name and a character vector called house
HarryPotter <- data.frame(name = c("Potter", "Malfoy", "Lovegood", "Chang",
              "Hagrid", "Diggory"), house = c("Gryffindor", "Slytherin",
              "Ravenclaw", "Ravenclaw", "Gryffindor", "Hufflepuff"))
HarryPotter
```

```
##       name      house
## 1   Potter Gryffindor
## 2   Malfoy  Slytherin
## 3 Lovegood  Ravenclaw
## 4    Chang  Ravenclaw
## 5   Hagrid Gryffindor
## 6  Diggory Hufflepuff
```

Use `str(HarryPotter$house)` and see what R says. R will list all observations in the variable and say it's a character variable, right? Now, we are going to convert `house`, a character variable, into a factor variable `house_f`, meaning that R will categorise the variable.

```
HarryPotter$house_f <- as.factor(HarryPotter$house)
str(HarryPotter$house_f)
```

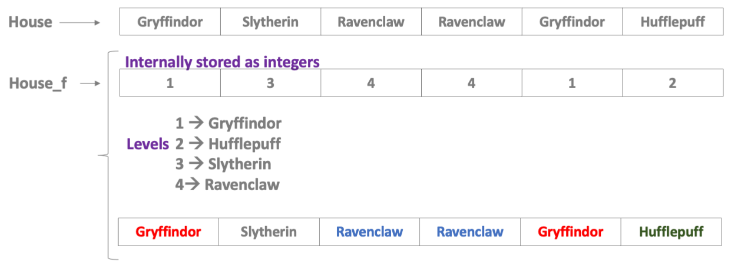
```
##  Factor w/ 4 levels "Gryffindor","Hufflepuff",..: 1 4 3 3 1 2
```

```
levels(HarryPotter$house_f)
```

```
## [1] "Gryffindor" "Hufflepuff" "Ravenclaw"  "Slytherin"
```

```
#try 'levels(HarryPotter$house)' and find the difference
```

Now, can you clearly understand what **factor** means in R? Factors are used to represent categorical data. Once created, factors can contain pre-defined set values, known as `levels`. Like we just converted 6-character data (`house`) into 4-factor data! (`house_f`).



1.14 How to import data

Programmers are lazy, and the whole point of using code-based interfaces is that we get to avoid doing unnecessary work, like point-and-click downloading of files. When data exists online in a suitable format, we can tell R to read the data from the web directly and cut out the middleman (that being ourselves in our pointing-and-clicking activity).

How can we do this? Well, think about what we do when we read in a file. We say, "Dear R, I would like to create a new object, please, and I will call this new object `my_dataframe`". We do this by typing the name we are giving the object and the assignment function `<-`(assignment operator). Then, on the right-hand side of the assignment function, there is the value to which we are assigning the variable. So it could be a bit of text (such as when you're creating a `my_text` object and you pass it the string "I love stats"), or it could be some function, for example, when you read a CSV file with the `read_csv()` function.

So, if we're reading a CSV, we also need to specify *where* to read the CSV from. Where should R look to find this data? This is where normally you are putting in the path to your file, right?

Something like:

```
my_dataframe <- read.csv('PATH_OR_URL_TO_CSV_FILE')
```

Well, what if your data does not live on your laptop or PC? Well, if there is a way that R can still access this data just by following a path, then this approach will still work! Tips! Please keep your folders simple. Just use one data folder for this module and do not create multiple folders such as 'data for week 1', 'data for week 2', or something like this.

All data we will use in the module are on Canvas, but we will also provide the links to where we saved the data for you. You know when you right-click on the link, and select "Save As..." or whatever you click on to save? You could also select "Copy Link Address". This just copies the webpage where this data is stored. If you want, find a csv file from a previous module on Canvas, and give this a go. Copy the address, and then paste it into your browser. It will take you to a blank page where a forced download of the data will begin. So what if you pasted this into the `read.csv()` function?

```
#example 1: when you download data directly from the webpage,
#you will use this code.
my_dataframe <- read.csv("www.data.com/data you want to import.csv")
#example 2: when you download data from Canvas and save it to your
#computer, you can use file.choose()
#file.choose() brings up a file explorer window that allows
#you to interactively choose a file path to work with.
my_dataframe <- read.csv(file.choose())
#example 3: if you can't find a URL from Canvas to use yet,
#you can try this random dataset. The dataset itself is irrelevant
#(although it is about UFO sightings - pretty interesting if you ask me),
#but it does show you how to load data in from a real URL.
ufo_sightings <- read.csv('https://raw.githubusercontent.com/rfordatascience/tidytuesda
```

In the first example, the my_dataframe object would be assigned the value returned from the read.csv() function reading in the file from the 'URL' link you provided. File path is no mysterious thing, file path is simply the *path* to the *file* you want to read. If this is a website, then so be it.

R also can read Stata (.dta) files using the `read_dta()` function in the Haven package and SPSS (.sav) files using the `read_spss()` function also in the Haven Package. There are so many different codes we can use to import data into R. In this course, you will learn one by one!

1.15 How to use 'comment'

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

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

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

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

Just keep in mind:

- You need one `#` per line, and anything after that is a comment that is not executed by R.
- You can use spaces after.

1.16 How to Quit RStudio

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

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

The truth is what is real is your scripts and the data that your scripts use as inputs. You don't need anything in your environment because you can recreate those things by re-running your scripts. I like keeping things tidy, so when I am asked whether I want to save the image, my answer is always no. Most long-time users of R never save the workspace or care about saving the history. Remember, what is real is your scripts and the data.

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

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

1.17 Summary

This week, we used the following R functions:

install and load a package

- install.packages()
- library()

generate and print data

- my_text <- "I love stats."
- print()

- say()

explore data

- search()
- skim()
- attribute()
- mean()
- summary()
- str()

transform variables into factor variables

- as_factor()

Chapter 2

Appendix

2.0.1 Expected frequencies

Notice that R is telling us that the minimum expected frequency is 41.68. Why? For the Chi-squared test to work, it assumes the cell counts are sufficiently large. Precisely what constitutes 'sufficiently large' is a matter of some debate. One rule of thumb is that all expected cell counts should be above 5. If we have small cells, one alternative is to rely on the Fisher's Exact Test rather than on the Chi-Square. We don't have to request it here. Our cells are large enough for Chi Square to work fine. But if we needed, we could obtain the Fisher's Exact Test with the following code:

```
BCS0708<-read.csv("https://raw.githubusercontent.com/uom-resquant/modelling_book/refs/heads/maste
library(gmodels)
BCS0708$rubbcomm <- as.factor(BCS0708$rubbcomm)
BCS0708$rubbcomm <- factor(BCS0708$rubbcomm,
                    levels = c("not at all common",
                    "not very common", "fairly common",
                    "very common"))
fisher.test(BCS0708$rubbcomm, BCS0708$bcsvictim, simulate.p.value=TRUE)
```

```
##
##  Fisher's Exact Test for Count Data with simulated p-value (based on
##  2000 replicates)
##
## data:  BCS0708$rubbcomm and BCS0708$bcsvictim
## p-value = 0.0004998
## alternative hypothesis: two.sided
```

The p-value is still considerably lower than our alpha level of .05. So, we can still conclude that the relationship we observe can be generalised to the population.

Remember that we didn't need the Fisher test. However, as suggested above, there may be times when you need them.

2.1 Logistic regression

2.1.1 Fitting logistic regression: alternative

Another way of fitting a logistic regression and getting the odds ratio with less typing is to use the `Logit()` function in the `lessR` package (you will need to install it if you do not have it).

```r
library(effects)
data(Arrests, package="effects")

Arrests$harsher <- relevel(Arrests$released, "Yes")
#Rename the levels so that it is clear we now mean yes to harsher treatment
levels(Arrests$harsher) <- c("No","Yes")
#Check that it matches in reverse the original variable

Arrests$colour <- relevel(Arrests$colour, "White")
library(lessR, quietly= TRUE)
Logit(harsher ~ checks + colour + sex + employed, data=Arrests, brief=TRUE)
```

```
##
## >>> Note:  colour is not a numeric variable.
##            Indicator variables are created and analyzed.
##
## >>> Note:  sex is not a numeric variable.
##            Indicator variables are created and analyzed.
##
## >>> Note:  employed is not a numeric variable.
##            Indicator variables are created and analyzed.
##
## Response Variable:   harsher
## Predictor Variable 1:  checks
## Predictor Variable 2:  colourBlack
## Predictor Variable 3:  sexMale
## Predictor Variable 4:  employedYes
##
## Number of cases (rows) of data:  5226
## Number of cases retained for analysis:  5226
##
##
```

```
##    BASIC ANALYSIS
##
## -- Estimated Model of harsher for the Logit of Reference Group Membership
##
##              Estimate    Std Err  z-value  p-value   Lower 95%   Upper 95%
## (Intercept)   -1.9035     0.1600  -11.898    0.000     -2.2170     -1.5899
##      checks    0.3580     0.0258   13.875    0.000      0.3074      0.4085
## colourBlack    0.4961     0.0826    6.003    0.000      0.3341      0.6580
##     sexMale    0.0422     0.1496    0.282    0.778     -0.2511      0.3355
## employedYes   -0.7797     0.0839   -9.298    0.000     -0.9441     -0.6154
##
##
## -- Odds Ratios and Confidence Intervals
##
##              Odds Ratio   Lower 95%   Upper 95%
## (Intercept)      0.1491      0.1089      0.2039
##      checks      1.4304      1.3599      1.5046
## colourBlack      1.6423      1.3967      1.9310
##     sexMale      1.0431      0.7779      1.3986
## employedYes      0.4585      0.3890      0.5404
##
##
## -- Model Fit
##
##     Null deviance: 4776.258 on 5225 degrees of freedom
## Residual deviance: 4330.699 on 5221 degrees of freedom
##
## AIC: 4340.699
##
## Number of iterations to convergence: 5
##
##
## Collinearity
##
##             Tolerance       VIF
## checks          0.908     1.101
## colourBlack     0.963     1.038
## sexMale         0.982     1.019
## employedYes     0.931     1.074
```

2.1.2 Assessing model fit: deviance and pseudo r squared

As you may remember, when looking at linear models, we could use an F test to check the overall fit of the model, and we could evaluate R squared. When running logistic regression, we cannot obtain the R squared (although there is a collection of pseudo-R^2 measures that have been produced). In linear regression, things are a bit simpler. As Menard (2010: 43) explains:

> "there is only one reasonable residual variation criterion for quantitative variables in OLS, the familiar error sum of squares… but there are several possible residual variation criteria (entropy, squared error, qualitative difference) for binary variables. Another hindrance is the existence of numerous mathematical equivalents to R^2 in OLS, which are not necessarily mathematically (same formula) or conceptually (same meaning in the context of the model) equivalent to R^2 in logistic regression… Moreover, in logistic regression, we must choose whether we are more interested in qualitative prediction (whether predicitons are correct or incorrect), quantitative prediction (how close predictions are to being correct), or both, because different measures of explained variation are appropriate for these two different types of prediction"

A common starting point for assessing model fit is to look at the **log-likelihood statistic** and the **deviance** (also referred to as -2LL).

The log-likelihood aims to provide a measure of how much-unexplained variation there is after you fit the mode. Large values indicate poor fit.

The deviance, on the other hand, is simply the log-likelihood multiplied by -2 and is generally abbreviated as -2LL. The deviance will be a positive value, and *larger values indicate worse prediction* of the response variable. It is analogous to the error sum of squares in linear regression. In the same way that OLS linear regression tries to minimise the error sum of squares, maximum likelihood logistic regression tries to minimise the -2LL.

The difference between the -2LL for the model with no predictors and the -2LL for the model with all the predictors is the closer we get in logistic regression to the regression sum of squares. This difference is often called **model chi-squared**, and it provides a test of the null hypothesis that all the regression coefficients equal zero. It is, thus, equivalent to the F test in OLS regression.

```
fitl_1 <- glm(harsher ~ checks + colour + sex + employed, data=Arrests, family = "binon
summary(fitl_1)
```

```
##
## Call:
## glm(formula = harsher ~ checks + colour + sex + employed, family = "binomial",
##     data = Arrests)
##
## Coefficients:
##             Estimate Std. Error z value             Pr(>|z|)
## (Intercept) -1.90346    0.15999 -11.898 < 0.0000000000000002 ***
## checks       0.35796    0.02580  13.875 < 0.0000000000000002 ***
## colourBlack  0.49608    0.08264   6.003        0.00000000194 ***
## sexMale      0.04215    0.14965   0.282                0.778    
## employedYes -0.77973    0.08386  -9.298 < 0.0000000000000002 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 4776.3  on 5225  degrees of freedom
## Residual deviance: 4330.7  on 5221  degrees of freedom
## AIC: 4340.7
##
## Number of Fisher Scoring iterations: 5
```

In our example, we saw that some measures of fit were printed below the table with the coefficients. The **null deviance** is the deviance of the model with no predictors, and the **residual deviance** is simply the deviance of this model. You clearly want the residual deviance to be smaller than the null deviance. The difference between the null and the residual deviance is what we call the model chi-squared. In this case, this is 4776.3 minus 4330.7. We can ask R to do this for us.

First, notice that the object we created has all the information we need already stored.

```r
names(fitl_1)
```

```
##  [1] "coefficients"      "residuals"         "fitted.values"    
##  [4] "effects"           "R"                 "rank"             
##  [7] "qr"                "family"            "linear.predictors"
## [10] "deviance"          "aic"               "null.deviance"    
## [13] "iter"              "weights"           "prior.weights"    
## [16] "df.residual"       "df.null"           "y"                
## [19] "converged"         "boundary"          "model"            
## [22] "call"              "formula"           "terms"            
## [25] "data"              "offset"            "control"          
## [28] "method"            "contrasts"         "xlevels"
```

So we can use this stored information in our calculations.

```r
with(fitl_1, null.deviance - deviance)
```

```
## [1] 445.5594
```

Is 445.6 small? How much smaller is enough? This value has a chi-square distribution, and its significance can be easily computed. For this computation, we need to know the degrees of freedom for the model (which equal the number of predictors in the model) and can be obtained like this:

```
with(fitl_1, df.null - df.residual)
```

```
## [1] 4
```

Finally, the p-value can be obtained using the following code to invoke the Chi-Square distribution:

```
#When doing it yourself, this is all you really need
#(we present the code in a separate fashion above so that you understand better what t
with(fitl_1, pchisq(null.deviance - deviance, df.null - df.residual, lower.tail = FALS
```

```
## [1] 3.961177e-95
```

We can see that the model chi-square is highly significant. Our model as a whole fits significantly better than a model with no predictors.

Menard (2010) recommends also looking at the likelihood ratio R^2, which can be calculated as the difference between the null deviance and the residual deviance divided by the null deviance.

```
#Likelihood ratio R2
with(fitl_1, (null.deviance - deviance)/null.deviance)
```

```
## [1] 0.09328629
```

Some authors refer to this as the Hosmer/Lemeshow R^2. It indicates how much the inclusion of the independent variables in the model reduces variation, as measured by the null deviance. It varies between 0 (when our prediction is catastrophically useless) and 1 (when we predict with total accuracy). There are many other pseudo R^2 measures that have been proposed, but Menard based on research on the properties of various of these measures recommends the likelihood ratio R^2 because:

- It is the one with a closer conceptual link to R^2 in OLS regression.
- It does not appear to be sensitive to the base rate (the proportion of cases that have the attribute of interest) of the phenomenon being studied and, therefore, will work even in cases with unbalanced probabilities.
- It varies between 0 and 1
- And it can be used in other generalised linear models (models for categorical outcomes with more than two levels, which we don't cover here)

2.1.3 Assessing model fit: ROC curves

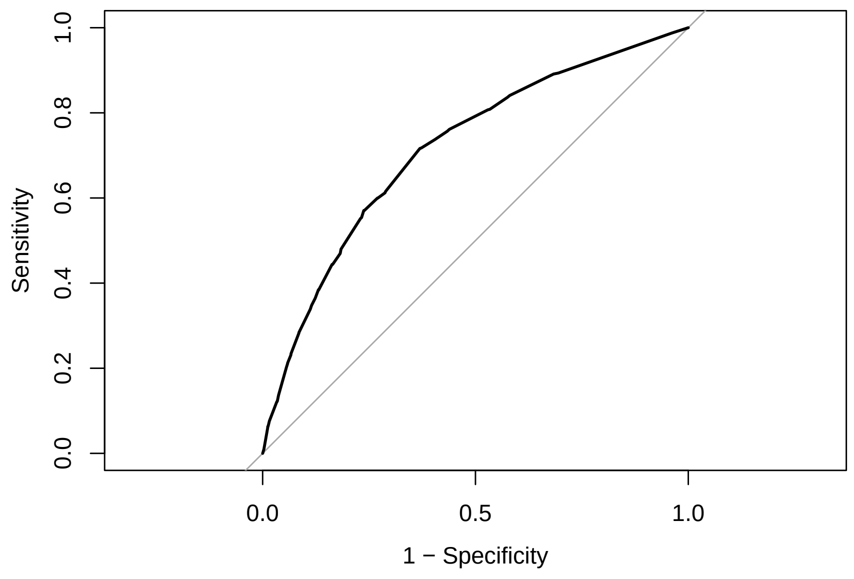
We may want to see what happens to sensitivity and specificity for different cut-off points. For this, we can look at **receiver operating characteristics** or simply ROC curves. This is essentially a tool for evaluating the sensitivity/specificity trade-off. The ROC curve can be used to investigate alternate cut-offs for class probabilities.

We can use the `pROC` package for this. We start by creating an object that contains the relevant information with the `roc()` function from the `pROC` package.

```
fitl_1_prob <- predict(fitl_1, type = "response")
library(pROC)
rocCURVE <- roc(response = Arrests$harsher,
                predictor = fitl_1_prob)
```

Once we have the object with the information, we can plot the ROC curve.

```
plot(rocCURVE, legacy.axes = TRUE) #By default, the x-axis goes backwards; we can use the specif
```



We can see the trajectory of the curve is at first steep, suggesting that sensitivity increases at a greater pace than the decrease in specificity. However, we then reach a point at which specificity decreases at a greater rate than the sensitivity increases. If you want to select a cut-off that gives you the optimal cut-off point, you can use the `coords()` function of the pROC package. You can pass arguments to this function so that it returns the best sum of sensitivity and specificity.

```
alt_cutoff1 <- coords(rocCURVE, x = "best", best.method = "closest.topleft")
#The x argument, in this case, is selecting the best cut-off using the "closest toplef
#(which identifies the point closest to the top-left part of the plot with perfect sen
alt_cutoff1
```

```
##   threshold specificity sensitivity
## 1 0.1696539   0.6305953   0.7163677
```

Here, we can see that with a cut-off point of .16 we get a specificity of .63 and a sensitivity of .71. Notice how this is close to the base rate of harsher treatment in the sample (17% of individuals actually received harsher treatment). For a more informed discussion of cut-off points and costs of errors in applied predictive problems in criminal justice, I recommend reading Berk (2012). Often, the selection of cut-off may be motivated by practical considerations (e.g., selecting individuals for treatment in a situation where resources to do so are limited).

The ROC curve can also be used to develop a quantitative assessment of the model. The perfect model is one where the curve reaches the top left corner of the plot. This would imply 100% sensitivity and specificity. On the other hand, a useless model would be one with a curve alongside the diagonal line splitting the plot in two, from the bottom right corner to the top right corner. You can also look at the **area under the curve** (AUC) and use it to compare models. An AUC of .5 corresponds to the situation where our predictors have no predictive utility. For a fuller discussion of how to compare these curves and the AUC, I recommend reading Chapter 11 of Kuhn and Johnson (2014).