Analysing social survey data using R

Pierre Walthéry, Nadia Kennar, Rihab Dahab, UK Data Service

UK Data Service

Table of contents

Introduction

This guide provides an introduction to analysing large scale social survey data using R, with examples from the British Social Attitudes Survey 2017. It is aimed at two categories of users:

1. Those inside or outside higher education, or who do not have access to commercial statistical software such as Stata, SPSS or SAS but who would like to conduct their own analysis beyond what is usually published by data producers: for example statistics for specific groups of the population. This guide provides this group of users with a range of procedures that will help them produce straightforward and robust analyses tailored to their needs without spending unnecessary time on learning the inner workings of R.
2. More advanced users who are already familiar with other data analysis tools but who would like to learn how to carry out their analyses in R.

A wealth of introductory content on R programming is already available online. The guide therefore focuses on providing succinct examples of common operations that most users of social surveys carry out in the course of their research, including how to:

- read in and open datasets.
- do common data manipulation operations.
- produce simple descriptive statistics or tabulations.
- handle survey weights and survey design variables

1 What is R ?

R is a free, user developed, object-oriented statistical programming language. It originates in the 'S' and 'S Plus' languages developed in the 1970s and 1980s. It has been widely used in science, statistics and data sciences and is increasingly being adopted in the social sciences for teaching and research purposes.

R can be downloaded from the Comprehensive R Archive Network (CRAN) website. Installation instructions as well as guides, tutorials and FAQ are available on the CRAN website. A considerable amount of R-related resources is available online.

Anyone can install and use R without charge, and to some extent contribute and amend the existing program itself. R is particularly favoured by users who want to develop their own statistical functions or implement technical advances that are not yet available in commercial packages. The existence of a vast number of user written packages (21964 at the time of writing this guide) is one of the strengths of R. Users interested in publishing their own packages on CRAN should be aware that a minimum set of rules need to be followed by contributors.

Although R can perform most of the analyses available in traditional statistical software such as Stata, SPSS, or SAS, it has broader applications used for mapping, data mining or machine learning. Its flexibility as a language allows users to carry out analyses in multiple ways, each with distinct advantages and disadvantages. Also, users can easily produce publication quality output in R using Markdown (now Quarto) and LaTeX document presentation systems. In addition, R graphs can easily be imported into html, MS Word or LibreOffice documents.

By contrast with other statistical software, the R interface is minimal and consists of a terminal. Similar to languages such as Python or C, R users can access it via a programming interface or Integrated Development Environment (IDE). In this guide we will use R Studio, one of the most common IDEs for R.
The data used in this introduction is the British Social Attitudes Survey, 2017, Environment and Politics: Open Access Teaching Dataset, which can be downloaded from the UK Data Service website without registration. The website also has instructions on how to acquire and download large-scale survey datasets. Links and further information about the other training resources available online are provided at the end of this document.

Although R has advantages over other statistical analysis software, it also has a few downsides, both of which are summarised below. Users should be reminded that as open-source software, R and its packages are mainly developed by volunteers, which makes it a very flexible and dynamic project, but at the same time reliant on developers' free time and goodwill.

Table 1.1: Advantages and downsides of R

Pros	Cons
R is free and allows users to perform almost any analysis they want.	The learning curve may be steep for users who do not have a prior background in statistics or programming.

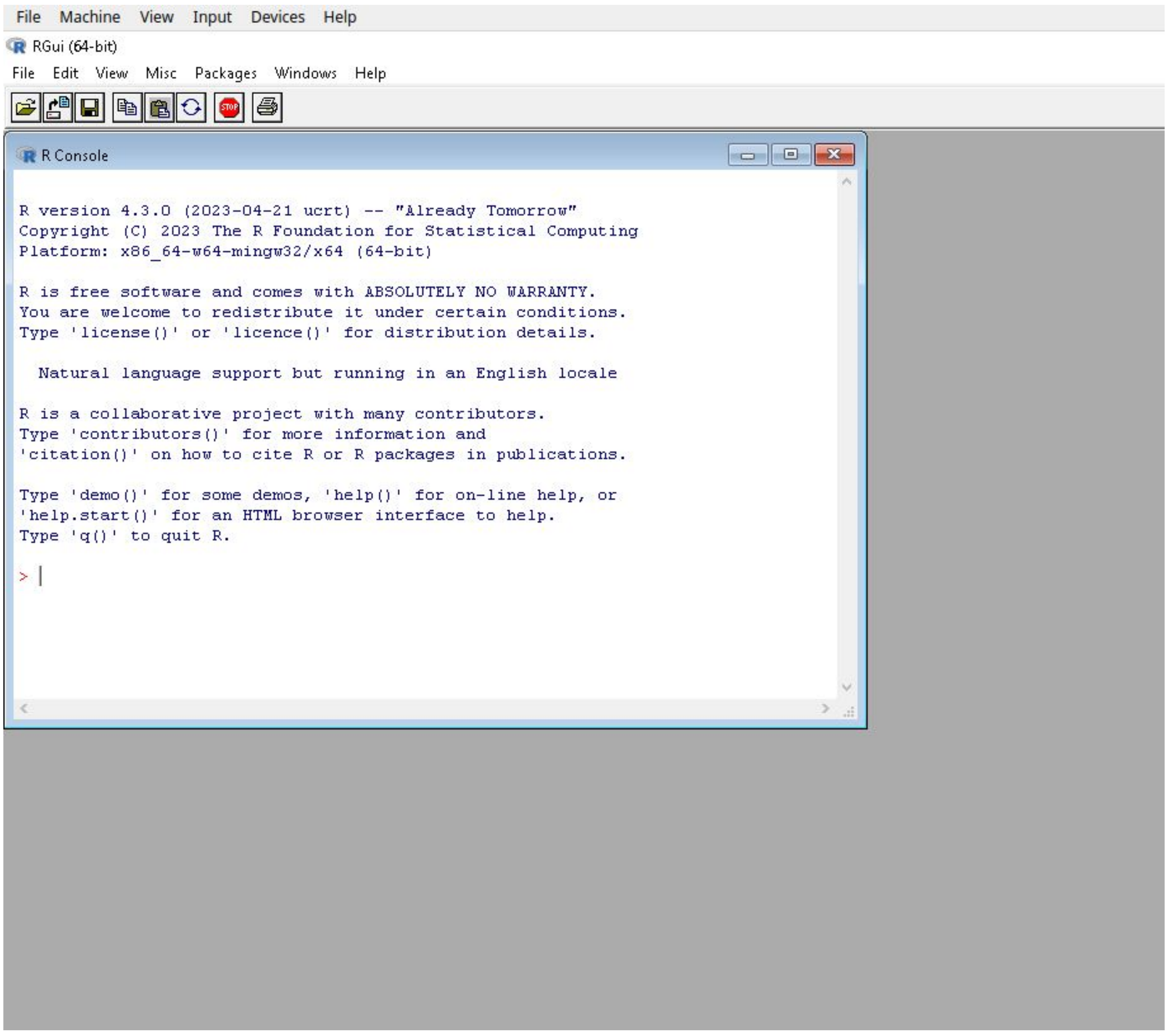
Pros	Cons
R puts statistical analysis closer to the reach of lay people rather than specialists.	
Transparency of use and programming of the software and its routines, which improves the peer-reviewing and quality control of the software.	
Very flexible.	Problem solving (for both advanced users and beginners) may be time-consuming, depending on how common the problem encountered, and may lead to more time spent solving technical rather than substantive issues.
Availability of a wide range of advanced techniques not provided in other statistical software	Many people who design R packages are, or will become busy academics. Packages can stop being maintained without notice.
A very large user base provides abundant documentation, tutorials, and web pages.	

There are several (sometimes many) ways of achieving a particular result in R. This can be confusing for novice researchers, but at the same time will allow users to tightly adjust their programmes to their needs.

2 Using R: essential information

2.1 Download and installation

R can be downloaded for free from the CRAN website, installed and run like any other application. Versions for Windows, Mac and Linux are available. The standard and rather minimalist interface that appears when the programme is launched on Windows is shown below.


File Machine View Input Devices Help
RGui (64-bit)
File Edit View Misc Packages Windows Help
R Console

```
R version 4.3.0 (2023-04-21 ucrt) -- "Already Tomorrow"
Copyright (C) 2023 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.

  Natural language support but running in an English locale

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.

> |
```

Figure 2.1: The standard R Windows interface

This interface merely allows the user to type in commands one by one in the console, and to install packages via pull-down menus. This setup is however rather minimal, not very ergonomic or user friendly. As with other statistical software, the preferred way of interacting with R for most users consist in writing code in separate files (also called scripts files) that are run whenever needed, which is not directly feasible with the standard R GUI.

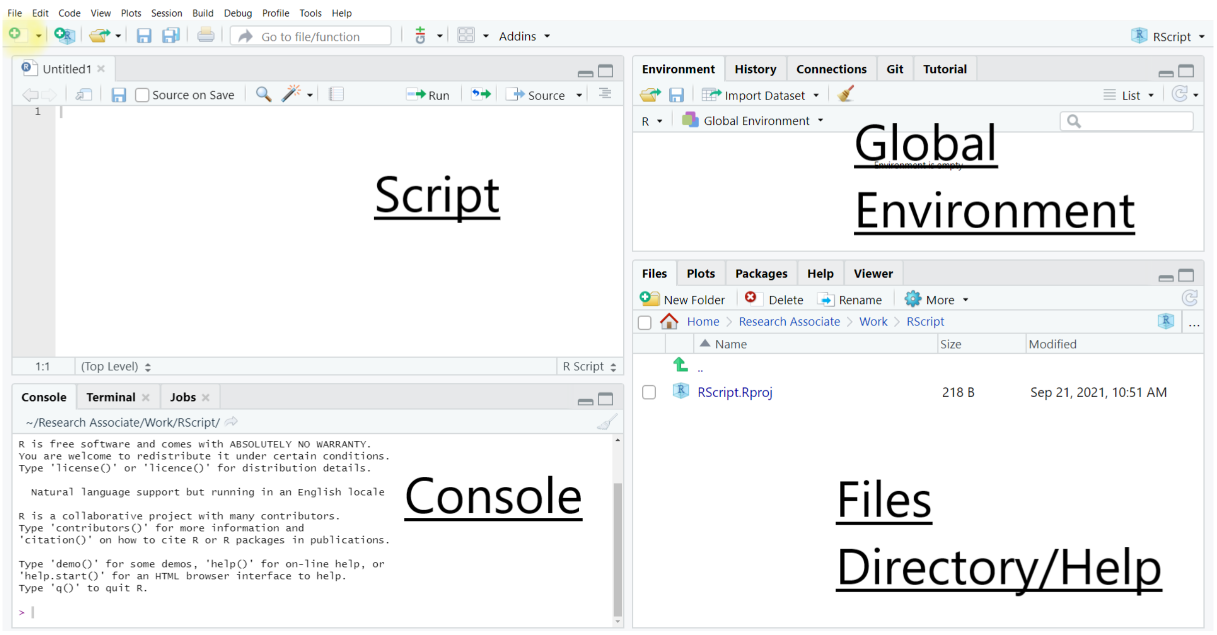
Writing R code and running script files are made easier via an Integrated Development Environment (IDE) such as RStudio for beginners to intermediate users Sublime Text, or the StatEt module for Eclipse for more advanced programmers. All are free, available for Windows, MacOS and Linux and offer users a large number of functionalities, such as syntax highlighting, integration with Github and Markdown/Quarto, and document previsualisation.

Since RStudio has a large user base and is relatively easy for beginners, we will use it throughout this guide. However, in order for the guide to remain accessible to users who do not work with other IDEs, RStudio is used below as a mere interface to the R engine without relying on its advanced features.

2.2 Installing and setting up RStudio

RStudio needs to be installed separately from R. The program can be downloaded from the RStudio website. The site will automatically generate a link to the version most compatible with the computer used to access it. Once downloaded double click on the file and follow the installation instructions.

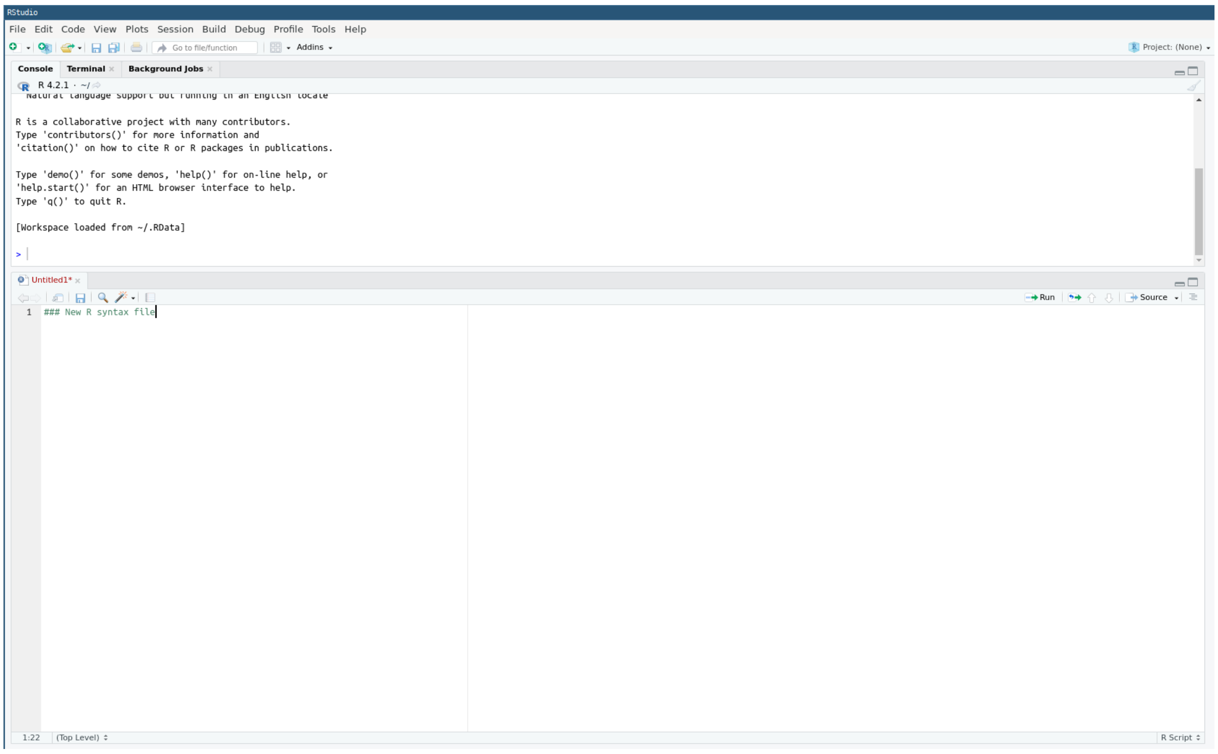
By default, the R Studio interface consists of four main panels, respectively known as the Script Editor (top left panel), the Console (bottom left panel), the Environment (top right panel) and the File/Directory/Help/Viewer (bottom right panel).



```
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.

  Natural language support but running in an English locale

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.

>
```

Figure 2.2: The R Studio default interface

As this rather complex interface can be visually overwhelming for some users and is not required for the purpose of this guide, we will minimise the Global Environment and Files/Directory/Help panels by clicking in the center of the window and dragging right to the edge of the screen. This way, only the script and console panels remain visible. The tiling of the panels can also be customised in `Tools>Global Options>Pane Layout`. For instance, Script can be moved to the bottom of the window and Console to the top:



```
  Natural language support but running in an English locale

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.

[Workspace loaded from ~/.RData]

>
```

```
### New R syntax file
```

Figure 2.3: A customised R Studio interface

2.3 Interacting with R

As already mentioned, one can type R commands directly in the console of RStudio and/or by writing sequences of commands in a script file.

Most R commands – also known as functions in R jargon – adopt the following syntax:

```
> command(parameter1, parameter2, ...)
```

All R commands are followed by brackets, even if some of them take no parameters.

In the following example we are going to set up a default working directory, that is the default location where files will be opened from and saved to in the , by using the `getwd()` and `setwd()` commands. First, let us visualise the current default working directory.

```
getwd()
```

```
[1] "/home/mscsepw2/Documents/R_UKDS"
```

Let us say that all the files used in this guide arre going to be located in a folder called ‘R_UKDS’, inside ‘My Documents’. To tell R to use the folder ‘R_UKDS’ we type:

For Windows:

```
> setwd("C:/Documents and Settings/<INSERT YOUR USERNAME HERE>/My Documents/R_UKDS")
```

For Mac:

```
> setwd("/Users/<INSERT YOUR USERNAME HERE>/Documents/R_UKDS")
```

For Linux:

```
setwd("~/Documents/R_UKDS")
```

Typing `getwd()` confirms that the change has been recorded.

```
getwd()
```

```
[1] "/home/mscsepw2/Documents/R_UKDS"
```

Notes:

- Any character string that is neither a command or the name of an object (such as a variable name) needs to be put between inverted commas or quotation marks, otherwise it will be interpreted as the name of an object and R will return an error.
- see the example below about loading user-created packages;
- Even when no parameters are specified for a command, brackets are compulsory as shown in the `getwd()` example above;
- R uses forward slashes rather than backslashes (unlike Windows applications, but like Linux) to separate directories. Using backlashes will return an error message;
- Although most R commands accept a large number of options to be specified. In many cases default values have been 'factory set' so that only the essential parameters need specifying.

Being object-oriented, the output of most R commands can be either directly displayed on the screen (as in the above example) or stored in objects that can be subsequently reused in further commands. This feature separates R from traditional statistical software.

For instance, typing:

```
a<-getwd()
```

will store the output of `getwd()` (that is, the name of the current default directory) into an object called 'a'. In order to view the content of a, one can just type its name:

```
a
```

```
[1] "/home/mscsepw2/Documents/R_UKDS"
```

Writing R scripts via R Studio

Most users will want to write their code in a script file, similar to the 'do' file in Stata or syntax file in SPSS. R script files can be identified with their .R suffix. To open an existing R script in RStudio select `File>Open File` (shortcut: Control+O) then the relevant script file. To create a new script select `File>New File>R Script` (shortcut: Control+Shift+N) this will open a new script window in which to type commands.

2.4 Installing and loading packages

Apart from a basic set of commands and functions, most of the tools offered by R are available in packages that need to be installed and downloaded separately from within R. This can also be done with the pull-down menus of RStudio.

For example, to install the 'foreign' package one need to type:

```
install.packages("foreign",repos = "https://cloud.r-project.org")
```

If the address of the package repository is not specified via the `repos=` option, a pull-down menu will appear, asking for one. Choosing `https://cloud.r-project.org` will automatically select the closest mirror site. Packages installation only needs to be done once.

Originally, `foreign` enabled users to import Stata (version 12 or older) or SPSS datasets. For Stata datasets saved under version 13 and above as well as SPSS datasets from version 16 onwards, the `haven` package is required.

```
install.packages("haven",repos = "https://cloud.r-project.org")
```

To use a package already installed in the local R library, the `library()` command is needed:

```
library(haven)
```

Simply typing:

```
> library()
```

Will list all packages installed on the computer that can be loaded in memory. This can be a rather long list!

In addition to the main archive of R packages, the CRAN website provides a series of manuals, including Writing R Extensions, which describes how users can write their own packages and submit them to CRAN.

Once a package is installed, it will be permanently stored in the local R library on the computer, unless it is deleted it with the `remove.packages()` command (this is not advised as this can break dependencies between packages!).

```
> remove.packages("name of the package")
```

Packages required for an analysis have to be loaded every time the programme is launched or a new R session is started (But not every time a syntax file is run!).

2.5 Using R’s internal help system

Within R, the most straightforward way to request help with a command consists of a question mark followed by the command name, without a space in between. The standard help system in R (unless using RStudio or Eclipse) relies on the default web browser installed on your computer (ie Chrome, Firefox or Edge in most cases) to display pages.

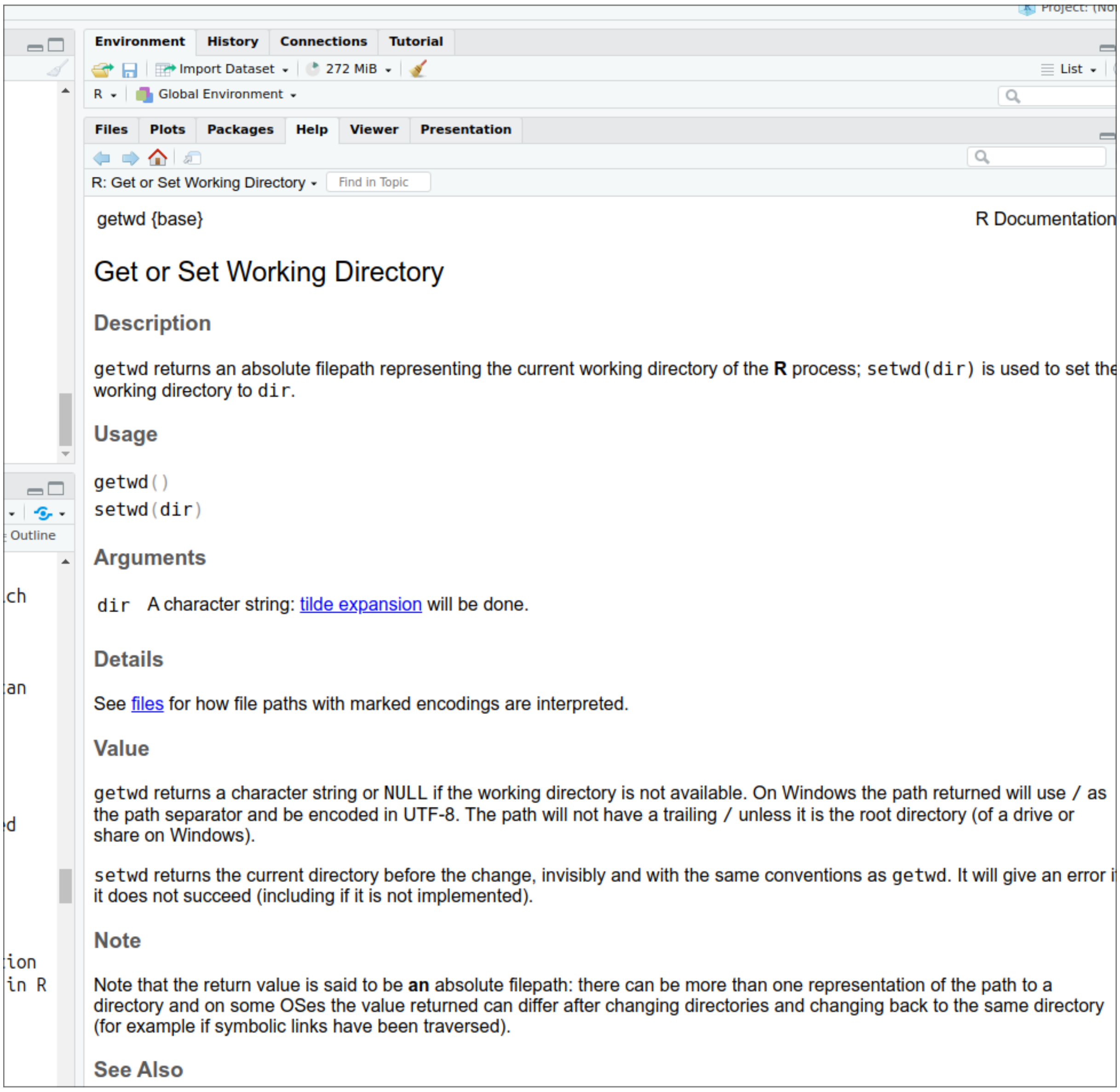
Typing:

```
> ?getwd
```

Is equivalent of:

```
help('getwd')
```

and will open the help page for the `getwd()` command in the default web browser or the viewer tab of RStudio.


Environment History Connections Tutorial

Import Dataset 272 MiB List

R Global Environment

Files Plots Packages Help Viewer Presentation

R: Get or Set Working Directory Find in Topic

getwd {base} R Documentation

Get or Set Working Directory

Description

getwd returns an absolute filepath representing the current working directory of the **R** process; `setwd(dir)` is used to set the working directory to `dir`.

Usage

```
getwd()
setwd(dir)
```

Arguments

`dir` A character string: tilde expansion will be done.

Details

See files for how file paths with marked encodings are interpreted.

Value

`getwd` returns a character string or `NULL` if the working directory is not available. On Windows the path returned will use `/` as the path separator and be encoded in UTF-8. The path will not have a trailing `/` unless it is the root directory (of a drive or share on Windows).

`setwd` returns the current directory before the change, invisibly and with the same conventions as `getwd`. It will give an error i it does not succeed (including if it is not implemented).

Note

Note that the return value is said to be **an** absolute filepath: there can be more than one representation of the path to a directory and on some OSes the value returned can differ after changing directories and changing back to the same directory (for example if symbolic links have been traversed).

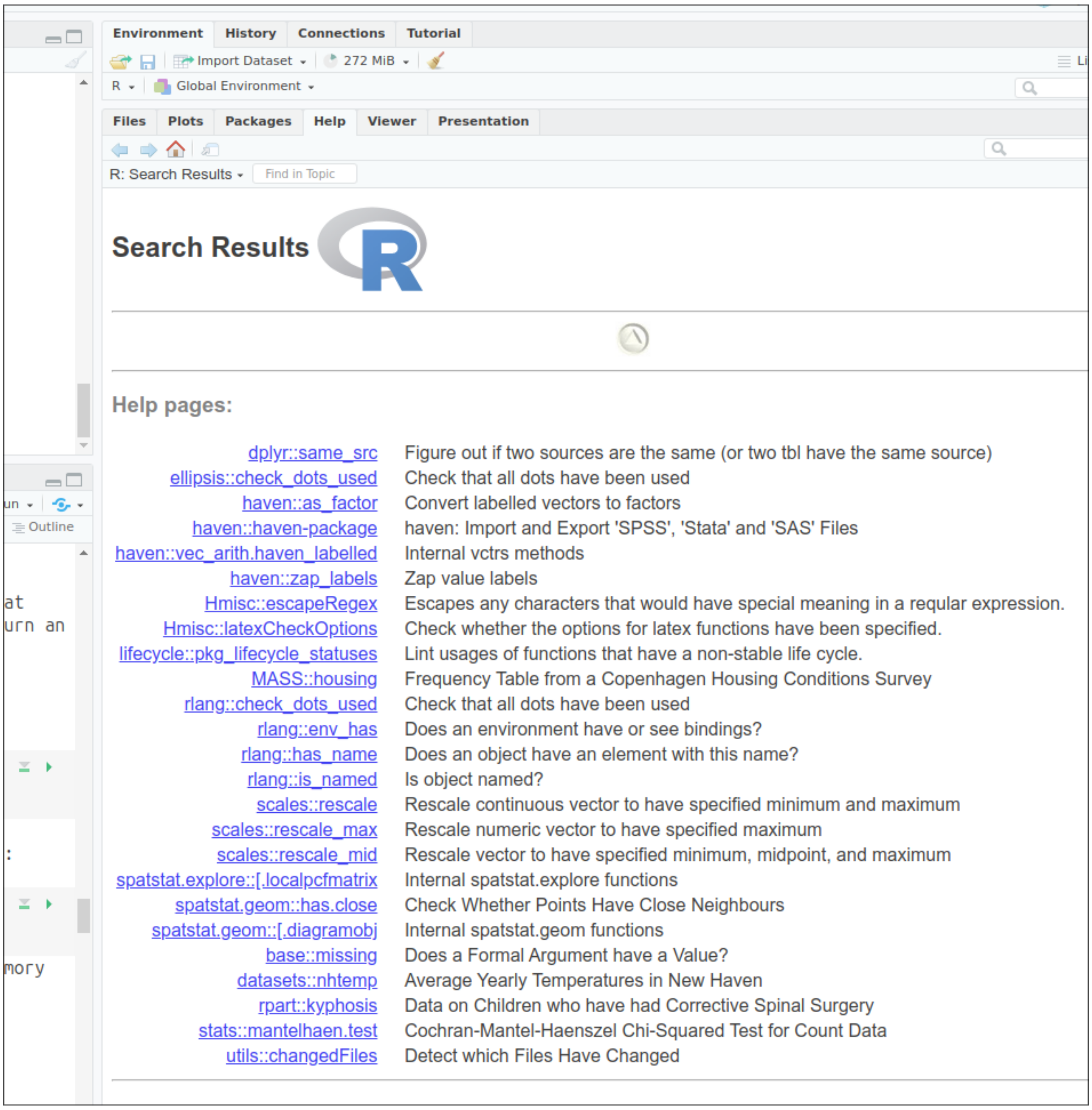
See Also

Figure 2.4: Help page for getwd()

This will work for any command directly available in the `Base` package that is loaded at startup or in other packages loaded via the `library()` command. Otherwise, R will return an error message.

Typing two question marks followed by a keyword will search all of R for the available documentation for that keyword in the installed packages:

```
??haven
```



Search Results

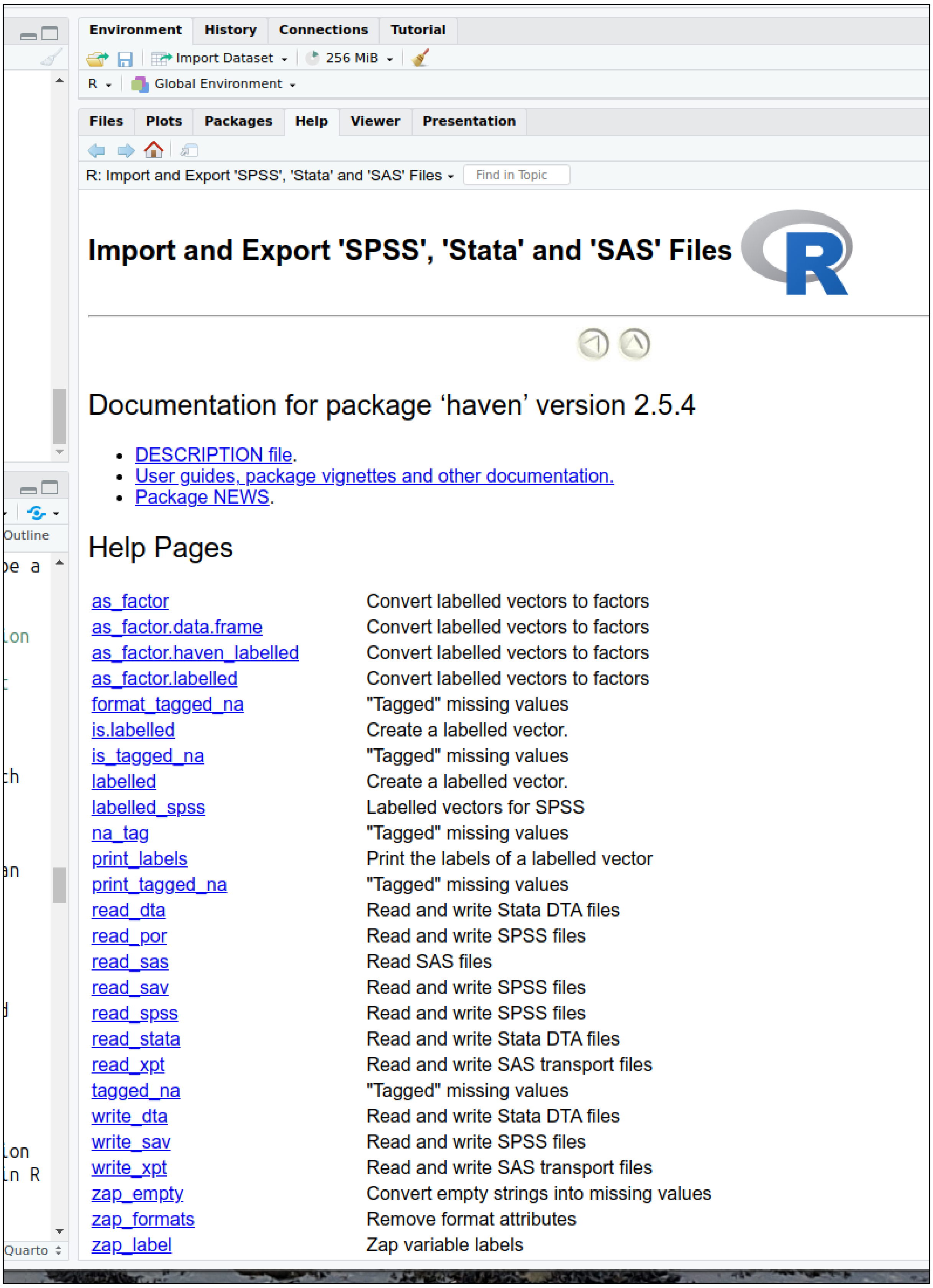
Help pages:

dplyr::same_src	Figure out if two sources are the same (or two tbl have the same source)
ellipsis::check_dots_used	Check that all dots have been used
haven::as_factor	Convert labelled vectors to factors
haven::haven-package	haven: Import and Export 'SPSS', 'Stata' and 'SAS' Files
haven::vec_arith.haven_labelled	Internal vctrs methods
haven::zap_labels	Zap value labels
Hmisc::escapeRegex	Escapes any characters that would have special meaning in a reqular expression.
Hmisc::latexCheckOptions	Check whether the options for latex functions have been specified.
lifecycle::pkg_lifecycle_statuses	Lint usages of functions that have a non-stable life cycle.
MASS::housing	Frequency Table from a Copenhagen Housing Conditions Survey
rlang::check_dots_used	Check that all dots have been used
rlang::env_has	Does an environment have or see bindings?
rlang::has_name	Does an object have an element with this name?
rlang::is_named	Is object named?
scales::rescale	Rescale continuous vector to have specified minimum and maximum
scales::rescale_max	Rescale numeric vector to have specified maximum
scales::rescale_mid	Rescale vector to have specified minimum, midpoint, and maximum
spatstat.explore::[.localpcfmatrix	Internal spatstat.explore functions
spatstat.geom::has.close	Check Whether Points Have Close Neighbours
spatstat.geom::[.diagramobj	Internal spatstat.geom functions
base::missing	Does a Formal Argument have a Value?
datasets::nhtemp	Average Yearly Temperatures in New Haven
rpart::kyphosis	Data on Children who have had Corrective Spinal Surgery
stats::mantelhaen.test	Cochran-Mantel-Haenszel Chi-Squared Test for Count Data
utils::changedFiles	Detect which Files Have Changed

Figure 2.5: Results of help search for ‘haven’

An index of all commands and functions in the `haven` package can be obtained by typing:

```
help(package='haven')
```



Import and Export 'SPSS', 'Stata' and 'SAS' Files

Documentation for package ‘haven’ version 2.5.4

- DESCRIPTION file.
- User guides, package vignettes and other documentation.
- Package NEWS.

Help Pages

as_factor	Convert labelled vectors to factors
as_factor.data.frame	Convert labelled vectors to factors
as_factor.haven_labelled	Convert labelled vectors to factors
as_factor.labelled	Convert labelled vectors to factors
format_tagged_na	"Tagged" missing values
is.labelled	Create a labelled vector.
is_tagged_na	"Tagged" missing values
labelled	Create a labelled vector.
labelled_spss	Labelled vectors for SPSS
na_tag	"Tagged" missing values
print_labels	Print the labels of a labelled vector
print_tagged_na	"Tagged" missing values
read_dta	Read and write Stata DTA files
read_por	Read and write SPSS files
read_sas	Read SAS files
read_sav	Read and write SPSS files
read_spss	Read and write SPSS files
read_stata	Read and write Stata DTA files
read_xpt	Read and write SAS transport files
tagged_na	"Tagged" missing values
write_dta	Read and write Stata DTA files
write_sav	Read and write SPSS files
write_xpt	Read and write SAS transport files
zap_empty	Convert empty strings into missing values
zap_formats	Remove format attributes
zap_label	Zap variable labels

Figure 2.6: Help content index for the *haven* package

Note: this command only works because the `haven` package was previously loaded in memory with the `library()` command.

More information about where to find help when using R is provided at the end of this document.

2.6 Objects

R is an object oriented language, which means that almost any information it processes is stored as 'objects' (i.e. containers) that can be manipulated independently. During an R session, multiple objects are available simultaneously (for instance datasets, but also summary tables or new variables produced from it). Typing:

```
> ls()
```

will list all the objects that are currently in memory.

Objects belong to `classes` or types which have distinct `properties`. There are many classes of objects in R. By comparison, Stata has only macros, variables and scalars that are directly available to most users. Common object classes include:

- factors: these are equivalent to categorical variables (see below);
- numeric: numerical variables – whether continuous or ordinal;
- character: alphanumeric variables;
- vectors: a list of items that are of the same type;
- lists: more complex list of objects, functions or datasets
- data frames (datasets);
- matrices, etc.

Some operations are understandably only possible with certain types of objects: for example, mathematical functions can only be used with numeric objects.

More advanced users can also create their own object classes. Describing R objects and their properties is well beyond the purpose of this guide and users interested should consult the online documentation for further explanations.

To create or assign a value to an object, one uses the assignment operator (<-). For example, we can assign the value 5 to an object called x.:

```
x <- 5
```

If you type the letter x, the value '5' will be returned in the console.

```
x
```

```
[1] 5
```

The object x will appear in the R environment after the ls() command.

```
ls()
```

```
[1] "x"
```

Deleting objects

The `rm()` function can be used to remove objects from the environment (session). These objects can be variables, lists, datasets, etc. For instance, to remove the object ‘x’, or the fictitious dataset called ‘mydata’:

```
rm(x)
ls()
```

```
character(0)
```

```
> rm(mydata)
```

Among the various classes of objects one may use in R, a few are essential to understand when analysing survey data. Their characteristics are briefly listed below;

Data frames

Data frames are typically the kind of objects in which survey datasets are stored. They are similar to datasets in traditional statistical software or worksheets in Microsoft Excel or LibreOffice Calc. They are objects that have indexed rows and columns, both of which may have names. Columns correspond to variables and lines or rows to observations. Any cell can be uniquely identified by its position.

Let’s assume that we have a small data frame called ‘mydata’. Here are a few basic commands to examine it:

Determining the size of a data frame:

the `dim()` command returns the number of rows and columns of a data frame

```
dim(mydata)
```

```
[1] 50  6
```

R tells us that our data is made of 50 rows and 6 columns, in other words of 50 observations and 6 variables.

What if I wanted a quick overview of the dataset?

```
head(mydata)
```

```
    RSex skipmeal                   Married
1 Female       NA Married/living as married
2 Female        1        Separated/divorced
3 Female       NA Married/living as married
4   Male       NA             Never married
5   Male        1             Never married
6   Male       NA Married/living as married
```

```
        Poverty1                         HEdQual3
1        Was not Higher educ below degree/A level
2        Was not Higher educ below degree/A level
3        Was not             O level or equiv/CSE
4        Was not Higher educ below degree/A level
5        Was not                 No qualification
6 Was in poverty                             <NA>
  NatFrEst
1        5
2       30
3       50
4       50
5       50
6       10
```

The `head()` command displays the first six lines of the dataset. Depending on the number of variables the output of `head()` can become quickly overwhelming, as the size of the lines on most screens is limited!

Obtaining the names of variables (or columns) in the dataset: This can be done using either `ls()` which we already have used, or the `names()` commands. `ls()` returns the variables names, sorted alphabetically, whereas `names()` returns them in their actual order in the data frame.

```
ls(mydata)
```

```
[1] "HEdQual3" "Married"  "NatFrEst" "Poverty1"
[5] "RSex"     "skipmeal"
```

```
names(mydata)
```

```
[1] "RSex"     "skipmeal" "Married"  "Poverty1"
[5] "HEdQual3" "NatFrEst"
```

We can see that in the data frame, the “RSex” column comes in fact before “Poverty1”.

Accessing variables:

Variables (or columns) of a data frame can be accessed by their name preceded by the $ sign. For example:

```
mydata$NatFrEst
```

```
 [1]  5 30 50 50 50 10  1 NA  5 50 60 25 30  3 25 10 80
[18]  5 50 50 40 45  2 82 60 40 10 10 40 15 30 30  2 10
[35] 75 99 70 50 10 30  1 10 85 45 70 25 40 30 10 25
attr(,"value.labels")
named numeric(0)
```

Lists all values of `NatFrEst`, as well as additional technical information at the bottom.

Alternatively, variables (columns)and observations (rows) can be identified numerically by their position in the data frame using square brackets:

```
dataframe[row number,column number]
```

Given that `RSex` is the first column of our dataset, typing

```
mydata[,1]
```

```
 [1] Female Female Female Male   Male   Male   Male  
 [8] Female Male   Male   Male   Female Female Male  
[15] Female Female Female Female Female Female Male  
[22] Female Female Male   Male   Female Female Female
[29] Female Male   Female Female Female Female Male  
[36] Male   Female Female Male   Female Male   Male  
[43] Female Male   Female Male   Female Female Female
[50] Male  
Levels: Male Female
```

Returns the same output as previously, that is all recorded observation of `NaRfEst`. Not specifying a row or column name within the square brackets tells R to display them all.

```
mydata[6,]
```

```
  RSex skipmeal                   Married
6 Male       NA Married/living as married
        Poverty1 HEdQual3 NatFrEst
6 Was in poverty     <NA>       10
```

Returns the values of all the variables for the sixth row of the data frame. Specifying both a row and column number, will return a unique observation:

```
mydata[6,6]
```

```
[1] 10
```

which in this case is 10. Finally, more than one column or row can be displayed by concatenating their number using the `c()` function:

```
mydata[c(6,9), 6]
```

```
[1] 10  5
```

The above command returns respectively the sixth and 9th observations for the sixth column. Explicit columns names can be used instead of their number, provided that they are placed between inverted commas:

```r
mydata[c(6,9),'NatFrEst']
```

```
[1] 10  5
```

Returns the same result as above.

We can explore other types of objects commonly found in R using the same dataset. The type of a variable can be displayed by simply using the `class()` function.

Numeric

`Numeric` objects are simple numerical vectors (ie a single or a list of numbers). They can be standalone or part of a data frame. Such is for example the case of the variable `NatFrEst`, which measures the proportion of people making wrong benefits claims estimated by respondents .

```r
class(mydata$NatFrEst)
```

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

Character

Character objects are alphanumeric vectors, that is variables which consist of text string(s).

```r
class(mydata$Married)
```

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

```r
head(mydata$Married) ### Let's look at the first five rows of Married
```

```
[1] "Married/living as married"
[2] "Separated/divorced"
[3] "Married/living as married"
[4] "Never married"
[5] "Never married"
[6] "Married/living as married"
```

Factors

A distinctive feature of R is that categorical variables whether ordinal or polynomial are stored in objects known as **factors**. Factors should be thought of as a special type of variable with a discrete set of values, known as `levels`. Factors can be unordered or ordered. In our data, `Rsex` (Gender of the respondent) is such an object:

```
class(mydata$RSex)
```

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

The main difference between factors and traditional categorical variables in Stata or SPSS is that they do **not** consist of arbitrary numerical values with which substantive value labels are associated.

It is always a good idea to check the ordering of factor levels in a newly created variable with the `level()` command

```
levels(mydata$RSex)
```

```
[1] "Male"   "Female"
```

returns the levels of `RSex`. 'Male' is the first level of `Rsex`, and 'Female' the second one, irrespective of the values originally assigned and described in the codebook of the dataset.

It is possible to change the ordering of factor levels with the `factor()` function.

```
mydata$RSex.n<-factor(mydata$RSex, # We re-use the existing factor levels from Rsex
                    levels = levels(mydata$RSex)[c(2,1)]) # and reorder them
```

The above code tells R to create a new factor variable -`RSex.n` - whose levels are identical to `RSex`, but with 'Female' (level number 2 in the original variable) coming first, and "Male" (level number 1) , second. The name of the new variable is arbitrary.

We can check the outcome:

```
levels(mydata$RSex.n)
```

```
[1] "Female" "Male"
```

In the next chapter an alternative approach to dealing with categorical variables imported from SPSS or Stata is presented.

3 Opening datasets in R

3.1 Importing files: essential information

Spreadsheet and text files

In principle, any dataset whether in CSV, or spreadsheet format can be imported into R:

- CSV files can be directly imported with read.csv() from Base R.
- MS Excel spreadsheets can be opened with the `read_excel()` function from the readXL package or opened/written with `read.xlsx()` and `write.xlsx()` from the xlsx package.
- ODS (Open Document Spreadheets) files from LibreOffice/OpenOffice Calc can be opened and written with `read_ods()/write_ods()` from the readODS package.

SAS, SPSS, or Stata

Currently, the `haven` packages provides the most versatile and straightforward route to importing data from other mainstream statistical software. SPSS, Stata and SAS files can be opened with respectively `read_spss()`, `read_dta()` and `read_sas()`. The only potential downside is that it relies on ad hoc data formats and data structures for converting labelled categorical variables and attempts to mimic SPSS/Stata's value and variable labels. More information is available here. We will be using `haven` throughout this guide.

For the record, the `foreign` package used to be the standard for importing MINITAB, SAS, SPSS and Stata datasets into R, but its development ceased in 2000 (Stata version 12).

3.2 The 2017 British Social Attitudes Survey

In the remainder of this guide, we will be using the *British Social Attitudes Survey, 2017, Environment and Politics: Open Access Teaching Dataset*, which can be downloaded from the UK Data Service website. We will import the SPSS version of the dataset, We will import the SPSS version of the dataset. We will assume that the data is saved in a folder named `UKDA` inside the `Documents` folder (on a Windows computer). We will set . `C:\\Users\\Your_User_Name_Here\\Documents` as the default working directory.

This way, we won't have to specify the full path each time that we will be opening or saving file.

```
setwd("C:\\Users\\Your_User_Name_Here\\Documents\\UKDS")
```

We can finally open the file:

```
bsa<-read_spss("8849spss_V1/bsa2017_open_enviropol.sav"
               )
```

Typing:

```
ls()
```

```
[1] "bsa"
```

will show us that the object ‘bsa’ has appeared in the environment.

3.3 Understanding the dataset

As mentioned in earlier sections, we can find out the number of observations and variables in the dataset by typing the following:

```
dim(bsa)
```

```
[1] 3988   25
```

We can see that there are 3988 observations and 25 variables in the dataset.

What if we want to get the list of all variables in the dataset? We need to type:

```
ls(bsa)
```

```
 [1] "actchar"           "actpol"            "carallow"          "carenvdc"
 [5] "carnod2"           "carreduc"          "cartaxhi"          "CCBELIEV"
 [9] "ChildHh"           "eq_inc_quintiles" "govnosa2"          "HEdQual3"
[13] "leftrigh"          "libauth"           "Married"           "PartyId2"
[17] "plnenvt"           "plnuppri"          "Politics"          "RAgeCat"
[21] "RClassGp"          "Rsex"              "Sserial"           "Voted"
[25] "WtFactor"
```

If we want to get a better sense of the data, we use the `head()` function which will return the first six rows.

```
head(bsa)
```

```
  Sserial Rsex RAgeCat Married ChildHh HEdQual3 eq_inc_quintiles RClassGp
1  290001    1       3       1       1        3                3        4
2  290002    2       7       2       2        2               NA        5
3  290003    2       4       1       1        1                3        1
4  290004    2       4       1       1        1                3        1
5  290005    1       7       4       2        2                4        1
6  290006    2       2       4       1        2                1        1
```

```
  CCBELIEV carallow carreduc carnod2 cartaxhi carenvdc plnenvt plnuppri
1        2       NA       NA      NA       NA       NA      NA       NA
2        3       NA       NA      NA       NA       NA      NA       NA
3       NA       NA       NA      NA       NA       NA      NA       NA
4       NA       NA       NA      NA       NA       NA      NA       NA
5       NA       NA       NA      NA       NA       NA      NA       NA
6       NA       NA       NA      NA       NA       NA      NA       NA
  Politics Voted actchar actpol govnosa2 PartyId2 leftrigh  libauth  WtFactor
1        3     1       5      5        2        2      1.0 4.500000 0.9380587
2        1     1      NA     NA        4        2      1.8 4.333333 0.6844214
3        3     1      NA     NA       NA        2       NA       NA 0.9060821
4        1     1       4      4        2        2      1.5 2.500000 1.3085513
5        4     1      NA     NA       NA        3      2.0 3.166667 0.4392823
6        4     1      NA     NA        3       NA      3.0 3.000000 0.5695720
```

Single variables may be also summarised with `head()`.

For example:

```
head(bsa$Rsex)
```

```
[1] Male   Female Female Female Male   Female
Levels: skipped or na Male Female Dontknow Refusal
```

displays the first six observations of `Rsex` (gender of respondents).

Simply typing the name of a variable as in:

```
bsa$Rsex
```

will list its first 1000 observations.

Other commands, such as `summary()` provide more synthetic information.

```
summary(bsa$Rsex)
```

```
skipped or na          Male        Female      Dontknow       Refusal
            0          1806          2182             0             0
```

`summary()` is a generic function that tailors the most appropriate output to an object class. Since `Rsex` is a categorical variable, the output of `summary()` is identical to what we would have obtained with the straightforward tabulation function `table()`:

```
table(bsa$Rsex)
```

```
skipped or na          Male        Female      Dontknow       Refusal
            0          1806          2182             0             0
```

When encountering a numeric variable, `summary()` will compute basic descriptive statistics (mean, median, quartiles, maximum and minimum). For example, in the case of the libertarian-authoritarian scale `libauth`:

```
summary(bsa$libauth)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
  1.167   3.000   3.500   3.511   4.000   5.000     775 
```

3.4 Identifying and selecting variables

As we have already seen, variables are objects. R automatically stores variables using the appropriate object class. Categorical variables are 'Factors' with 'Levels' as categories within these, while continuous variables are 'Numeric' types of objects. The `class()` displays the type of an object:.

```
class(bsa$Rsex)
```

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

The levels() function returns the categories of the variable.

```
levels(bsa$Rsex)
```

```
[1] "skipped or na" "Male"          "Female"        "Dontknow"     
[5] "Refusal"      
```

3.5 Factor variables in the haven package

As we have seen in the previous chapter, statistical software such as SPSS or Stata treat categorical variables as numerical variables. Values labels are then 'attached', allocating a substantive meaning to these values. On the other hand, R records categorical variables in objects that are called *factors*, that may be ordered or not. Factors consist of a limited number of *levels* sequentially numbered values (ranging from 1 to the number of levels) that are paired with arbitrary character strings. The number of a factor level is therefore distinct from the values categorical variables are allocated in codebooks.

Unfortunately, there are no straightforward ways to convert SPSS or Stata labelled categorical variables into R factors. The approach followed by the `haven` package is to preserve the arbitrary numeric values of the original variables, and add *attributes* that contain the value labels that can be manipulated separately. Attributes are a special type of R objects that have a name, and can be retrieved using the `attr()` function. `haven`-converted categorical variables all have a 'label' and 'labels' attribute. The former is the variable description aka variable label, the latter the value labels.

Let's examine the original variable description and value labels with the `attr()` function.

```
attr(bsa$HEdQual3, "label")
```

```
[1] "Highest educational qual obtained - dv"
```

We do the same with value labels:

```
attr(bsa$HEdQual3, "labels")
```

```
                           Degree Higher educ below degree/A level 
                                1                                2 
              O level or equiv/CSE                 No qualification 
                                3                                4 
                    DK/Refusal/NA 
                                8 
```

The value labels displayed above include both the text description and the original (i.e. codebook) numeric values from SPSS/Stata.

These `haven`-imported numeric variables can be converted into R factors using `as_factor()`, with their numeric values simply reflecting their order in the vector of levels. The factor levels in the converted variables can consist of either the value labels:

```
levels(
       as_factor(bsa$HEdQual3, levels="labels")
       )
```

```
[1] "Degree"                           "Higher educ below degree/A level"
[3] "O level or equiv/CSE"             "No qualification"                
[5] "DK/Refusal/NA"                   
```

… or, as in the earlier output, of both the original SPSS/Stata numeric values and the value labels:

```
levels(
       as_factor(bsa$HEdQual3, levels="both")
       )
```

```
[1] "[1] Degree"                          
[2] "[2] Higher educ below degree/A level"
[3] "[3] O level or equiv/CSE"            
[4] "[4] No qualification"                
[5] "[8] DK/Refusal/NA"                   
```

4 Essentials of Data Manipulation

In this section we will cover how to recode variables and deal with missing data.

4.1 Creating and transforming numeric variables

Let's say we would like to transform our numerical political orientation variable: `leftrigh` into a logarithmic scale. We can use the `log()` function which is available in R Base and simply returns the natural logarithm (base-e). We will use the assignment operator (<-) to create a new variable called 'lnleftright' from the original variable

```
bsa$lnleftrigh <- log(bsa$leftrigh)
```

Note that if we had not specified `bsa$` the command would have created a transformed variable outside of the BSA data frame. We can now check the results with `summary()`

```
summary(cbind(bsa$lnleftrigh,bsa$leftrigh))
```

```
       V1                V2
 Min.   :0.0000   Min.   :1.00
 1st Qu.:0.6931   1st Qu.:2.00
 Median :0.8755   Median :2.40
 Mean   :0.8707   Mean   :2.52
 3rd Qu.:1.0986   3rd Qu.:3.00
 Max.   :1.6094   Max.   :5.00
 NA's   :782      NA's   :782
```

It is not possible to pass several variables names directly to `summary()`. We need to group them first into a temporary object using `cbind()`. In the output `V1` refers to the first variable, `lnleftrigh`.

We can easily create completely new variables in the dataset. For instance, the following will create `test` with a constant value of 1.

```
bsa$test <- 1

summary(bsa$test)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      1       1       1       1       1       1
```

4.2 Recoding variables

Recoding categorical and numeric variables is very common in survey research. For example, let us create a dichotomic version of the marital status of BSA respondents. The original marital status is recorded by `Married`. Simply using `summary` will return descriptive statistics of its numeric values, which is not what we need here:

```
summary(bsa$Married)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
  1.000   1.000   1.000   1.977   3.000   4.000       1 
```

What we want instead is to work with the factor-converted version of `Married`.

```
summary(as_factor(bsa$Married))
```

```
Married/living as married         Separated/divorced                    Widowed 
                     2209                        530                        380 
            Never married             No information                       NA's 
                      868                          0                          1 
```

Even better, we can produce the exhaustive list of all of `Married`'s factor levels:

```
levels(as_factor(bsa$Married))
```

```
[1] "Married/living as married" "Separated/divorced"       
[3] "Widowed"                   "Never married"            
[5] "No information"           
```

We can now move on to creating a new variable called `Married2` where respondents are categorised into two new categories: 'Not partnered' and 'Partnered'. The "separated/divorced" and 'Never married' categories of the 'Married' variable are recoded as 'Not partnered'. It is always advised to create new variables when recoding old ones so the original data is not tampered with.

```
bsa$Married.f<-as_factor(bsa$Married,"labels")
bsa$Married2 <- ifelse(bsa$Married.f=="Married/living as married",
                                      "Partnered",bsa$Married)
bsa$Married2 <- ifelse(bsa$Married.f=="Widowed" |
                       bsa$Married.f=="Never married" |
                      bsa$Married.f=="Separated/divorced",
                                      "Not partnered",bsa$Married2)

bsa$Married2<-as.factor(bsa$Married2)
levels(bsa$Married2)
```

```
[1] "Not partnered" "Partnered"
```

```
summary(bsa$Married2)
```

```
Not partnered     Partnered           NA's
         1778          2209              1
```

The second and fourth categories have been renamed to ‘Not partnered’. Now we have two levels: ‘Partnered’ and ‘Not partnered’

`ifelse()` is a convenient tool when it is required to work with Base R functions only or when variables have a limited number of categories. The syntax consists of three terms:

1. the condition to be evaluated: for example `bsa$Married.f=="Widowed"`
2. what happens if the condition is met, for example the new variable takes the value “Not partnered”
3. what happens if the condition is not met, for example, the new variable retains its existing value

More complex cases may require a more advanced function. The `dplyr` library provides a comprehensive set of data manipulation tools, such as `case_when()`.

```
library(dplyr)
bsa<-bsa%>%
     mutate(Married3=case_when(
                Married.f == "Married/living as married" ~ "Partnered",
                Married.f == "Separated/divorced" |
                Married.f == "Widowed" ~ "Not Partnered",
                Married.f == "Never married" ~ "Not Partnered"
)
)
bsa$Married3<-as.factor(bsa$Married3)
summary(bsa$Married3)
```

```
Not Partnered     Partnered           NA's
         1778          2209              1
```

The syntax above created the `Married3` variable, which is identical to `Married2`. Let’s decompose it:

- `dplyr` use the `pipe` symbol ie `%>%` or `|>` which enables to sequentially combine functions. We will come back to this later in this guide.
- `mutate()` is the generic variable creation/alteration command, and can handle complex combinations of conditions as well as multiple simultaneous variable creation operations.
- `case_when()` is the function that allows recoding numerical, character, or factor variables. On the left hand side of the tilde `~` are the condition or the values that need to be matched in the original variable , and on the right hand side, the attributed ie recoded values in the new variable. Note that in this case, the recoded variable is by default a character object and needs to be converted into a factor for easier manipulation.

Extra tips:

- As with any data manipulation exercise, caution is required, and it is recommended to create new variables with the recoded value rather than alter an original variable when handling missing values.
- The standard value attribution command in R is `<-`. However, `=` will also work in many cases.
- Unless explicitly specified (in our case, by adding the bsa$ prefix to variable name), the objects created are not included in the data frame from which they were computed.

4.3 Missing Values

Explicit or system missing values in R (i.e. values that R itself considers as missing) are represented as `NA` for factors and numerical variables. For character variables, missing values are simply empty strings, ie `""`. R has a series of functions specifically designed to handle NAs.

R has fewer safety nets than other packages for handling missing values. Most functions won't warn users about whether there are observations with missing values that have been dropped. On the other hand, some commands will return error messages and by default won't run when missing values are present. This is the case of `mean()` for example.

4.3.1 Inspecting missing data

The logical function `is.na()` assesses each observation in variables and identifies whether cases are valid or missing. The result will appear as a boolean TRUE/FALSE vector for each observation. `is.na()` can be combined with other functions:

- With `table()` in order to get the frequencies of missing values of a specific variable.
- With `sum()` in order to count the number of missing observations of variables or whole datasets.

```
table(                         # number of missing values in the leftright variable
      is.na(bsa$leftrigh)
      )
```

```
FALSE  TRUE
 3206   782
```

```
sum(                           # of missing values in the whole dataset
   is.na(bsa)
   )
```

```
[1] 36593
```

```
mean(                          # proportion of NAs for a variable
     is.na(bsa$leftrigh)
     )
```

```
[1] 0.1960883
```

```
mean(                         # proportion of NAs in the dataset
     is.na(bsa)
     ) # returns the proportion in the dataset
```

```
[1] 0.3058592
```

4.3.2 Recoding missing values as NA (continuous variables)

It may sometimes be useful to recode implicit missing values (ie considered by the data producer as missing, but not by R) of either numeric objects or factors into `<NA>`, in order to simplify case selection when conducting analyses. This can either be done with Base R code or the more advanced data manipulation functions from the dplyr package that we explored earlier.

Let's assume for a moment that we would like to eliminate respondents aged under 25 for our analysis. A safe way to proceed is by creating a new dataset.

```
# convert labelled numeric variable into factor for clarity
bsa$RAgeCat.f <- as_factor(bsa$RAgeCat)
table(bsa$RAgeCat.f)
```

```

     18-24      25-34      35-44      45-54      55-59      60-64        65+
       223        591        650        729        320        333       1138
DK/Refused
         0
```

```
# retains all values except those that match the condition:
bsa.adults<-bsa[!bsa$RAgeCat.f=="18-24", ]
table(bsa.adults$RAgeCat.f)
```

```

     18-24      25-34      35-44      45-54      55-59      60-64        65+
         0        591        650        729        320        333       1138
DK/Refused
         0
```

We can also notice that although there are now no observations left in the 18-24 category, it is still displayed by `table()`. This is because levels are attributes of factors and are not deleted with observations. We can remove unused levels permanently with `droplevels()`

```
bsa.adults$RAgeCat.f<-droplevels(bsa.adults$RAgeCat.f)
table(bsa.adults$RAgeCat.f)
```

```

25-34 35-44 45-54 55-59 60-64   65+
  591   650   729   320   333  1138
```

4.3.3 Working with missing values

Explicit missing values (coded as NA) can be taken care of by R's own missing values functions. For instance using the `na.rm=T` or `na.rm=TRUE` option will remove missing values from an analysis (typing `?na.rm` will provide more information). Below is a summary of how NAs are dealt with by common R commands:

Table 4.1: Treatment of missing values by R commands

Command	Default action	Parameter
mean(), sd(),median()	Includes NA (may return an error)	na.rm=T
cor(),cov()	Includes NA (may return an error)	*use="complete.obs"*
table()	Excludes NA	*useNA = "always"* to display NAs
xtabs()	Excludes NA	*addNA = T* to display NAs
lm(),glm()	Excludes NA	*na.action=NULL*

4.4 Subsetting datasets

When analysing survey data. it is often necessary to limit the scope of computation to specific groups or subset of the data we may be interested in. There are many ways of subsetting datasets in R. We will review the most common here.

Using Base R

Most subsetting commands involve some form of conditions whereby the characteristics of a subsample of interest are specified. Suppose we would like to examine the interest for politics among people aged 18-24.

We can either create an adhoc data frame:

```
table(bsa$Politics)
```

```
   1    2    3    4    5
 739  982 1179  708  379
```

```
bsa.young<-bsa[bsa$RAgeCat.f=="18-24",]
table(bsa.young$Politics)
```

```
 1  2  3  4  5
29 41 72 56 25
```

Or we can directly limit the extent of the analysis on the go:

```
  table(
        bsa$Politics[bsa$RAgeCat.f=="18-24"]
        )
```

```
 1  2  3  4  5 
29 41 72 56 25 
```

In the first example it was necessary to include a comma after the condition. This is meant to indicate that we want to retain all variables ie columns in the dataset. The comma is not necessary in the second example as we are already working with a single variable.

Using dplyr*

Now suppose we want to further restrict the analysis to people self-identifying as males. We could use the same Base R syntax as above, but with more than one condition coding tends to become a bit cumbersome. We could instead use the more convenient syntax from the `dplyr` package. Either:

```
bsa.young.males<-bsa%>%
  filter(RAgeCat.f=="18-24" & as_factor(Rsex)=="Male")

table(as_factor(bsa.young.males$Politics))
```

```
Item not applicable   ... a great deal,        quite a lot,                some, 
                  0                  15                  22                   30 
    not very much,     or, none at all?          Don`t know              Refusal 
                 18                   9                   0                    0 
```

Or as before, embed it as a condition within `table()` :

```
  table(
         as_factor(
          bsa%>%
          filter(RAgeCat.f=="18-24" & as_factor(Rsex)=="Male")%>%
          select(Politics)
  )
  )
```

```
Politics
Item not applicable   ... a great deal,        quite a lot,                some, 
                  0                  15                  22                   30 
    not very much,     or, none at all?          Don`t know              Refusal 
                 18                   9                   0                    0 
```

`filter()` and `select()` are the functions that specify respectively rows and columns to be kept/removed. They can be combined or used independently and used in any order.

We are now equipped with the necessary information to move to the next stage and carry out basic analysis using R.

5 Descriptive statistics

5.1 Continuous variables

Producing descriptive statistics in R is relatively straightforward, as key functions are included by default in the Base package. We have already seen above that the `summary()` command provides essential information about a variable. For instance,

```
summary(bsa$leftrigh)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
   1.00    2.00    2.40    2.52    3.00    5.00     782 
```

provides information about the mean, median and quartiles of the political scale of respondents.

The `describe()` command from the `Hmisc` package provides a more detailed set of summary statistics.

```
library(Hmisc)
describe(bsa$leftrigh)
```

```
Error in proxy[, ..., drop = FALSE]: incorrect number of dimensions
```

The code above returns an error because `describe()` expects numeric values, and ‘leftrigh’ isn’t a pure numeric variable:

```
class(bsa$leftrigh)
```

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

It is a numeric variable: ‘double’, but with some extra metadata, including `haven`-generated value and variable labels. In order for `describe()` to run properly, we need to convert `leftrigh` to Base R numeric format, either as a new variable or as shown below, temporarily:

```
describe(as.numeric(bsa$leftrigh))
```

```
as.numeric(bsa$leftrigh) 
       n  missing distinct     Info     Mean  pMedian      Gmd      .05 
    3206      782       30    0.993     2.52      2.5   0.8831      1.2 
     .10      .25      .50      .75      .90      .95 
     1.4      2.0      2.4      3.0      3.6      4.0 

lowest : 1    1.2  1.4  1.5  1.6 , highest: 4.4  4.6  4.75 4.8  5
```

`describe()` also provides the number of observations (including missing and unique observations), deciles as well as the five largest and smallest values.

Commands producing single statistics are also available:

```
mean(bsa$leftrigh, na.rm = T)
```

```
[1] 2.519911
```

```
sd(bsa$leftrigh, na.rm = T)
```

```
[1] 0.7852958
```

```
median(bsa$leftrigh, na.rm = T)
```

```
[1] 2.4
```

```
max(as.numeric(bsa$leftrigh), na.rm = T)
```

```
[1] 5
```

```
min(as.numeric(bsa$leftrigh), na.rm = T)
```

```
[1] 1
```

As previously, the `na.rm = T` option prevents missing values from being taken into account (in which case the output would have been NA, as this is the default behaviour of these functions). Similarly to `describe()` earlier, `max()` and `min()` need the variable to be converted into numeric format to deliver the desired output.

We could combine the output from the above commands into a single line using the `c()` function:

```
c(
  mean(bsa$leftrigh, na.rm = T),
  sd(bsa$leftrigh, na.rm = T),
  median(bsa$leftrigh, na.rm = T),
  max(as.numeric(bsa$leftrigh), na.rm = T),
  min(as.numeric(bsa$leftrigh), na.rm = T)
)
```

```
[1] 2.5199106 0.7852958 2.4000000 5.0000000 1.0000000
```

Using these individual commands may come in handy, for instance when further processing of the result is needed:

```
m <- mean(bsa$leftrigh, na.rm= T)
```

Let's round the results to two decimal places:

```
rm <- round(m,2)
```

We can see the final results by typing:

```
rm
```

```
[1] 2.52
```

Note:

```
round(mean(bsa$leftrigh,na.rm=T),2)
```

```
[1] 2.52
```

would produce the same results using just one line of code .

5.2 Bivariate association between continuous variables

The Base R installation comes with a wide range of bivariate statistical functions. `cor()` and `cov()` provide basic measures of association between two variables. For instance, in order to measure the correlation between the left-right the libertarian-authoritarian scales:

```
cor(bsa$leftrigh, bsa$libauth, use='complete.obs')
```

```
[1] 0.009625928
```

The latter variable is records how far someone sits on the libertarian – authoritarian scale ranging from 1 to 5.

A correlation of 0.009 indicates a positive but very small relationship. It can be interpreted as 'an increase in authoritarianism is associated with a marginal increase in rightwing views.

Note: When using `cor()` and `cov()`, missing values are dealt with the `use=` option, which can either take "everything", "all.obs", "complete.obs", "na.or.complete", or "pairwise.complete.obs" values. See `?cor` for additional information.

5.3 Categorical Variables

As with continuous variables, R offers several tools that can be used to describe the distribution of categorical variables. One- and two-way contingency tables are the most commonly used.

5.3.1 One way frequency tables

There are several R commands that we can use to create frequency tables. The most common ones `table()`, `xtabs()` or `ftable()` which return the frequencies of observations within each level of a factor. For example, in order to obtain the political affiliation of BSA respondents in 2017:

```
table(as_factor(bsa$PartyId2))
```

```
     Not applicable          Conservative                Labour     Liberal Democrat
                  0                  1263                  1479                  241
        Other party                  None           Green Party Other answer/DK/Ref
                193                   515                    79                    0
```

As with any other R functions, the outcome of `table()` can be stored as an object for further processing:

```
a<-table(as_factor(bsa$PartyId2))
```

It is not directly possible to have proportions or percentages computed with `table()`. Proportions are obtained using the `prop.table()` function which in turn does not produce percentages. It is also a good idea to round the results for greater readability.

Either:

```
round(
  100*
    prop.table(a),
  1)
```

```
     Not applicable          Conservative                Labour     Liberal Democrat
                0.0                  33.5                  39.2                  6.4
        Other party                  None           Green Party Other answer/DK/Ref
                5.1                  13.7                   2.1                  0.0
```

… or:

```
round(100*
        prop.table(
          table(as_factor(bsa$PartyId2))
        ),
      1)
```

```
     Not applicable          Conservative                Labour     Liberal Democrat
                0.0                  33.5                  39.2                  6.4
        Other party                  None           Green Party Other answer/DK/Ref
                5.1                  13.7                   2.1                  0.0
```

5.3.2 Two way or more contingency table

The simplest way to produce a two-way contingency table is to pass a second variable to `table()`:

```
table(as_factor(bsa$PartyId2), as_factor(bsa$Rsex))
```

```
                     skipped or na Male Female Dontknow Refusal
  Not applicable                 0    0      0        0       0
  Conservative                   0  627    636        0       0
  Labour                         0  644    835        0       0
  Liberal Democrat               0  124    117        0       0
  Other party                    0   97     96        0       0
  None                           0  199    316        0       0
  Green Party                    0   31     48        0       0
  Other answer/DK/Ref            0    0      0        0       0
```

By default `table` does not discard empty factor levels - (i.e. categories with no observations), which may sometimes result in slightly cumbersome result. Using `droplevels()` on each variable resolves the issue:

```
table(droplevels(as_factor(bsa$PartyId2)), droplevels(as_factor(bsa$Rsex)))
```

```
                  Male Female
  Conservative     627    636
  Labour           644    835
  Liberal Democrat 124    117
  Other party       97     96
  None             199    316
  Green Party       31     48
```

However, when dealing with more than one variable it is recommended to use `xtabs()` instead as it has a number of desirable functions directly available as options. The syntax is slightly different as it relies on a `formula` – a R object consisting of elements separated by a tilde '~'. The variables to be tabulated are specified on the right hand side of the formula. In order to lighten the syntax, we will also recode `PartyId2` and `Rsex` permanently into factors.

```
bsa$PartyId2.f<-as_factor(bsa$PartyId2)
bsa$Rsex.f<-as_factor(bsa$Rsex)

xtabs(~PartyId2.f +Rsex.f,
      data = bsa)
```

```
                     Rsex.f
PartyId2.f            skipped or na Male Female Dontknow Refusal
  Not applicable                 0    0      0        0       0
  Conservative                  0  627    636        0        0
  Labour                        0  644    835        0        0
  Liberal Democrat              0  124    117        0        0
  Other party                   0   97     96        0        0
  None                          0  199    316        0        0
  Green Party                   0   31     48        0        0
  Other answer/DK/Ref           0    0      0        0        0
```

The `data=` parameter does not have to be explicitly specified as simply using ´`bsa`' will work. Other useful options are:

- `subset=`, which allows direct specification of a subpopulation from which to derive the table;
- `drop.unused.levels=T` to remove empty levels;
- `weights~` variables on the right hand side of the formula will be treated as weights, a useful feature for survey analysis.

As previously `prop.table()` is necessary in order to obtain proportions:

```
b<-xtabs(~PartyId2.f +Rsex.f,
         bsa,
         drop.unused.levels = T)

round(100*
        prop.table(b),
      1)  ### Cell percentages
```

```
                  Rsex.f
PartyId2.f         Male Female
  Conservative     16.6   16.9
  Labour           17.1   22.1
  Liberal Democrat  3.3    3.1
  Other party       2.6    2.5
  None              5.3    8.4
  Green Party       0.8    1.3
```

The largest group in the sample (22.1%) is made of labour-voting females and the smallest, of green-voting males.

```
round(100*
        prop.table(b,1),
      1)  ### Option 1 for row percentages
```

```
                  Rsex.f
PartyId2.f         Male Female
  Conservative     49.6   50.4
  Labour           43.5   56.5
  Liberal Democrat 51.5   48.5
  Other party      50.3   49.7
  None             38.6   61.4
  Green Party      39.2   60.8
```

Conservative voters are more or less evenly split between men and women, whereas Labour and Green voters are more likely to be women.

```
round(100*
        prop.table(b,2),
      1) ### Option 2 for column percentages
```

```
                  Rsex.f
PartyId2.f         Male Female
  Conservative     36.4   31.1
  Labour           37.4   40.8
  Liberal Democrat  7.2    5.7
  Other party       5.6    4.7
  None             11.6   15.4
  Green Party       1.8    2.3
```

Similar proportions of men voted Conservative and Labour (36-37%), whereas women were clearly more likely to vote Labour.

There is not a straightforward way to obtain percentages in three-way contingency tables with either `xtabs()` or `table()`. This is where `ftable()` function comes handy. For convenience, we converted `RAgeCat` into a factor.

```
bsa$RAgeCat.f<-as_factor(bsa$RAgeCat)

round(100*
        prop.table(
          ftable(RAgeCat.f~PartyId2.f+Rsex.f,
                 data=droplevels(bsa)
                   )
         ,1)
      ,1) ### Option 1 for row,  2 for column percentages
```

```
                        RAgeCat.f 18-24 25-34 35-44 45-54 55-59 60-64  65+
PartyId2.f       Rsex.f
Conservative     Male               3.0   7.8  13.9  15.0   8.6   9.3 42.4
                 Female             2.7   7.1   8.8  18.6   8.8   8.7 45.4
Labour           Male               7.6  16.1  14.3  21.3   7.5   9.3 23.9
                 Female             7.9  20.2  19.2  18.8   7.1   7.1 19.7
Liberal Democrat Male               0.8  13.7  19.4  15.3   9.7   8.9 32.3
                 Female             4.3   9.4  26.5   6.0   6.0   9.4 38.5
Other party      Male               3.1  14.4  11.3  17.5  11.3  16.5 25.8
                 Female             5.2  14.6  15.6  16.7  11.5   8.3 28.1
None             Male               7.5  22.1  20.6  17.1  11.1   6.0 15.6
                 Female             8.5  22.5  20.6  21.8   7.3   6.3 13.0
Green Party      Male               6.5  32.3  16.1  29.0   6.5   3.2  6.5
                 Female             6.2  18.8  25.0  20.8   6.2  10.4 12.5
```

The table gives the relative age breakdown for each gender/political affiliation combination (ie row percentages). Here again we used `droplevels()`: this removes unused factor levels which would otherwise be displayed and make the table difficult to read. `droplevels()` can be applied either to entire data frames or single variables.

5.4 Grouped summary statistics for continuous variables

A common requirement in survey analysis is the ability to compare descriptive statistics across subgroups of the data. There are different ways to do this in R. We demonstrate below the most straightforward one, which is obtained by using some of the functions available in the `dplyr` package.

```
bsa%>%
  group_by(PartyId2.f)%>%
  summarise(mdscore=median(libauth,na.rm=T),
            sdscore=sd(libauth,na.rm=T))
```

```
# A tibble: 7 x 3
  PartyId2.f       mdscore sdscore
  <fct>              <dbl>   <dbl>
1 Conservative        3.67   0.587
2 Labour              3.33   0.774
3 Liberal Democrat    3.17   0.726
4 Other party         3.67   0.739
5 None                3.67   0.584
6 Green Party         2.83   0.872
7 <NA>                3.67   0.564
```

The above command produces a table of summary values (median and standard deviations) of the Liberal vs authoritarian scale. We can see from the first one that Green party voters are the most liberal, followed by Labour, whereas non voters and Conservatives are the most authoritarian. Liberal Democrats are the most cohesive group (i.e. with the smallest standard deviation). We chose to leave non-responses for `PartyId2` for this analysis. Some users might want to remove them instead before computing their results as in the table below. We do this by using `is.na()`, which checks variables for the presence of system missing values, in conjunction with `filter()`.

```
bsa%>%
  filter(!is.na(PartyId2.f)) %>%
  group_by(Rsex.f,PartyId2.f) %>%
  summarise(mnscore=sd(libauth,na.rm=T),
            mdscore=median(libauth,na.rm=T))
```

```
# A tibble: 12 x 4
# Groups:   Rsex.f [2]
   Rsex.f PartyId2.f       mnscore mdscore
   <fct>  <fct>              <dbl>   <dbl>
 1 Male   Conservative       0.607    3.67
 2 Male   Labour             0.765    3.33
 3 Male   Liberal Democrat  0.766    3.17
 4 Male   Other party       0.703    3.83
 5 Male   None              0.616    3.67
 6 Male   Green Party       1.04     2.67
 7 Female Conservative      0.565    3.67
 8 Female Labour            0.781    3.33
 9 Female Liberal Democrat  0.688    3.17
10 Female Other party       0.773    3.67
11 Female None              0.565    3.67
12 Female Green Party       0.744    3
```

When further broken down by gender, we can see that overall the same trends remain valid, with some nuances: male Green supporters are markedly more liberal than their female counterpart, the opposite being true among Conservative supporters.

Instead of tables of summary statistics, we may want to have summary statistics computed as variables that will be added to the current dataset for each corresponding gender/political affiliation group. This is straightforward to do with dplyr, we just need to use the `mutate()` command.

```
bsa<-bsa%>%
  group_by(Rsex.f,PartyId2.f)%>%
  mutate(msscore=sd(libauth,na.rm=T),
         mdscore=median(libauth,na.rm=T))
```

However, we also need to add the newly created variables into the existing bsa dataset, which the first line of the syntax above does. We can check that the variables have been created and that the correct values have been assigned to each sex/affiliation category.

```
names(bsa)
```

```
 [1] "Sserial"          "Rsex"             "RAgeCat"          "Married"
 [5] "ChildHh"          "HEdQual3"         "eq_inc_quintiles" "RClassGp"
 [9] "CCBELIEV"         "carallow"         "carreduc"         "carnod2"
[13] "cartaxhi"         "carenvdc"         "plnenvt"          "plnuppri"
[17] "Politics"         "Voted"            "actchar"          "actpol"
[21] "govnosa2"         "PartyId2"         "leftrigh"         "libauth"
[25] "WtFactor"         "PartyId2.f"       "Rsex.f"           "RAgeCat.f"
[29] "msscore"          "mdscore"
```

```
bsa[4:8,c("Rsex","PartyId2","mdscore")]
```

```
# A tibble: 5 x 3
  Rsex        PartyId2                mdscore
  <dbl+lbl>   <dbl+lbl>                 <dbl>
1 2 [Female]   2 [Labour]                3.33
2 1 [Male]     3 [Liberal Democrat]      3.17
3 2 [Female] NA                          3.67
4 1 [Male]     3 [Liberal Democrat]      3.17
5 2 [Female]   6 [Green Party]           3
```

6 Producing weighted and survey design-informed estimates

Most users of social surveys will be looking to produce representative estimates when conducting analyses. Conducting such population inference entail using weights, which are meant to correct estimates for the under/over representation of certain groups in the sample due to the sampling process and non-response. Producing sound results relies not just on weighting estimates but also on computing adequate standard errors or confidence intervals, which measure the precision of the estimates.

The recommended approach to inferring confidence intervals and standard errors involves accounting for the survey design (ie the way sampling was carried out) when conducting analyses – which can be done with the `survey` package in R, a topic described in Section 6.3. At the same time, for users who are concerned with quickly computing reasonably accurate point estimates, rather than publication-quality results, it may be useful to be aware which common R commands and operations can be used with weights.

6.1 Central tendency and dispersion (continuous variables)

The `stats` package, part of Base R, includes `weighted.mean()` which, as indicated by its name, computes weighted estimates of the mean of a variable when weights are provided. However, the `Hmisc` package includes a more comprehensive set of functions that can be used when weighting estimates: `wtd.mean()`, `wtd.var()` and `wtd.quantile()`. The code below provides an illustration of weighted means, variance and median of the left-right political attitudes score used in previous chapters, each time comparing it with the unweighted estimates:

```
### Mean
c(mean(bsa$leftrigh,na.rm=T),
  wtd.mean(bsa$leftrigh,bsa$WtFactor)
  )
```

```
[1] 2.519911 2.521589
```

```
### Variance
c(var(bsa$leftrigh,na.rm=T),
  wtd.var(bsa$leftrigh,bsa$WtFactor))
```

```
[1] 0.6166894 0.6195378
```

```
### Median and quartiles
c(quantile(bsa$leftrigh,na.rm=T,probs=c(.25,.5,.75)),
  wtd.quantile(bsa$leftrigh,bsa$WtFactor,probs=c(.25,.5,.75)))
```

```
25% 50% 75% 25% 50% 75%
2.0 2.4 3.0 2.0 2.4 3.0
```

The above functions can be used in conjunction with `group_by()` and `summarise()` in order to compute weighted estimates of continuous variables by groups of categorical variables:

```
bsa%>%
  filter(!is.na(RAgeCat.f))%>%
  group_by(RAgeCat.f)%>%
  summarise(Mean=wtd.mean(leftrigh,WtFactor),
            Var=wtd.var(leftrigh,WtFactor),
            Median=wtd.quantile(leftrigh,WtFactor,probs=c(.5))
            )
```

```
# A tibble: 7 x 4
  RAgeCat.f  Mean   Var Median
  <fct>     <dbl> <dbl>  <dbl>
1 18-24      2.49 0.556    2.4
2 25-34      2.56 0.577    2.6
3 35-44      2.52 0.615    2.4
4 45-54      2.53 0.671    2.6
5 55-59      2.54 0.653    2.4
6 60-64      2.46 0.685    2.4
7 65+        2.52 0.613    2.4
```

6.2 Frequencies and contingency tables

Neither `ftable()` nor `table()` that were used in previous chapter allow weights. Although the `Hmisc` packages includes a `wtd.table()` function for one-way frequency tables, we recommend using `xtabs()` as previously, as it is more versatile and allows weights. Indeed, as variables used with `xtabs()` are specified on the right hand side of a formula:

```
> xtabs(~var1, data=mydata)
```

or

```
> xtabs(~var1 + var2, data=mydata)
```

… A variable containing weights can be passed to `xtabs()` by specifying its name on the left hand-side of the equation (or the tilde `~`)

```
> xtabs(weights~var1 + var2, data=mydata)
```

Let's apply this technique to investigate respondents' agreement with the sentence: *People should be able to travel by plane as much as they like, even if this harms the environment* as recorded in the `plnenvt` variable.

```
bsa$plnenvt.f<-as_factor(bsa$plnenvt) # Converts the original variable into a factor

## Unweighted vs weighted frequency tables
cbind(
  Unweighted=round(
    100*prop.table(
      xtabs(~plnenvt.f,bsa,
            drop.unused.levels = T)
        ),
    1),
  Weighted=round(
    100*prop.table(
      xtabs(WtFactor~plnenvt.f,bsa,
            drop.unused.levels = T)
      ),
    1)
)
```

```
                           Unweighted Weighted
agree strongly                    4.6      4.8
agree                            16.0     15.9
neither agree nor disagree       33.2     33.2
disagree                         36.3     37.0
disagree strongly                10.0      9.0
```

Obtaining a weighted contingency table of respondents' views about flying by gender follow the same logic:

```
## Unweighted vs weighted contingency tables
cbind(
  round(
    100*prop.table(
      xtabs(~plnenvt.f+Rsex.f,bsa,
            drop.unused.levels = T),
      1),
    1),
  round(
    100*prop.table(
      xtabs(WtFactor~plnenvt.f+Rsex.f,bsa,
            drop.unused.levels = T),
      1),
    1)
)
```

```
                           Male  Female Male  Female
agree strongly             50.0    50.0 46.1    53.9
agree                      47.2    52.8 53.5    46.5
neither agree nor disagree 40.8    59.2 43.6    56.4
disagree                   47.9    52.1 52.7    47.3
disagree strongly          42.3    57.7 41.5    58.5
```

6.3 Inference using survey procedures

The weighting procedures described above could be described as 'quick and dirty' in that they compute representative point estimates – i.e. single values. Computing the precision of survey data estimates – for example via their standard error – requires more than just adding weights to a command. Information about the survey design, its primary sampling units, strata and clusters is required so that robust standard errors, statistical tests and/or confidence interval can be computed. The `Survey` package was designed in order to deal with this set of issues. It provides functions for computing common estimates while accounting for the survey design. Its most important features are described below.

In order to use survey functions one first needs to create a `svydesign` object, in essence a version of the data that incorporates the sample design information available, then compute the required estimate using the `svydesign` object.

A common issue with survey datasets available in the UK is that sampling information is often only available in a secured version of the data, restricting its access to authorised users in a secure lab. Although it is sometimes possible to use available variables to account for aspects of the sample design – region as a strata in the case of stratified samples – in most cases users are left with computing standard errors without sample design information, which amounts to assuming that the sample was drawn purely at random. Even if this is the case however, using the `survey` package is recommended, as it provides a coherent framework for computing survey parameters.

```
library(survey) ### Loading the package in memory
bsa.design<-svydesign(ids =~1,
                      weights=~WtFactor,
                      data=bsa)
```

The code above simply declares the survey design by creating the `bsa.design` object (the name is arbitrary). The `ids=` parameter is where primary sampling units are declared, as well as any clustering information as a formula ie `~PSU+cluster2id...`. When PSU information is unavailable `ids` is given the value 1 or 0. A `strata=` and `fpc=` are available in order to declare the sampling strata and the variable used for finite population correction. None of these are available in the bsa dataset, and estimation commands will therefore rely on the assumption of simple random sampling.

We can now compute estimates comparable to those in the previous sections. The code below provides the mean of the left vs right political orientation indicator, as well as its 95% confidence interval:

```
a<-svymean(~leftrigh,
        bsa.design,
        na.rm = T)### Computes the mean and its standard error...

a
```

```
           mean     SE
leftrigh 2.5216 0.0155
```

```
confint(a) ### ... and confidence interval
```

```
            2.5 %   97.5 %
leftrigh 2.491277 2.551902
```

We can similarly

```
oldsvyquantile(~leftrigh,
            bsa.design,
            quantiles=.5,
            na.rm = T)
```

```
         0.5
leftrigh 2.4
```

Frequency and contingency tables are computed using `svytable()`, whose syntax relies on formulas similarly to `xtabs()` in the previous chapter.

```
### A frequency table...
round(100*
        prop.table(
          svytable(~RAgeCat.f,bsa.design)
        ),
      1)
```

```
RAgeCat.f
18-24 25-34 35-44 45-54 55-59 60-64   65+
 11.2  17.2  16.1  17.9   7.9   6.8  22.8
```

```
### And a two-way contingency table:

round(100*
        prop.table(
          svytable(~RAgeCat.f+Rsex.f,bsa.design),
          1),
      1)
```

```
         Rsex.f
RAgeCat.f Male Female
    18-24 51.1   48.9
    25-34 50.2   49.8
    35-44 49.7   50.3
    45-54 49.3   50.7
    55-59 48.8   51.2
    60-64 49.0   51.0
    65+   45.4   54.6
```

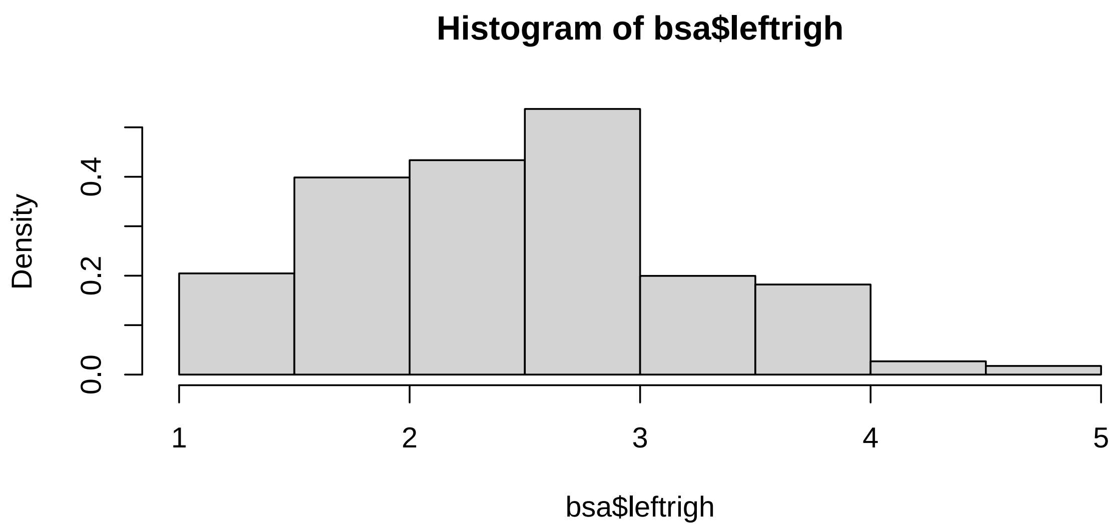
7 Graphs and plots

There are two common ways to visualise data in R: either by using the straightforward but rather basic plotting commands from the Base package, or instead by delving into the more complex but much nicer looking functionalities of the `ggplot` package.

7.1 Distributional graphs for continuous variables

Plots such as histograms or box plots are a convenient way to gain a quick overview of the distribution of a variable and are easy to produce. Going back to the BSA data, we can plot the distribution of left-right political orientations scores with the `hist()` command.

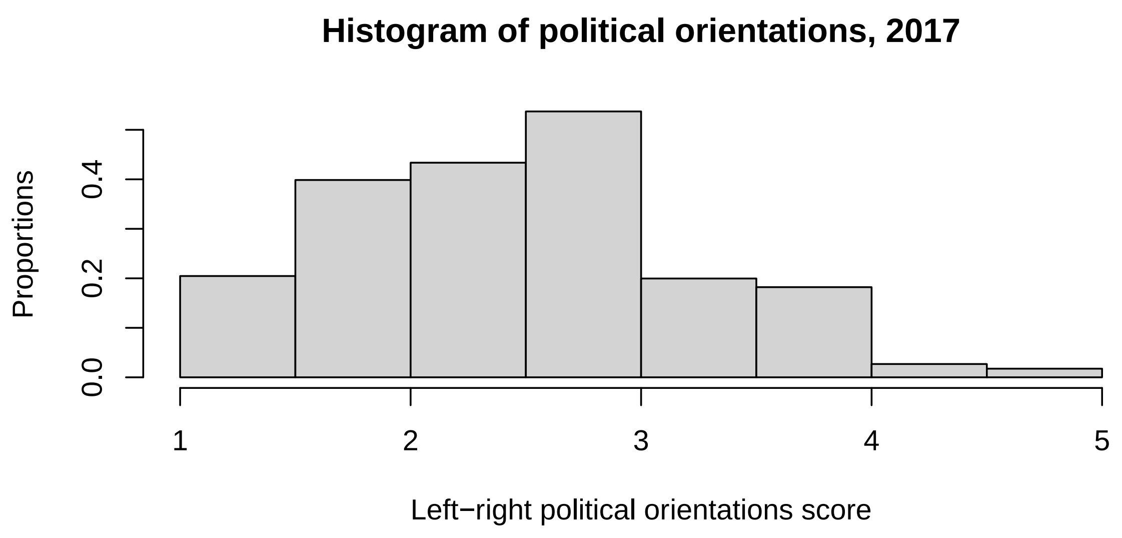
```
hist(bsa$leftrigh,freq=FALSE)
```



The histogram should appear in the 'Plot' tab on the right hand side of the R Studio window. It shows us that political orientations are slightly skewed towards the left. The `freq=FALSE` option requires the y-axis to be expressed in terms of proportions rather than number of observations.

Titles and labels can easily be added:

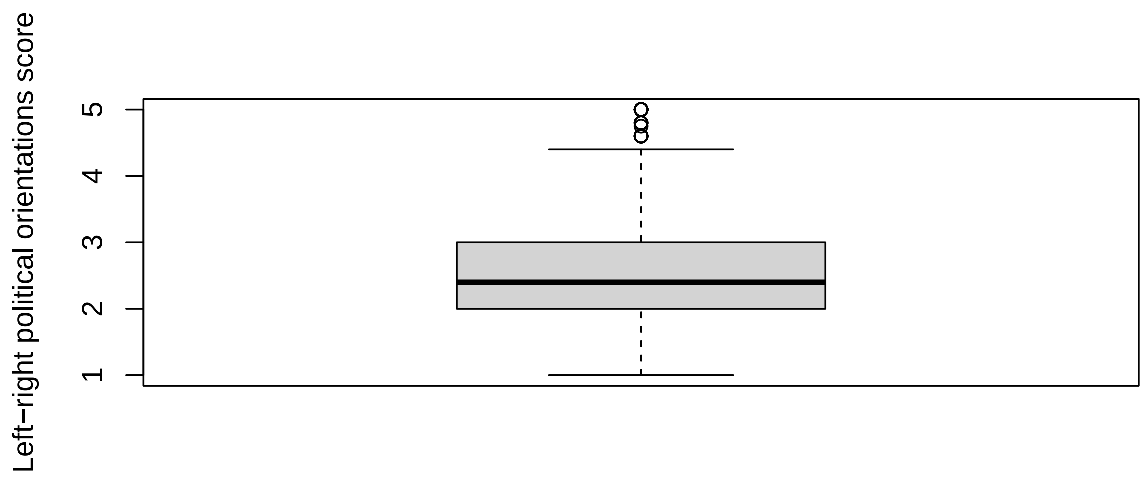
```
hist(bsa$leftrigh,
     freq=FALSE,
     main="Histogram of political orientations, 2017",
     ylab="Proportions",
     xlab="Left-right political orientations score")
```



Note that `main`, `ylab` and `xlab` can be used with any Base R plot commands.

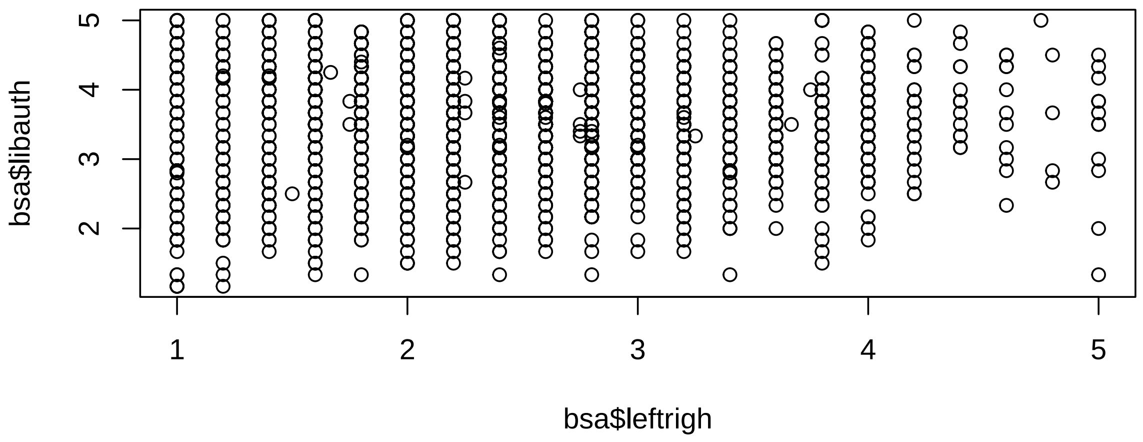
We can also produce a box and whisker plot of the same variable in order to get a better sense of the distribution of outliers:

```
boxplot(bsa$leftrigh,
        # main="Box and whisker plot of political orientations",
        ylab="Left-right political orientations score"
)
```



The generic `plot()` command produces scatterplots. Let's try it with our left-right political orientations score, in conjunction with `libauth`, a libertarian vs authoritarian scale.

```
plot(bsa$leftrigh,bsa$libauth) ### Scatterplot of left-right political orientations score
```

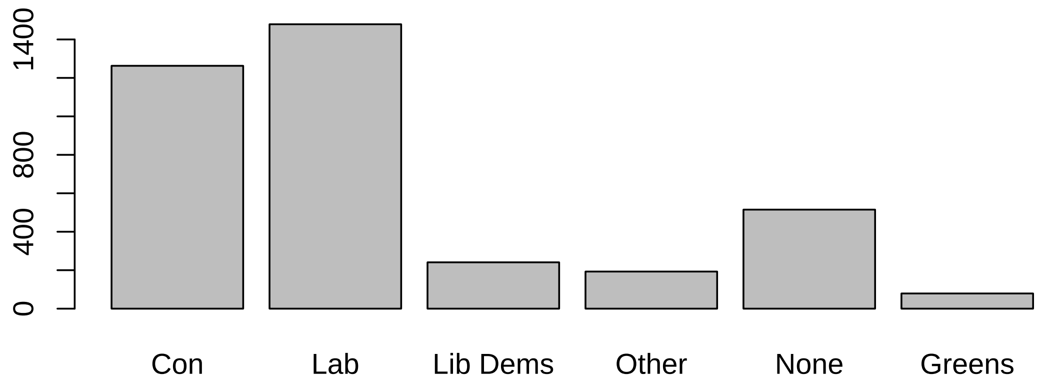


The scatterplot shows us that there is little association between the two variables. However, slightly fewer respondents simultaneously score high on the ‘authoritarianism’ and ‘right’ scales, perhaps unsurprisingly.

7.2 Plotting categorical variables

The generic `plot()` function provides a quick way to produce bar plots of categorical data. For example, we can examine the distribution of political party affiliations (`Politics` variable). In order to do this, we convert it into a factor (i.e. categorical) variable as previously. Some preliminary abbreviation of the factor levels are also required in order for them to be displayed properly.

```
bsa$PartyId2.f<-droplevels( # Getting rid of unused factor levels for neater output
  as_factor(bsa$PartyId2)    # Converting haven labelled variable to factor
)
levels(bsa$PartyId2.f)<-c(
  "Con","Lab","Lib Dems","Other","None", "Greens" # Shorter level names
  )

plot(bsa$PartyId2.f)
```

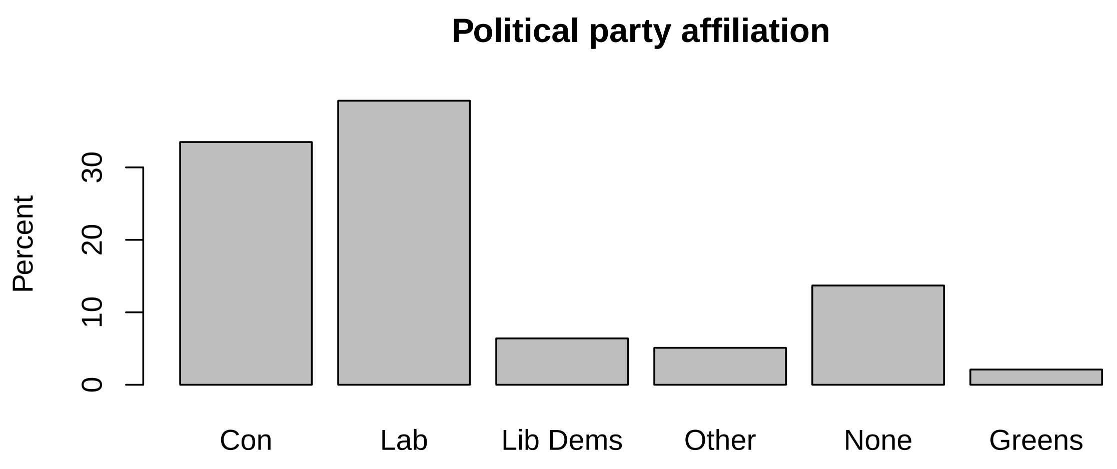


More advanced plots require the barplot() function, which can be used in conjunction with table(). Whereas table() creates the data that will be plotted, barplot() does the actual plotting. For instance, we can produce the same bar plot, but this time with percentages, by creating a frequency table as we did above in Section 5.2, then plot it.

```
party.tab<-round(100*prop.table(
  table(bsa$PartyId2.f)
),
1)

party.tab
```

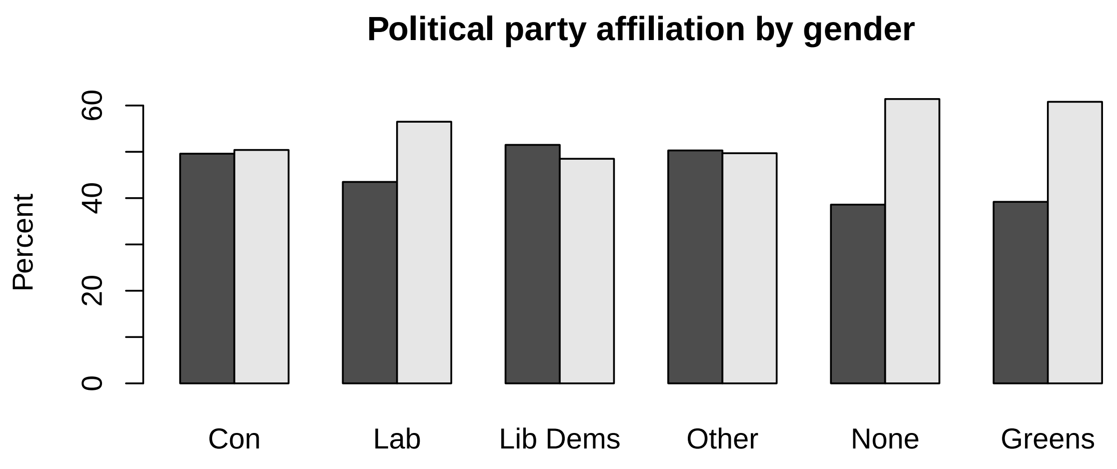
```
     Con      Lab Lib Dems    Other     None   Greens 
    33.5     39.2      6.4      5.1     13.7      2.1 
```

```
barplot(party.tab,
        main="Political party affiliation",
        ylab="Percent")
```



We can go further and create plots for two-way contingency tables of party affiliation by gender. This time we will do it in a single command:

```
#t<-xtabs(~PartyId2.f+Rsex.f,bsa)       # First, let's get the contingency table
t<-xtabs(~Rsex.f+PartyId2.f,bsa)       # First, let's get the contingency table
barplot(
  round(100*
          prop.table(t,2),              ## Column % (here, gender)
    1),                ## Rounded to 1 decimal
  beside = T,      ## Side-by-side bars
  main="Political party affiliation by gender",
  ylab="Percent")
```



7.3 More advanced plots

Social science research often requires more advanced plots than just bar charts in order to conduct insightful analyses, such as for instance comparing the mean or median value of a continuous outcome across two or more categorical variables. The `ggplot` package provides one of the most advanced set of tools for data visualisation currently available. A few examples are provided below.

Three way barplots using ggplot2

Say we would like to explore how differences in political party affiliations vary by gender and whether respondents have a degree-level education.

Let us first prepare the data: we need to create the table of result, the proportion of degree vs non degree holders by gender and political party. This is a three-way contingency table, that we can obtain with `ftable()` as shown in Section 5.2, combined with `prop.table()` for the computation of proportions and `round()`.

As they are more straightforward to handle in `ggplot`, we convert the table object created by `ftable` into a data frame. Although it is possible to specify titles and axis labels in the plotting command, we will keep things simple and have them already in the data.

Rather than using the full range of educational achievements recorded in `HEdQual3`, we would like instead to have a dichotomic variable measuring whether respondents are degree holders or not. Adding it directly in the `ftable` command as a boolean expression return a dichotomic variable: “TRUE” for Degree educated respondents, and “FALSE” for everyone else. We just need to change the levels of this factor variable to make them more intelligible. Finally, we change the variable names in our data frame.

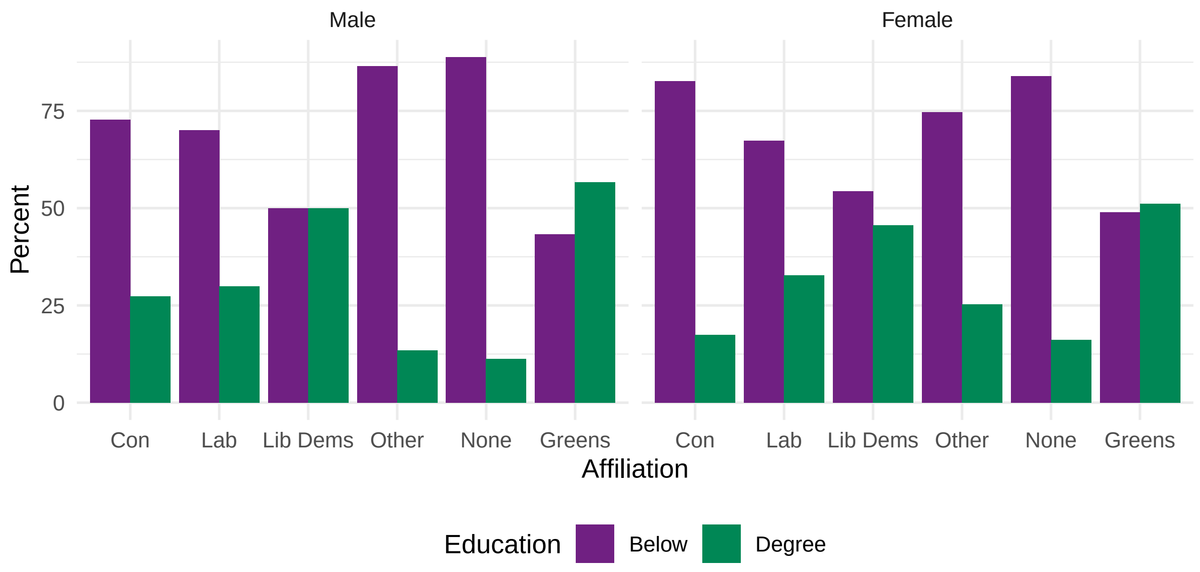
```
bsa$HEdQual3.f<-droplevels(as_factor(bsa$HEdQual3))
pa<-round(100*
            prop.table(
              ftable(bsa$PartyId2.f,bsa$Rsex.f,(bsa$HEdQual3.f=="Degree")
                     ),
              1),
          1)

pa<-data.frame(pa)
levels(pa$Var3)<-c("Below","Degree")
names(pa)<-c("Affiliation","Gender","Education","Percent")
pa
```

```
   Affiliation Gender Education Percent
1          Con   Male     Below    72.7
2          Lab   Male     Below    70.1
3     Lib Dems   Male     Below    50.0
4        Other   Male     Below    86.5
5         None   Male     Below    88.8
6       Greens   Male     Below    43.3
7          Con Female     Below    82.6
```

```
8          Lab Female     Below    67.3
9     Lib Dems Female     Below    54.4
10       Other Female     Below    74.7
11        None Female     Below    83.9
12      Greens Female     Below    48.9
13         Con   Male    Degree    27.3
14         Lab   Male    Degree    29.9
15    Lib Dems   Male    Degree    50.0
16       Other   Male    Degree    13.5
17        None   Male    Degree    11.2
18      Greens   Male    Degree    56.7
19         Con Female    Degree    17.4
20         Lab Female    Degree    32.7
21    Lib Dems Female    Degree    45.6
22       Other Female    Degree    25.3
23        None Female    Degree    16.1
24      Greens Female    Degree    51.1
```

We are now ready to plot the data. `ggplot()` functions usually work as a succession of layers or options that are added to an initial plot specification. Each extra layer is added after a `+` sign. In the example below, we specify the data and the *aesthetic* (i.e. the basic parameters of the plot) with the first command: the x and y variables as well as the first grouping variable, in our case education). `geom_bar()` stipulates the bar plot, with the `position="dodge"` option for the bars to be located side by side (position=“stack”would have them on top of each other). Finally, `facet_wrap()` splits the plot by gender.

```
ggplot(data=pa,aes(y=Percent,x=Affiliation,fill=Education))+
  geom_bar(position="dodge",stat="identity")+
  facet_wrap(~Gender)+
  theme_minimal()+                  ### Theme for visualisation
  scale_fill_manual(values=c("#702082", "#008755"))+ ### Custom colours (optional)
  theme(legend.position = "bottom")
```



Mosaic plots

Mosaic plots provide a visualisation tool for two-way or more contingency tables. Table cells are plotted as rectangles whose surface is proportional to their overall number of observations whereas their length represents their frequency relative to that of the second variable for each category of the first one - this is equivalent to column percentages in a contingency table. In the example below, we are using the `mosaic()` function from the `vcd` package which provides both descriptive and model-based plots.

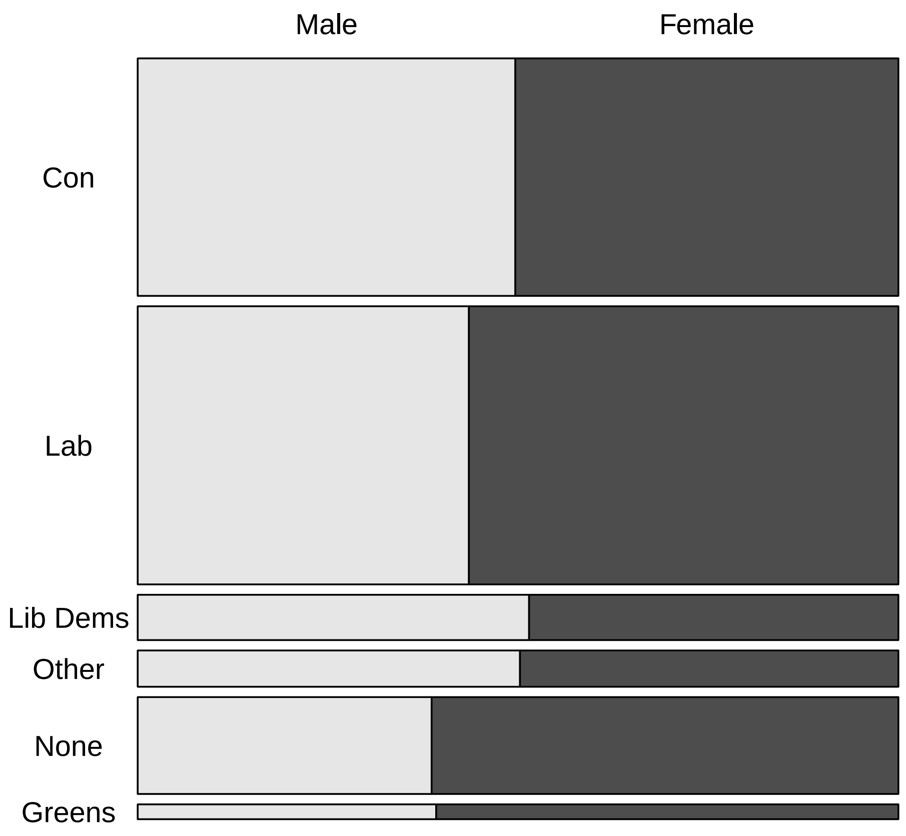
The basic parameters of a descriptive mosaic plot consist of:

- A R formula where the row and columns variables are specified;
- The highlighting variable, for the display of relative percentage;
- By default, the contrasting colours are shades of grey but can be customised, as can the direction of the bars.

Labelling options are a bit mode arcane than Base R plot functions. They need to be specified within a `labeling= labeling_border()` statement. See `help(labeling_border)` for more detail. We used `Varnames=F` to hide the variable names from the plot, `rot_labels`, to specify that we did not want any rotation of the value labels, and `offset_labels` to prevent them to overlap with the rectangles.

```
library(vcd)                 # Loading the vcd package

mosaic(~PartyId2.f+Rsex.f,
       data=bsa,
       highlighting = "Rsex.f",
       labeling= labeling_border(varnames = F,
                                 rot_labels = c(0,0,0,0),
                                 offset_labels = c(0, 0, 0, 1)
                                ), # labelling functions from mosaic. See
                                   # help(labeling_border) for more detail
                    main="Mosaic plot of political party affiliation by gender"
              )
```

Mosaic plot of political party affiliation by gender



8 Significance testing

This section describes how to implement common statistical tests in R both without and with weights and survey design information. A working knowledge of these tests and their theoretical assumptions is assumed.

8.1 Differences between means

Two common ways of conducting significance testing consist in testing whether sample means significantly differ from 0 (one sample t-test), or between two groups (two samples t test). In the latter case, one can further distinguish between independent samples (where means come from different groups), or paired samples (when the same measure is taken at several points in time). Given that it is probably one of the most widely used statistical tests in social sciences, we will only cover the former here. Several R packages provide functions for conducting t tests. We will be using `t.test()`, provided by the `stats` package (R Base).

Suppose we would like to test whether the libertarianism vs authoritarianism score `libauth` significantly differs between men and women using a t test. A two sided test is the default, with H_0 or the null hypothesis being that there are no differences between groups, and H_1 or the alternative hypothesis that the group means do indeed differ. The test is specified with a formula with on the left-hand side the quantity to be tested and on the right-hand side the grouping variable.

One sided tests can be conducted by specifying that the alternative hypothesis (H_1) is that quantities are either **greater** or **smaller**. `t.test()` assumes that by default the variances are unequal. This can be changed with the `var.equal=T` option.

```
# Testing for significant differences in liberal vs authoritarian score
summary(
t.test(libauth~Rsex.f,
       data=bsa)
)
```

```
            Length Class  Mode
statistic   1      -none- numeric
parameter   1      -none- numeric
p.value     1      -none- numeric
conf.int    2      -none- numeric
estimate    2      -none- numeric
null.value  1      -none- numeric
stderr      1      -none- numeric
alternative 1      -none- character
method      1      -none- character
data.name   1      -none- character
```

No significant differences (ie the difference in `libauth` between men and women is not significantly different from zero)

```
# Testing for whether men have a lower (ie more left-wing)  score
t.test(leftrigh~Rsex.f,
       data=bsa,
       alternative="less")
```

```

    Welch Two Sample t-test

data:  leftrigh by Rsex.f
t = -2.0687, df = 2858, p-value = 0.01933
alternative hypothesis: true difference in means between group Male and group Female is less than 0
95 percent confidence interval:
        -Inf -0.01197607
sample estimates:
  mean in group Male mean in group Female
            2.487564             2.546087
```

Men have a significantly lower score on the scale (at the .05 threshold) and are therefore on average leaning more to the left than women.

The result of the above tests may be biased as they do not account for bias from either sample design or non-response. When results representative of the British population are required, a survey designed informed version of the t test should be used. The `survey` package that we used earlier in Chapter 6 provides such a function.

```
library(survey)
bsa.design<-svydesign(ids =~1,           # Declaring the survey design
                      weights=~WtFactor,
                      data=bsa)

svyttest(libauth~Rsex.f,bsa.design)   # SD informed t-test of libauth by gender
```

```

    Design-based t-test

data:  libauth ~ Rsex.f
t = 0.10069, df = 3211, p-value = 0.9198
alternative hypothesis: true difference in mean is not equal to 0
95 percent confidence interval:
 -0.05411275  0.05997165
sample estimates:
difference in mean
       0.002929454
```

```
svyttest(leftrigh~Rsex.f,bsa.design) # SD informed t-test of leftrigh by gender
```

```
    Design-based t-test

data:  leftrigh ~ Rsex.f
t = 2.308, df = 3204, p-value = 0.02106
alternative hypothesis: true difference in mean is not equal to 0
95 percent confidence interval:
 0.01085242 0.13337485
sample estimates:
difference in mean
        0.07211364
```

In this case the output of `svyttest()` did not lead to a different conclusion than the one we drew above. However, we can notice that the significance of differences in political affiliation has decreased.

8.2 Differences in variance

Another common significance test in social science is the **variance test** which consists of testing whether the variances of the same variable across two groups are equal. This is usually achieved by testing whether the ratio of the variance between the two groups is significantly different from zero. With the BSA data, this amounts to testing whether men and women are more homogeneous with regard to their political views.

The syntax for the variance test `var.test()` also included in `stats` is almost identical to that of `t.test()`

```
# Testing for gender differences in liberal vs authoritarian score
var.test(libauth~Rsex.f,
         data=bsa)
```

```
    F test to compare two variances

data:  libauth by Rsex.f
F = 1.0892, num df = 1434, denom df = 1777, p-value = 0.0879
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.9873927 1.2022204
sample estimates:
ratio of variances
          1.089239
```

Significant differences in the variance between men and women was observed, but only at the .1 threshold.

```r
# Testing for whether men have a lower (ie more left-wing)  score
var.test(leftrigh~Rsex.f,
         data=bsa,
         alternative="greater")
```

```

    F test to compare two variances

data:  leftrigh by Rsex.f
F = 1.3218, num df = 1433, denom df = 1771, p-value = 1.263e-08
alternative hypothesis: true ratio of variances is greater than 1
95 percent confidence interval:
 1.217167      Inf
sample estimates:
ratio of variances 
            1.3218 
```

The variance of left-right political leaning is larger among men than women, in other words there are more divergence between men than between women.

8.3 Significance of measures of association

Between continuous variables

Another common statistical test in social science examines whether a coefficient of correlation is significantly different from 0 (alternative hypothesis).

```r
cor.test(bsa$leftrigh, bsa$libauth,
         use='complete.obs')
```

```

    Pearson's product-moment correlation

data:  bsa$leftrigh and bsa$libauth
t = 0.54472, df = 3202, p-value = 0.586
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.02501074  0.04423951
sample estimates:
        cor 
0.009625928 
```

As we could have suspected, the correlation coefficient between the two scales is too small to be considered significantly different from zero.

Between categorical variables

Let us go back to an earlier example, and test whether gender differences in political affiliations are due to chance or not using a chi-squared test of independence .

The chi-squared test is a very common test of association between categorical variables. It consists in examining whether a pattern of association between two variables is likely to be random or not, in other words whether the variability observed in a contingency table is significantly different from what could be expected were it due to chance.

We will be using `chisq.test()`, also from the `stats` package. By contrast with the test of correlation, the `chisq.test()` needs to be applied to contingency tables that have already been computed separately.

```
t<-xtabs(~PartyId2.f +Rsex.f,bsa)
chisq.test(t)
```

```

    Pearson's Chi-squared test

data:  t
X-squared = 27.191, df = 5, p-value = 5.236e-05
```

As the R output shows, there are highly significant gender differences in political affiliations (p<.001).

Does this remain true if we account for the survey design, as we did above for the t test? The `survey` package also has a survey design version of the chi square test:

```
svychisq(~PartyId2.f +Rsex.f, # We directly specify the contingency table here
         bsa.design,
          statistic = "Chisq" # And we specify the kind of test we would like
         )
```

```

    Pearson's X^2: Rao & Scott adjustment

data:  svychisq(~PartyId2.f + Rsex.f, bsa.design, statistic = "Chisq")
X-squared = 12.069, df = 5, p-value = 0.1126
```

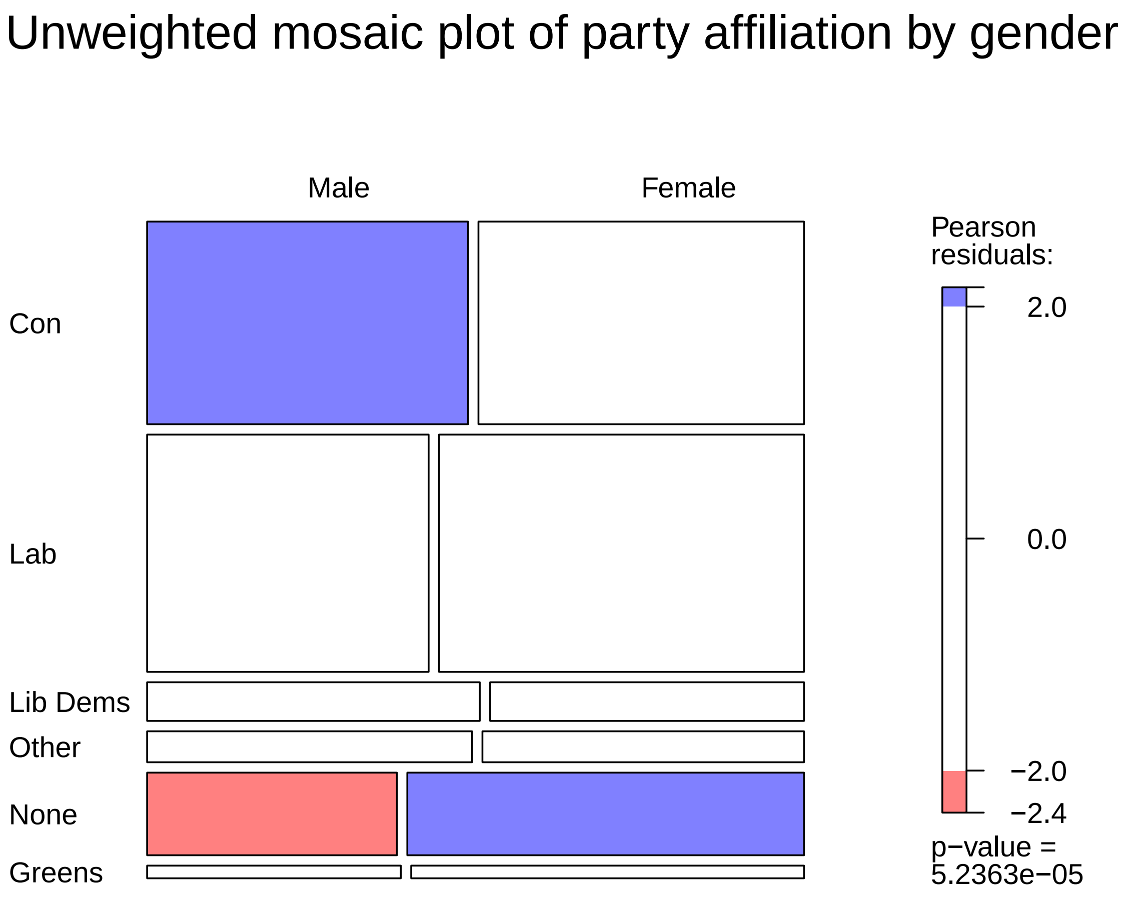
Interestingly this time, when accounting for survey design, sampling and non-response, gender differences in political affiliations are not significant anymore.

Visualising association in contingency tables with mosaic plots

In the previous chapter we used mosaic plots for representing contingency tables of political party affiliation by gender. A nice feature of these plots is that they can also be used to visualise significant deviations between observed and expected values.

This relies on a function specified with the `gp=` option which defines the shading of the colours of the respective cells according to the size of these deviations fom expected values, also known as residuals. Thresholds for shading can be customised as required.

```
mosaic(t,
       shade=T,
       gp = shading_hsv,               # shading function
       labeling=labeling_border(
         rot_labels = c(0,0,0,0),      # no label rotation on any plot facet
         varname=F,                    # no  variable names on the plot
         just_labels="left",           # labels left justified
         gp_labels=gpar(fontsize = 12),# label font size
         offset_labels = c(0, 0, 0, 3) # margins between label and plot facet
         ),
main = "Unweighted mosaic plot of party affiliation by gender" # Plot title
         )
```



The red and blue shaded rectangles in the figure above denote respectively lower and higher numbers of observations than expected if the two variables were independent from each other.

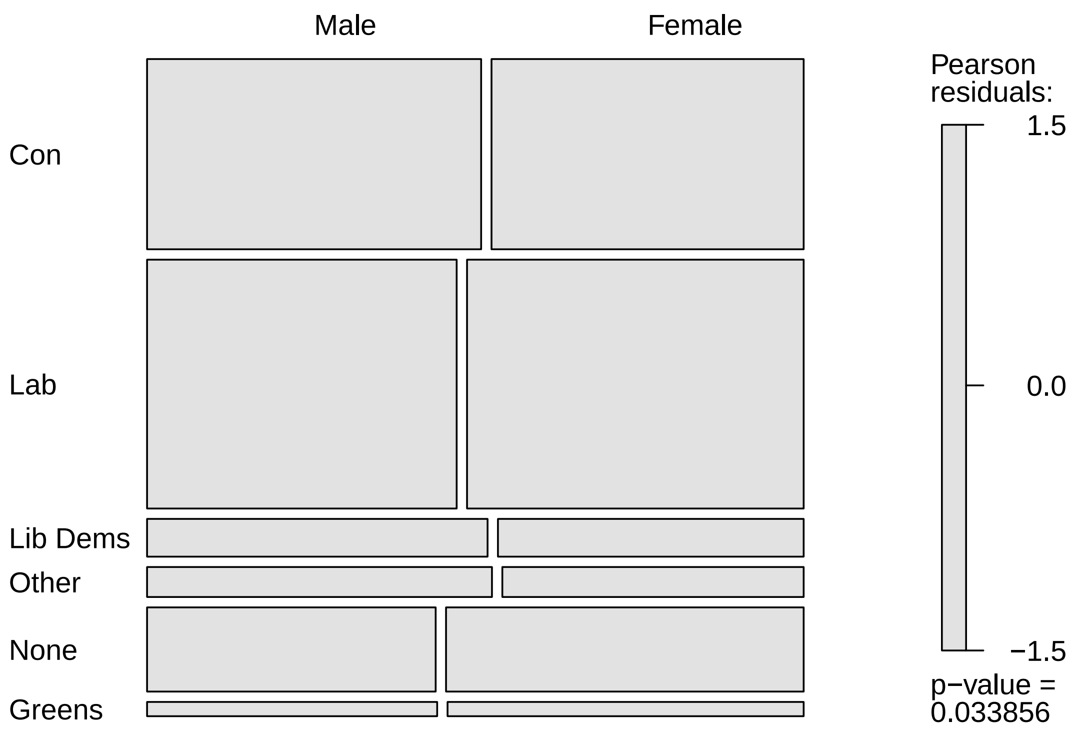
Another convenient feature of `mosaic()` is that it readily accepts weighted contingency tables produced by the `survey` package. If we repeat what we did in Chapter 6, namely, declare the survey design, and perform a survey design-informed chi square test, we can then feed the weighted frequencies computed by `svychisq` into the mosaic plot.

```
library(survey) ### Loading the package in memory
bsa.design<-svydesign(ids =~1,
                      weights=~WtFactor,
                      data=bsa)
t<-svychisq(~PartyId2.f+Rsex.f,bsa.design,statistic = "Chisq")$observed
```

The plot below does not display shades of blue or red anymore, reflecting the fact that the weighted distributions of political party affiliation and gender are weakly associated.

```
mosaic(t,
       shade=T,
       gp = shading_hcl,
       labeling=labeling_border(
         rot_labels = c(0,0,0,0),
         varname=F,
         just_labels="left",
         gp_labels=gpar(fontsize = 12),
         offset_labels = c(0, 0, 0, 3)
),
main = "Weighted mosaic plot of party affiliation by gender"

)
```

Weighted mosaic plot of party affiliation by gender



9 Regression analysis

Regression is one of the most common modelling tools used in social science. The `glm()` function from Base R can be used for fitting linear and non-linear models. These include OLS regression, logistic/probit regression, and more generally any model falling under the Generalized Linear Model (GLM) framework.
In this section, we will use it to investigate the association between the level of education and whether someone voted or not at the last general elections. But let's first briefly explore the variables using the `class()` and `table()`:

```
class(bsa$HEdQual3.f)
```

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

```
table(bsa$HEdQual3.f)
```

```

                       Degree Higher educ below degree/A level
                         1050                             1086
          O level or equiv/CSE                 No qualification
                         1026                              747
```

and

```
class(bsa$Voted)
```

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

```
table(bsa$Voted)
```

```

   1    2
2205  776
```

We converted earlier `HEdQual3` into a factor variable, but as we can see above, `Voted` is still a labelled numeric variable. It is a good idea to convert it to a factor as well. This is not absolutely necessary, but gives greater flexibility, for instance if we need to change the reference category on the go in the regression model.

```
bsa$Voted.f<-droplevels(as_factor(bsa$Voted)) # As before, factor conversion
levels(bsa$Voted.f)                           # Note that Yes comes before No
```

```
[1] "Yes" "No"
```

For greater readability we can also shorten the levels of `HEdQual3.f` using the `level()` function:

```
levels(bsa$HEdQual3.f) ### The original level names  are a bit verbose...
```

```
[1] "Degree"                          "Higher educ below degree/A level"
[3] "O level or equiv/CSE"            "No qualification"
```

```
### ... We can shorten them easily
levels(bsa$HEdQual3.f) <- c("Degree","A level and above","GCSE or equiv","No Qual")

table(bsa$HEdQual3.f)
```

```
           Degree A level and above     GCSE or equiv           No Qual 
             1050              1086              1026               747 
```

What about the levels for our dependent variable? By default, the first level of a factor will be used as the reference category. This can be also checked with the `contrasts()` function. In this case, 1 is associated with 'No', so the model will be predicting the probability of NOT voting. If the 1 was associated with 'Yes' then the model will be predicting the probability of voting.

```
contrasts(bsa$Voted.f)
```

```
    No
Yes  0
No   1
```

As we are interested in the latter, we need to change the reference category using the `relevel()` function. We will create a new variable named `Voted2` so as to keep the original variable intact.

```
# Reverse the order
bsa$Voted2 <- relevel(bsa$Voted.f, ref = "No")

# Check the contrasts
contrasts(bsa$Voted2)
```

```
    Yes
No    0
Yes   1
```

Since the outcome variable (`Voted2`) has a binomial distribution, we need to specify to the `glm()` function that we will be fitting a logistic regression model. We will do this by setting the argument `family` to 'binomial' and the `link` function to 'logit'. We could also have used 'probit' instead as a link function. The code below runs the model and stores the result into an object called `fit1`:

```
fit1 <- glm(Voted2 ~ HEdQual3.f,
            data=bsa,
            family=binomial(link=logit)
            )

summary(fit1)
```

```
Call:
glm(formula = Voted2 ~ HEdQual3.f, family = binomial(link = logit),
    data = bsa)

Coefficients:
                            Estimate Std. Error z value Pr(>|z|)
(Intercept)                  1.49561    0.09188  16.278  < 2e-16 ***
HEdQual3.fA level and above -0.21342    0.12514  -1.706   0.0881 .
HEdQual3.fGCSE or equiv     -0.64062    0.12191  -5.255 1.48e-07 ***
HEdQual3.fNo Qual           -0.83672    0.12769  -6.553 5.65e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 3297.6  on 2916  degrees of freedom
Residual deviance: 3240.4  on 2913  degrees of freedom
  (1071 observations deleted due to missingness)
AIC: 3248.4

Number of Fisher Scoring iterations: 4
```

To run a model controlling for gender `Rsex`and age `RAgeCat`, one simply needs to add them on the right-hand side of the formula, separated with a plus (+) sign.

```
fit2 <- glm(Voted2 ~ HEdQual3.f + Rsex.f + RAgeCat.f,
            data=bsa,
            family=binomial(link=logit)
            )

summary(fit2)
```

```
Call:
glm(formula = Voted2 ~ HEdQual3.f + Rsex.f + RAgeCat.f, family = binomial(link = logit),
    data = bsa)

Coefficients:
                            Estimate Std. Error z value Pr(>|z|)
(Intercept)                  1.11251    0.20044   5.550 2.85e-08 ***
HEdQual3.fA level and above -0.38676    0.13215  -2.927 0.003427 **
```

```
HEdQual3.fGCSE or equiv     -0.99023    0.13109  -7.554 4.23e-14 ***
HEdQual3.fNo Qual           -1.90625    0.15687 -12.152  < 2e-16 ***
Rsex.fFemale                -0.15708    0.09218  -1.704 0.088363 .
RAgeCat.f25-34              -0.24604    0.19670  -1.251 0.210996
RAgeCat.f35-44               0.20668    0.19808   1.043 0.296764
RAgeCat.f45-54               0.85685    0.20000   4.284 1.83e-05 ***
RAgeCat.f55-59               0.84062    0.23225   3.619 0.000295 ***
RAgeCat.f60-64               1.60276    0.25272   6.342 2.27e-10 ***
RAgeCat.f65+                 2.16408    0.21450  10.089  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 3293.1  on 2912  degrees of freedom
Residual deviance: 2922.5  on 2902  degrees of freedom
  (1075 observations deleted due to missingness)
AIC: 2944.5

Number of Fisher Scoring iterations: 4
```

Model interpretation

`summary()` provides a broad overview of the model output, comparable to other statistical software. But `fit1` and `fit2` contain more information than what is displayed by `summary()`. For an overview, one can type:

```
ls(fit1)
```

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

… Which displays a list of all the content items stored by it. We can request specific elements, the regression coefficients, by specifying its name following the `$` sign:

```
fit1$coefficients # extracting regression coefficients
```

```
                (Intercept) HEdQual3.fA level and above
                  1.4956126                  -0.2134240
    HEdQual3.fGCSE or equiv           HEdQual3.fNo Qual
                 -0.6406223                  -0.8367243
```

Shortcuts to some of these contents are available as functions such as `coef()` to extract the regression coefficients, or `deviance()` for the log-likelihood of the fitted model.

```
ls(fit2)
```

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

```
round(fit2$coefficients,2)
```

```
              (Intercept) HEdQual3.fA level and above
                     1.11                       -0.39
    HEdQual3.fGCSE or equiv            HEdQual3.fNo Qual
                    -0.99                       -1.91
             Rsex.fFemale               RAgeCat.f25-34
                    -0.16                       -0.25
           RAgeCat.f35-44               RAgeCat.f45-54
                     0.21                        0.86
           RAgeCat.f55-59               RAgeCat.f60-64
                     0.84                        1.60
             RAgeCat.f65+
                     2.16
```

```
### The coef() function will give the same output:
round(coef(fit2),2)
```

```
              (Intercept) HEdQual3.fA level and above
                     1.11                       -0.39
    HEdQual3.fGCSE or equiv            HEdQual3.fNo Qual
                    -0.99                       -1.91
             Rsex.fFemale               RAgeCat.f25-34
                    -0.16                       -0.25
           RAgeCat.f35-44               RAgeCat.f45-54
                     0.21                        0.86
           RAgeCat.f55-59               RAgeCat.f60-64
                     0.84                        1.60
             RAgeCat.f65+
                     2.16
```

It is beyond the remit of this guide to describe the full output of `glm()`. See the `stats` package documentation for more detailed explanations.

Computing odds ratios

Standard logistic regression coefficients measure the logarithmic effect of variables on the probability of the outcome such as $log(\beta_X) = P(y)$. It is common practice to convert these into odd ratios by exponentiating them, such as that $\beta_X = exp(P(y))$.

Using the `coef()` and `confint()` functions, the code above respectively extracts the coefficients and the associated 95% confidence intervals from `fit2` then collate them using `cbind()`.

```
cbind(
  Beta_x=exp(
    coef(fit2)
    ),
  exp(
    confint(fit2)
    )
)
```

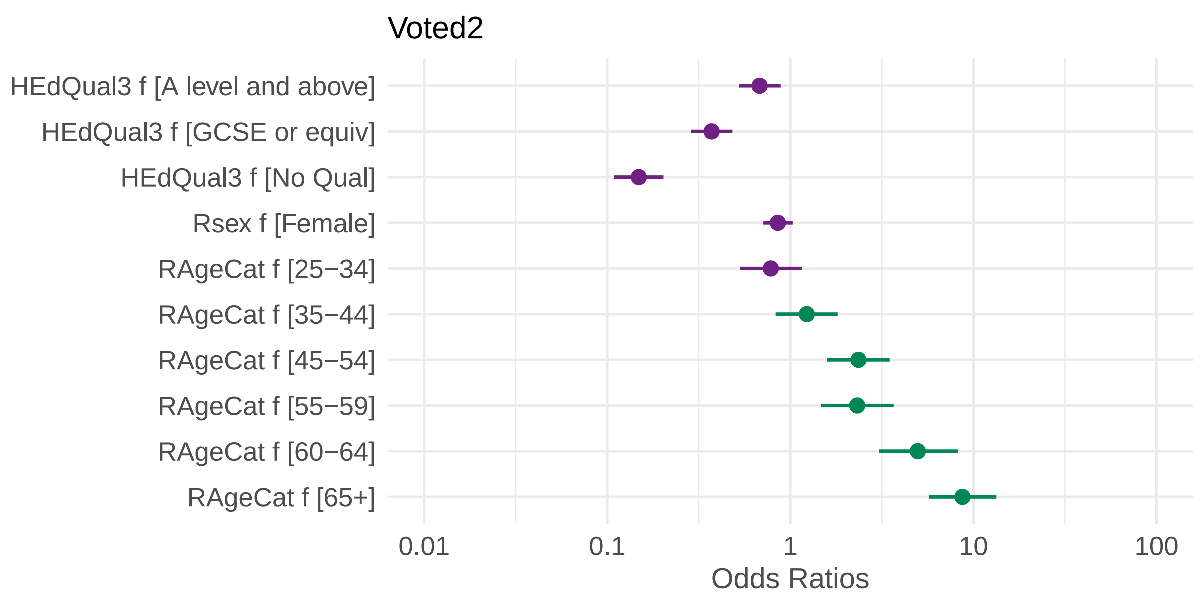
```
                                Beta_x     2.5 %     97.5 %
(Intercept)                  3.0419750 2.0618311  4.5276614
HEdQual3.fA level and above 0.6792550 0.5237628  0.8795335
HEdQual3.fGCSE or equiv     0.3714919 0.2868078  0.4796010
HEdQual3.fNo Qual           0.1486366 0.1089442  0.2015607
Rsex.fFemale                0.8546343 0.7130680  1.0235515
RAgeCat.f25-34              0.7818917 0.5300310  1.1469983
RAgeCat.f35-44              1.2295882 0.8316608  1.8095457
RAgeCat.f45-54              2.3557310 1.5886305  3.4827615
RAgeCat.f55-59              2.3178122 1.4718049  3.6621685
RAgeCat.f60-64              4.9667132 3.0428167  8.2090111
RAgeCat.f65+                8.7065916 5.7183039 13.2696921
```

Plotting regression coefficients

We can visualise the odd ratios and their confidence intervals using the `plot.model()` function from the `sjPlot` package. If not already present, `sjPlot` needs to be installed and loaded first.

```
install.packages('sjPlot')
```

```
library(sjPlot)
set_theme(base = theme_minimal()) ### Default sets of options
plot_model(fit2,
           colors = c("#702082", "#008755") ### Added for better accessibility
)
```



Assessing model fit

The Akaike Information Criterion (AIC) is a measure of relative fit for maximum likelihood fitted models. It is used to compare the improvement in how several models fit some data relative to each other, allowing for the different number of parameters or degrees of freedom. The smaller the AIC, the better the fit. In order for the comparison to be valid, we need to ensure that the models were run with the same number of observations each time. As it is likely that the second model was run on a smaller sample, due to missing values for the Age and Sex variables, we will need to re-run the first one without these.

```
fit1 <- glm(Voted2 ~ HEdQual3.f,
            data=bsa%>%filter(!is.na(Rsex.f) & !is.na(RAgeCat.f)),
            family=binomial(link=logit))

c(fit1$aic,fit2$aic)
```

```
[1] 3244.507 2944.535
```

We can see that the model controlling for gender and sex is a better fit to the data than the one without controls as it has an AIC of 2944.5 against 3244.5 for fit1.

With the information about the deviance from `fit1` and `fit2`, we can also compute the overall significance of the model, that is whether the difference between the deviance (another likelihood-based measure of fit) for the fitted model is significantly different from that of the empty or null model. This is usually carried out by conducting a chi square test, accounting for the differences in the number of parameters (ie degrees of freedom) between the two models. As with other R code, this can be achieved step by step or in one go:

```
dev.d<-fit2$null.deviance - fit2$deviance # Difference in deviance
df.d<-fit2$df.null - fit2$df.residual     # ... and degrees of freedom
p<-1 - pchisq(dev.d, df.d)
c(dev.d,df.d,p)
```

```
[1] 370.5486  10.0000   0.0000
```

The Chi square test indicates that the difference in deviance of 370.5 with 10 degrees of freedom is highly significant (P<.001)

10 Further information

Packages of interest

The following non exhaustive list provides a few examples of commands and packages that tackle common types of analysis which might be relevant to users of large scale social surveys:

- Further regression analysis: the `glm()` command can be used for fitting a large number of regression models including Poisson and multinomial logistic regression. The packages `lme4` and `nlme` include functions to fit respectively linear and non linear multilevel models, also known as mixed models.
- For users interested in latent variable modelling the `factanal()` command from the `stats` package enables to conducts factor analysis. Other resources are available in the `poLCA` (Latent Class Analysis), `ltm` (Latent Trait modelling), `sem` (Structural Equation Modelling) packages. The `Lavaan` package also provides a wide range of functions for structural equation modelling including models with categorical outcomes.
- For those conducting longitudinal and time series analysis the `stats`and the 'tseries' packages contains useful functions. The packages `survival` and `eha` deal with event history and survival analysis, whereas `plm` is designed for panel data and fixed and random effects models.
- Using R for creating maps is now common among social scientists and geographers with packages such as `rmaps`, `sp`, `rgdal`, `rgeos` and `ggmaps`

Additional online resources

There are hundreds of web sites dedicated to R, in addition to CRAN and the main R help list, R-Help with its searchable archives. A few common ones are listed below:

- As with other statistical packages, the UCLA and Princeton University websites provide a good starting point for beginners
- The University of North Texas provides useful links to R resources
- The R Bloggers website contains many posts about R.
- Stackexchange is not specific to R but contains many forum-type questions and answers raised by R users
- This website at Harding University presents useful information about basic plots in R.
- This blog presents detailed tutorials for advanced data visualisation using `ggplot2`
- The Centre for Multilevel modeling at Bristol University has several pages and training courses dedicated to R users interested in Multilevel modelling.
- The UK Data Service has produced training material about creating maps in R, as part of an introduction to mapping crime data

11 References

- R Core Team. (2023). R: A language and environment for statistical computing. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from https://www.r-project.org/ RStudio Team.
- RStudio: Integrated Development for R. Boston, USA: RStudio, Inc. Retrieved from http://www.rstudio.com/
- Tennekes, M. (2018) tmap: Thematic Maps. R package version 1.10. Retrieved from https://cran.r-project.org/package=tmap
- Wickham, H., & Francois, R. (2023). dplyr: A Grammar of Data Manipulation. R package version 1.1.4. Retrieved from https://cran.r-project.org/package=dplyr
- Wickham, H. ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York, 2016. Retrieved from https://cran.r-project.org/package=ggplot2