Data manipulation with R

Jordi Llorens*

September, 2020

1 Data import/export

It is evident that before performing any analysis in R you need to import the data of interest. But data come in many different formats, so you'll have to adapt and learn. Luckily, R has a plethora of input options[1], among them Excel files and databases. Through some useful packages, such as `foreign` or `xlsReadWrite`, many other data formats can be imported as well, such as SAS, SPSS or STATA.

Before getting any data into R, it is advisable to create a data directory in your machine where to store it. We will use it as the working directory - the default path where R will look to when importing/exporting data.

1.1 Importing native .RData files

This is the simplest way, but usually datasets are not disseminated publicly in this format. You can use `load` function to load such a file. Word of caution. Be careful with loading such files, they might contain many R objects, some of them with same names that you have in your current working environment. If this is the case, when you load the file, objects with same names in your environment will be overwritten.

```
load('data/mtcars.Rdata')
```

1.2 Importing plain text files

One of the standard ways to exchange data (specially in organizations with a low IT profile) are plain text files. There are two paradigms for storing data tables in plain text files: **delimited text** and **fixed width**. In both cases they may or may not have a **header**

*I am truly indebted to Hrvoje Stojic, who originally created these materials for several offerings of the brush-ups in Data Science. The current handout is a re-adaptation of those.

[1] See R Data Import/Export at cran.r-project.org/doc/manuals/r-release/R-data.pdf

(column names) and before importing you can explore its contents with a plain text editor or a spreadsheet program (except if the file is huge, some programs may not support too many gigabytes).

In **delimited text files** the data columns are explicitly delimited, typically with `;`, `,` or a tabulator (coded as `\t`). You may find other characters as separators. The standard file is **CSV** (comma-separated values, `,`).

The function `read.table` can import most of the delimited files you will find. Some non-standard files may need other specific functions, or they might be simply too big. Here we cover only `read.table`.

Download the *Demographic data of census sections in Barcelona* from BCN Open Data.[2] to your working directory. It contains demographic information for each census division. Inspect it with a plain text editor (if you cannot understand the headers look at their online description). This is an example of a standard import - it has a header row (column names) and the field separator is `;`.

```
## Import
censusBCN <- read.table("data/MAP_SCENSAL.csv", header = TRUE,
                        sep = ";")

## Translating headers into English
names(censusBCN)

##  [1] "DATA_DADES"    "HOMES"         "SECCIO_CENSAL" "DONES"
##  [5] "EDAT_0_A_14"   "EDAT_15_A_24"  "EDAT_25_A_64"  "EDAT_65_A_MES"
##  [9] "NACIONALS"     "COMUNITARIS"   "ESTRANGERS"

names(censusBCN) <- c("Date", "Men", "CensusDivision",
    "Women", "AGE_0_14", "AGE_15_A_24", "AGE_25_A_64",
    "AGE_65_plus", "NATIONALS", "EUCommunity",
    "Overseas")

## Data summary
summary(censusBCN)

##      Date                Men         CensusDivision      Women
##  Length:1068        Min.   : 285.0   Min.   : 1001   Min.   : 261.0
##  Class :character   1st Qu.: 593.0   1st Qu.: 3041   1st Qu.: 667.0
##  Mode  :character   Median : 683.0   Median : 6036   Median : 777.0
##                     Mean   : 712.2   Mean   : 5784   Mean   : 792.4
##                     3rd Qu.: 791.0   3rd Qu.: 8092   3rd Qu.: 888.0
##                     Max.   :2085.0   Max.   :10237   Max.   :1607.0
##     AGE_0_14    AGE_15_A_24     AGE_25_A_64      AGE_65_plus      NATIONALS
##  Min.   : 66   Min.   : 45.0   Min.   : 330.0   Min.   : 92.0   Min.   : 462
```

[2] Link to data

```
##  1st Qu.:144   1st Qu.:104.0   1st Qu.: 699.0   1st Qu.:275.8   1st Qu.:1083
##  Median :173   Median :125.0   Median : 820.0   Median :324.0   Median :1240
##  Mean   :188   Mean   :131.3   Mean   : 855.6   Mean   :329.7   Mean   :1263
##  3rd Qu.:215   3rd Qu.:150.0   3rd Qu.: 960.0   3rd Qu.:375.0   3rd Qu.:1412
##  Max.   :652   Max.   :412.0   Max.   :2204.0   Max.   :703.0   Max.   :2847
##   EUCommunity        Overseas
##  Min.   :  3.00   Min.   :  10.0
##  1st Qu.: 29.00   1st Qu.:  91.0
##  Median : 53.00   Median : 134.0
##  Mean   : 70.10   Mean   : 171.9
##  3rd Qu.: 89.25   3rd Qu.: 199.0
##  Max.   :454.00   Max.   :1733.0
```

```
## Computing a new column: percent of senior
## citizens
censusBCN$percentSenior <- censusBCN$AGE_65_plus /
    (censusBCN$Men + censusBCN$Women)
summary(censusBCN$percentSenior)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.04845 0.19616 0.22296 0.22383 0.25212 0.48013
```

Download the Aging Population file from the UK Open Data site to your working directory[3]. Inspect it with a plain text editor, and you will see the key import characteristics - it has a header row (column names), the field separator is `,` , and some strings are quoted with double quotation marks. With this in mind we can import it.

```
## Import
agingPopulation <- read.table("data/aging-population-2008.csv",
                              header = TRUE,
                              sep = ",",
                              quote = "\"")

## Rename last column
names(agingPopulation)
```

```
## [1] "SOA.Code"
## [2] "SOA.Name"
## [3] "Ward.Name"
## [4] "Locality"
## [5] "District.Borough"
## [6] "Total.Population"
## [7] "X..of.Total.Population.aged.60...Females...65...Males."
```

[3] opendata.s3.amazonaws.com/aging-population-2008.csv

```
names(agingPopulation)[7] <- "PercentSeniors"

## Data summary
summary(agingPopulation)
```

```
##    SOA.Code           SOA.Name           Ward.Name           Locality
##  Length:333         Length:333         Length:333         Length:333
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##  District.Borough   Total.Population   PercentSeniors
##  Length:333         Length:333         Length:333
##  Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character
```

```
sapply(agingPopulation, class)
```

```
##         SOA.Code         SOA.Name        Ward.Name         Locality
##      "character"      "character"      "character"      "character"
## District.Borough Total.Population   PercentSeniors
##      "character"      "character"      "character"
```

Fixed-width files have no delimiter between columns, so you will need a description defining the width of each column. We will work out an example from the INE (Spanish statistical office): Survey on Human Resources in Science and Technology 2009.[4] INE's microdata (individual responses to surveys) are usually stored as fixed/width files accompanied with a metadata spreadsheet. Download both the raw data[5] and its metadata[6], and import it into R.

```
## Metadata for the selected columns
RRHH09Widths <- c(6, 2, 4, 2, 4, 1, 4, 4, 4, 1,
                  1, 2, 2, 2, 2, 2, 1, 1)

RRHH09Names <-c("MUIDENT", "CCAARESI", "ANONAC",
                "CCAANAC", "CONTNACIM", "RELA", "CONTNAC1",
                "CONTNAC2", "CONTNAC3", "SEXO", "ESTADOCIVIL",
                "DEPEN5", "DEPEN18", "DEPENMAS", "NIVESTPA",
                "NIVESTMA", "NIVPROFPA", "NIVPROFMA")

## Fixed-width import function
fwfDataFrame <- read.fwf(file = "data/RRHH09.txt",
                         n = 100,
                         widths = RRHH09Widths,
                         col.names = RRHH09Names)
```

[4] ine.es/en/prodyser/microdatos_en.htm

[5] ftp://www.ine.es/temas/recurciencia/micro_recurciencia.zip

[6] ftp://www.ine.es/temas/recurciencia/disreg_recurciencia.xls

```
## Data summary
summary(fwfDataFrame)
```

```
##     MUIDENT         CCAARESI        ANONAC       CCAANAC
##  Min.   :10003   Min.   : 8.0   Min.   :1940   Length:100
##  1st Qu.:20007   1st Qu.:10.0   1st Qu.:1961   Class :character
##  Median :30008   Median :10.0   Median :1968   Mode  :character
##  Mean   :23433   Mean   :10.8   Mean   :1966
##  3rd Qu.:30049   3rd Qu.:10.0   3rd Qu.:1971
##  Max.   :30096   Max.   :16.0   Max.   :1979
##   CONTNACIM              RELA        CONTNAC1           CONTNAC2
##  Length:100         Min.   :1.00   Length:100         Length:100
##  Class :character   1st Qu.:1.00   Class :character   Class :character
##  Mode  :character   Median :1.00   Mode  :character   Mode  :character
##                     Mean   :1.04
##                     3rd Qu.:1.00
##                     Max.   :3.00
##    CONTNAC3              SEXO       ESTADOCIVIL       DEPEN5        DEPEN18
##  Length:100         Min.   :1.00   Min.   :1.00   Min.   :0.00   Min.   :0.00
##  Class :character   1st Qu.:1.00   1st Qu.:1.00   1st Qu.:0.00   1st Qu.:0.00
##  Mode  :character   Median :1.00   Median :1.00   Median :0.00   Median :0.00
##                     Mean   :1.46   Mean   :1.93   Mean   :0.38   Mean   :0.72
##                     3rd Qu.:2.00   3rd Qu.:1.00   3rd Qu.:1.00   3rd Qu.:1.00
##                     Max.   :2.00   Max.   :6.00   Max.   :2.00   Max.   :4.00
##     DEPENMAS       NIVESTPA       NIVESTMA      NIVPROFPA      NIVPROFMA
##  Min.   :0.00   Min.   : 1.0   Min.   :1.00   Min.   :1.00   Min.   :1.00
##  1st Qu.:0.00   1st Qu.: 2.0   1st Qu.:2.00   1st Qu.:1.00   1st Qu.:1.00
##  Median :0.00   Median : 3.0   Median :3.00   Median :1.00   Median :5.00
##  Mean   :0.27   Mean   : 4.5   Mean   :3.59   Mean   :1.26   Mean   :3.41
##  3rd Qu.:0.00   3rd Qu.: 7.0   3rd Qu.:6.00   3rd Qu.:1.00   3rd Qu.:5.00
##  Max.   :3.00   Max.   :10.0   Max.   :9.00   Max.   :3.00   Max.   :5.00
```

We imported only a sample of the data (the first 100 rows and the first 18 columns). It is usually a good idea for an initial exploration of the data.

Practice fixed-width import. Import all the columns of the `RRHH09.txt` file (but only 1000 rows). You will have to look at the metadata for the column widths and names. Summarize the columns, in particular the labor market situation (SITLAB). Is the employment rate among respondents high? (decypher its code values reading the metadata).

1.3 Exporting data

After any serious data analysis we might want to output its results. The most common function to export data in R is `write.table`. Not by chance, the name reminds of the import function `read.table`. If no folder is specified, the file will be saved at the working directory.

Before writing data into a plain text file, we must make decisions about its output format: the field separator string, whether to output the column and row names, whether to quote strings or not, or the decimal separator.

Let us see an example by exporting the `mtcars` data:

```
# Typical csv export
write.table(mtcars, file = "data/mtcars.csv", sep = ",",
            quote = FALSE, row.names = FALSE)

# Custom export
write.table(mtcars, file = "data/mtcars.dat", sep = "\t",
            row.names = TRUE, dec = ",")
```

Saving in the native .RData file allows you to save multiple objects in any format, while saving in text files you are usually constrained to saving data frames. First argument specifies the object, second the name of the file to be saved.

```
save(mtcars, file="mtcars.RData")
```

Same as with import, the packages like `foreign` and `xlsReadWrite` make it easy to export data in proprietary formats[7], such as MS Excel, SPSS, SAS, or Stata.

2 Transforming data

Transforming datasets and variables is essential in any data-oriented project. You will need, even for the most basic programming task, to select rows from a data frame or to merge two data sets.

2.1 Subsetting in more details

When we introduced data frames we already saw how to select specific parts of a data set (given certain conditions). We will refresh the basics and dig a little deeper.

Subsetting in data frames uses indices on rows/columns: `[optional rows condition, optional columns condition]`. Note that you can use negative numbers to indicate that the rows/columns with those indices should be *removed*.

Logical conditions are very often used to subset the data. The idea is to produce a logical vector whose length will be the same as the number of rows (if we want to subset according to rows) or the number of columns. Then, those rows indicated with `TRUE` will be produced

[7] statmethods.net/input/exportingdata.html

as an output of subsetting. We have following **logical operators** that we can use in R: `<`, `>`, `<=`, `>=`, `!=` and `==`.

```
# Logical conditions on data frame values
mtcars[mtcars$hp > 200, ][1:5,]
```

```
##                      mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
## Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
## Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
```

```
mtcars[mtcars$cyl == 6, ][1:5,]
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

```
mtcars[mtcars$cyl != 6, ][1:5,]
```

```
##                    mpg cyl  disp  hp drat   wt  qsec vs am gear carb
## Datsun 710        22.8   4 108.0  93 3.85 2.32 18.61  1  1    4    1
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.44 17.02  0  0    3    2
## Duster 360        14.3   8 360.0 245 3.21 3.57 15.84  0  0    3    4
## Merc 240D         24.4   4 146.7  62 3.69 3.19 20.00  1  0    4    2
## Merc 230          22.8   4 140.8  95 3.92 3.15 22.90  1  0    4    2
```

Note that the second brackets are there simply to shorten the output to the first 5 lines. Multiple conditions can be connected with **logical expressions**: `!`, `&`, `&&`, `|`, `||` and `xor` function. Inputs to these functions need to be logical vectors.

```
# Multiple conditions on data frame values
mtcars[mtcars$hp > 200  & mtcars$mpg > 14, ]
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Duster 360        14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
## Chrysler Imperial 14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
## Ford Pantera L    15.8   8  351 264 4.22 3.170 14.50  0  1    5    4
## Maserati Bora     15.0   8  301 335 3.54 3.570 14.60  0  1    5    8
```

```
mtcars[mtcars$hp >= 250 | mtcars$hp <= 65, ]
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
```

Conditions on both rows and columns.

```
mtcars[row.names(mtcars) %in% c("Fiat 128", "Fiat X1-9"),
       c("mpg", "cyl", "wt")]
```

```
##            mpg cyl    wt
## Fiat 128  32.4   4 2.200
## Fiat X1-9 27.3   4 1.935
```

Conditions using functions.

```
mtcars[substr(row.names(mtcars), 1, 4) == "Fiat",
c("mpg", "cyl", "wt")]
```

```
##            mpg cyl    wt
## Fiat 128  32.4   4 2.200
## Fiat X1-9 27.3   4 1.935
```

```
mtcars[mtcars$hp == max(mtcars$hp), ]
```

```
##               mpg cyl disp  hp drat   wt qsec vs am gear carb
## Maserati Bora  15   8  301 335 3.54 3.57 14.6  0  1    5    8
```

Using the library `data.table`, you can invoke the function `like` for doing partial matching comparisons:

```
if(!require(data.table)) install.packages('data.table') else library(data.table)
'Fiat' %like% 'Fiat1' # this is FALSE
```

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

```
'Fiat1' %like% 'Fiat' # this is TRUE
```

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

```
mtcars[row.names(mtcars) %like% 'Fiat',]
```

```
##            mpg cyl disp hp drat    wt  qsec vs am gear carb
## Fiat 128  32.4   4 78.7 66 4.08 2.200 19.47  1  1    4    1
## Fiat X1-9 27.3   4 79.0 66 4.08 1.935 18.90  1  1    4    1
```

Using stored conditions.

```
hpPattern <- mtcars$hp >= 250 | mtcars$hp <= 65
mtcars[hpPattern, ]
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
```

2.2 Sorting

Sorting a vector:

```
sort(mtcars$hp, decreasing = TRUE)
```

```
##  [1] 335 264 245 245 230 215 205 180 180 180 175 175 175 150 150 123 123 113 110
## [20] 110 110 109 105  97  95  93  91  66  66  65  62  52
```

The subsetting notation can be also used for sorting data frames using the `order` function:

```
mtcars[order(mtcars$hp), ][1:5,]
```

```
##                 mpg cyl  disp hp drat    wt  qsec vs am gear carb
## Honda Civic    30.4   4  75.7 52 4.93 1.615 18.52  1  1    4    2
## Merc 240D      24.4   4 146.7 62 3.69 3.190 20.00  1  0    4    2
## Toyota Corolla 33.9   4  71.1 65 4.22 1.835 19.90  1  1    4    1
## Fiat 128       32.4   4  78.7 66 4.08 2.200 19.47  1  1    4    1
## Fiat X1-9      27.3   4  79.0 66 4.08 1.935 18.90  1  1    4    1
```

```
mtcars[order(mtcars$hp, decreasing = TRUE), ][1:5,]
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Maserati Bora     15.0   8  301 335 3.54 3.570 14.60  0  1    5    8
## Ford Pantera L    15.8   8  351 264 4.22 3.170 14.50  0  1    5    4
## Duster 360        14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
## Camaro Z28        13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
## Chrysler Imperial 14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
```

Ordering by multiple columns is straightforward (for clarity, we first store the row conditions on a vector):

```
# Index of sorted rows
hpOrder <- order(mtcars$hp, mtcars$mpg, decreasing = TRUE)

# Using the stored order conditions
mtcars[hpOrder, ][1:5,]
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Maserati Bora     15.0   8  301 335 3.54 3.570 14.60  0  1    5    8
## Ford Pantera L    15.8   8  351 264 4.22 3.170 14.50  0  1    5    4
## Duster 360        14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
## Camaro Z28        13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
## Chrysler Imperial 14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
```

2.3 Appending

To combine vectors you have already seen that you can use `c` function. We have used it until now to create atomic vectors, but depending on the objects you are combining, the output might be a list.

```
# combining objects in a vector
c(mtcars[1,1], mtcars[1,3])  # atomic vector
```

```
## [1]  21 160
```

```
c(mtcars[1,1], mtcars[1:3,1:3])  # list
```

```
## [[1]]
## [1] 21
## 
## $mpg
## [1] 21.0 21.0 22.8
## 
## $cyl
## [1] 6 6 4
## 
## $disp
## [1] 160 160 108
```

Binding together several data frames with a common structure is easy. To combine data frames column-wise and row-wise, you should use functions **`cbind`** and **`rbind`**.

```
# combining data frames
cbind(mtcars[,1], mtcars[,3])[1:5,]
```

```
##      [,1] [,2]
## [1,] 21.0  160
## [2,] 21.0  160
## [3,] 22.8  108
## [4,] 21.4  258
## [5,] 18.7  360
```

```
rbind(mtcars[1:2,], mtcars[5:6,])
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Be careful with the **broadcast** feature of R. Usually it is very useful, and you do not even notice it is at work, however, at other times, if you are not careful it can produce an undesired output that you will not notice - R will not show any warning as it assumes you know what you are doing.

```
# broadcasting allows hand abbreviations such as
x <- matrix(NA, 4, 4)

# instead of
matrix(rep(NA,16), 4, 4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   NA   NA   NA   NA
## [2,]   NA   NA   NA   NA
## [3,]   NA   NA   NA   NA
## [4,]   NA   NA   NA   NA
```

```
# comes handy in creating data frames
cbind(1, x)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1   NA   NA   NA   NA
## [2,]    1   NA   NA   NA   NA
## [3,]    1   NA   NA   NA   NA
## [4,]    1   NA   NA   NA   NA
```

```
# however, here it will broadcast y to fill out the structure
# if this was not an intention, it will be difficult
# to detect an error
y <- c(1,2)
cbind(y, x)
```

```
##      y
## [1,] 1 NA NA NA NA
## [2,] 2 NA NA NA NA
## [3,] 1 NA NA NA NA
## [4,] 2 NA NA NA NA
```

Practice ordering, subsetting, appending. Order the mtcars data frame by ascending number of carburators and weight, create two data frames with the top 3 and bottom 3 rows according to this order, append them both.

2.4 Merging

Adding data from a data frame to another data frame using some joining condition is an essential operation when manipulating data, because most information is stored in tables that relate to each other by some common identifier. A function that should be used for this purpose is merge.

```
authors <- data.frame(
     surname = I(c("Tukey", "Venables", "Tierney", "Ripley", "McNeil")),
     nationality = c("US", "Australia", "US", "UK", "Australia"),
     deceased = c("yes", rep("no", 4)))

books <- data.frame(
     name = I(c("Tukey", "Venables", "Tierney",
              "Ripley", "Ripley", "McNeil", "R Core")),
     title = c("Exploratory Data Analysis",
               "Modern Applied Statistics ...",
               "LISP-STAT",
               "Spatial Statistics", "Stochastic Simulation",
               "Interactive Data Analysis",
               "An Introduction to R"),
     other.author = c(NA, "Ripley", NA, NA, NA, NA,
                      "Venables & Smith"))

merge(authors, books, by.x = "surname", by.y = "name")
```

```
##    surname nationality deceased                         title other.author
## 1   McNeil   Australia       no     Interactive Data Analysis         <NA>
## 2   Ripley          UK       no            Spatial Statistics         <NA>
## 3   Ripley          UK       no         Stochastic Simulation         <NA>
## 4  Tierney          US       no                     LISP-STAT         <NA>
## 5    Tukey          US      yes     Exploratory Data Analysis         <NA>
## 6 Venables   Australia       no Modern Applied Statistics ...       Ripley
```

2.5 Variable transformations

We have already seen how to create new variables from existing ones. Here we will look at some special (and useful) cases:

Sometimes you need to transform a numerical variable into a categorical one. For example, divide horsepower into 2 categories: low (below average) and high (above average).

A naive approach would be:

```
# Replicate the data frame (for keeping the
# original data unchanged)
mtcarsBis <- mtcars


# Create the above- and below-average bins
```

```
mtcarsBis$hpCateg[mtcarsBis$hp < mean(mtcarsBis$hp)] <- "Low"
mtcarsBis$hpCateg[mtcarsBis$hp >= mean(mtcarsBis$hp)] <- "High"
```

A more sophisticated way:

```
mtcarsBis$hpCateg <- ifelse(test = mtcarsBis$hp >
mean(mtcarsBis$hp), yes = "Low", no = "High")
```

Binning numerical variables into more than 2 categories could be tedious following the previous examples, but the `cut` function clears the way.

Practice: binning into multiple categories with `cut` function. Bin horsepower (from the mtcarsBis dataset) into 4 categories using the `cut` function. Add a new column to the dataset with this categorical values.

3 Data display

The first thing to do when having data at hand data is exploring it. We use the dataset `mtcars`, one of the several pre-loaded datasets in R, as an introductory example.[8]

First we must make sure the import process was successful (check the number of rows and columns, make sure the numeric fields are not imported as strings, etc.).

```
class(mtcars)

## [1] "data.frame"

str(mtcars)

## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...

dim(mtcars)

## [1] 32 11
```

[8] stat.ethz.ch/R-manual/R-devel/library/datasets/html/mtcars.html

```
names(mtcars)
```

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

```
summary(mtcars)
```

```
##       mpg             cyl             disp             hp
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0
##       drat             wt             qsec             vs
##  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000
##  Median :3.695   Median :3.325   Median :17.71   Median :0.0000
##  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
##  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
##        am              gear             carb
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
##  Median :0.0000   Median :4.000   Median :2.000
##  Mean   :0.4062   Mean   :3.688   Mean   :2.812
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
##  Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

```
sapply(mtcars, class)
```

```
##       mpg       cyl      disp        hp      drat        wt      qsec        vs
## "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##        am      gear      carb
## "numeric" "numeric" "numeric"
```

Then we can inspect visually our table. Since most of our tables have many more rows than our screens can show we start by looking at the top and bottom rows.

```
head(mtcars)
```

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

```
tail(mtcars)
```

```
##                 mpg cyl  disp  hp drat    wt qsec vs am gear carb
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
## Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.6  1  1    4    2
```

```
# you can specify the number of rows
head(mtcars, 10)
```

```
##                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

After this visual inspection we can describe individual columns, summary tables for categorical data, histograms and descriptive statistics for numeric data, etc.

4 Basic graphics

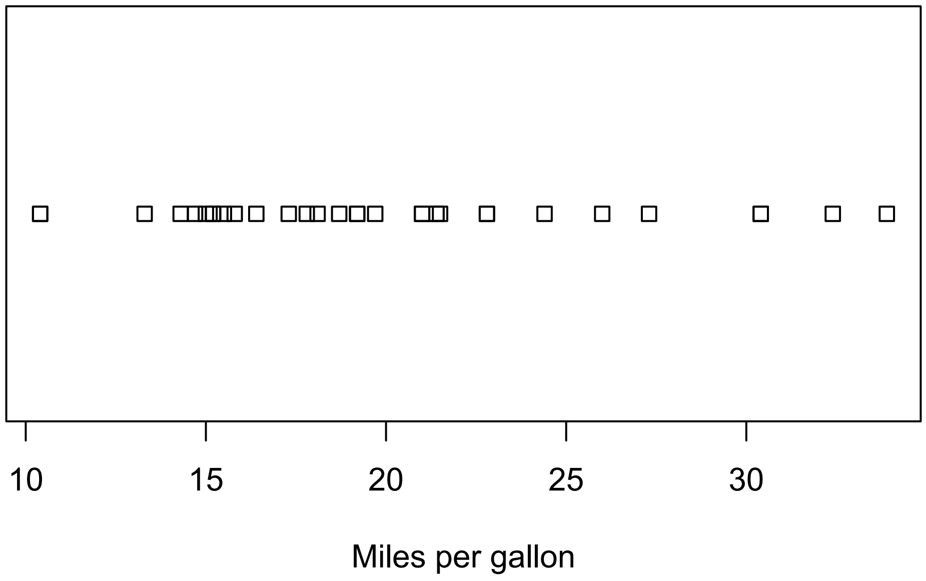
Visualizing results for your analysis is critical for its success - conveying the right message). R is extraordinary powerful at graphing data, allowing a great degree of personalization and having several state-of-the-art packages.

We will start with the basics[9], and later on we will use `ggplot2` - the package for advanced plotting in R.

Stripcharts are one-dimensional scatter plots and provide a (somewhat simplistic) first look at univariate series. Note the optional parameter `xlab` for setting the X-axis title.

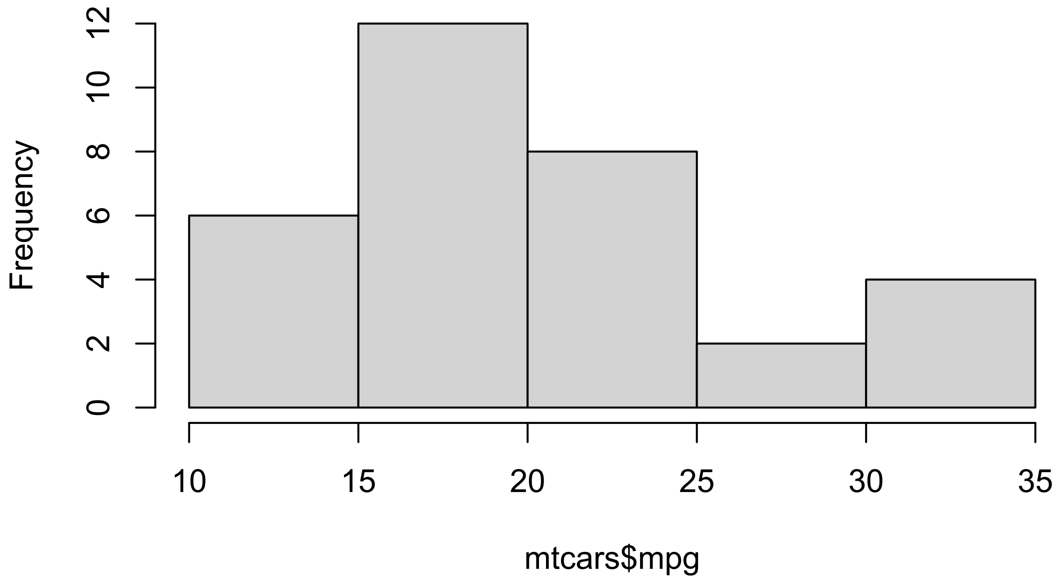
```
stripchart(mtcars$mpg, xlab = "Miles per gallon")
```

[9] statmethods.net/graphs/density.html



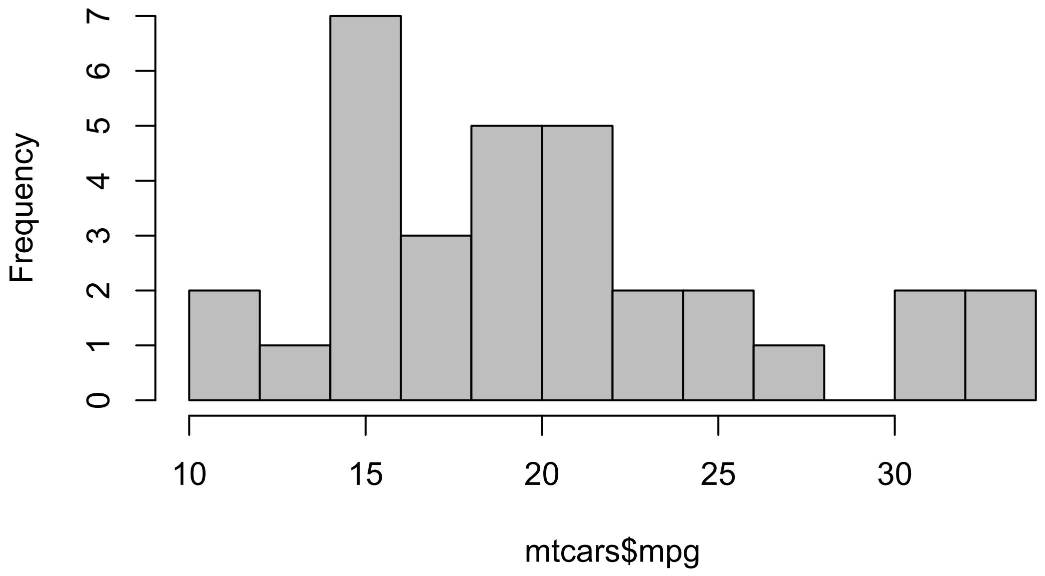
A **histogram** cuts series in discrete bins, while a continuous distribution varies smoothly along the series. Histogram of mileage in the `mtcars` dataset.

```
hist(mtcars$mpg, main = "")
```



The default number of bins may be misleading, we can set more bins (maybe after some trial and error). Using the optional parameter `col` (setting the fill color for the histogram bins) helps focusing on the important message - the distribution.

```
hist(mtcars$mpg, col = "gray", breaks = 10, main = "")
```

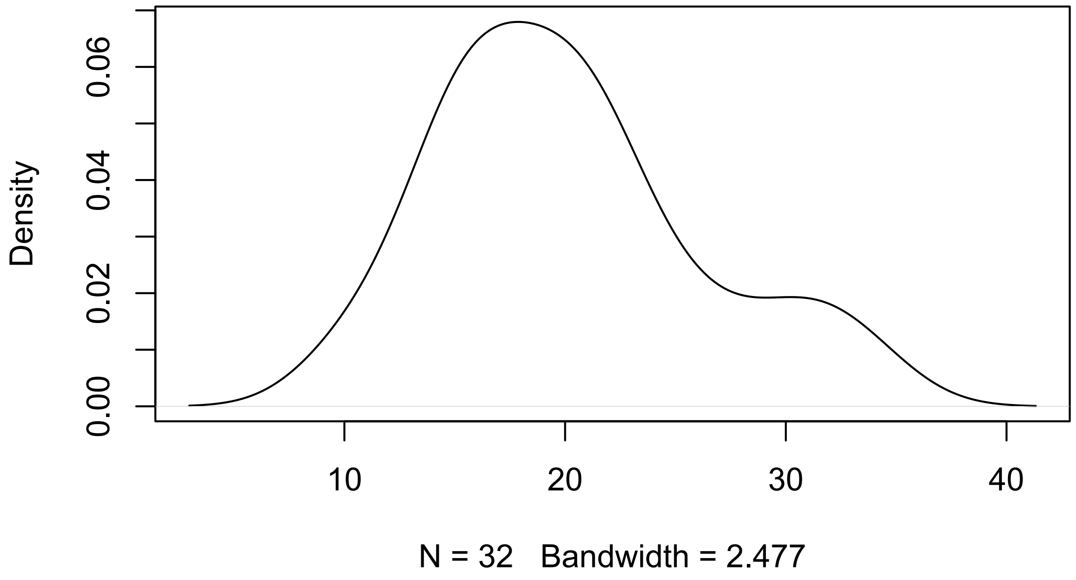


The **kernel density** estimate is a hypothetical continuous distribution generating a univariate series and provides a smooth approximation for the actual distribution. Kernel density estimates are closely related to histograms, but can be endowed with properties such as smoothness or continuity by using a suitable kernel.

Note we must first compute the estimated density with the `density` function.

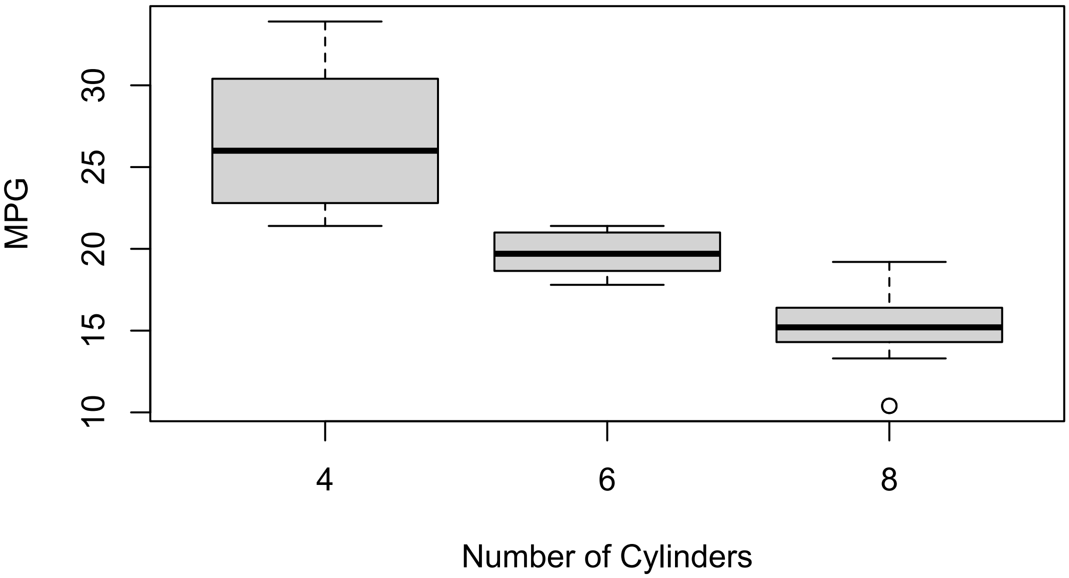
```
# Compute the density data
d <- density(mtcars$mpg)

# Graph the results
plot(d, main = "")
```



Boxplots summarize univariate series in a single plot (including the range of the variable, its quartiles, and its outliers).[10] In future lectures we will dig deeper on summarizing distributions. Here we will only plot the classical boxplot, grouping by the number of cylinders.

```
boxplot(mpg ~ cyl, data = mtcars, ylab = "MPG",
        xlab = "Number of Cylinders")
```

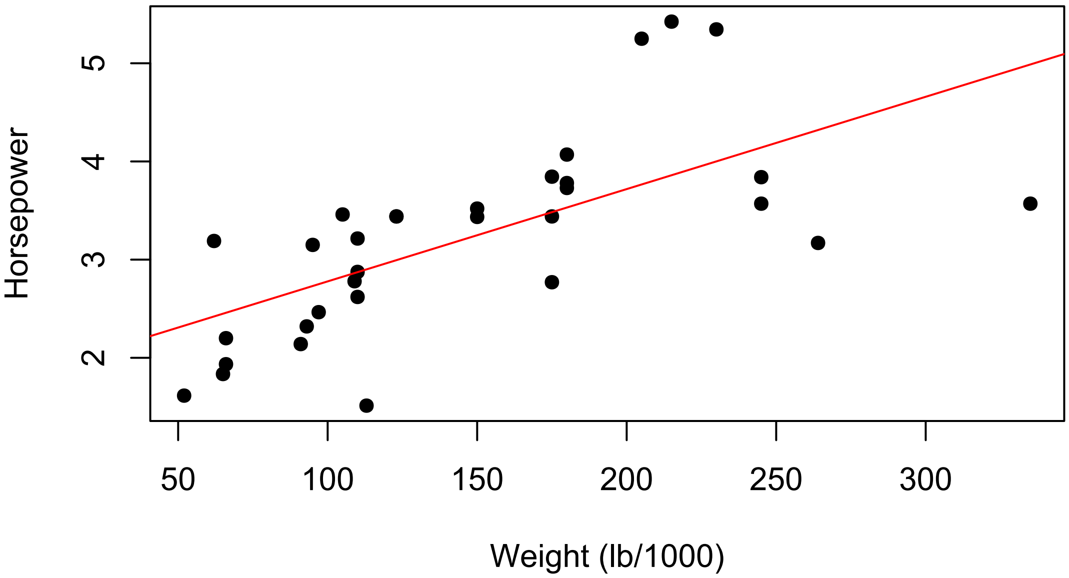


Scatterplots display values for two variables for a set of data, and are essential when looking for relationships between them (e.g. linear correlation). In R the simplest way to plot them is the `plot` function, where we can also add a regression line.

```
# Scatterplot
plot(x = mtcars$hp, y = mtcars$wt, ylab = "Horsepower",
     xlab = "Weight (lb/1000)", pch = 16)

# Linear regression line
abline(lm(mtcars$wt ~ mtcars$hp), col = "red")
```

[10] statmethods.net/graphs/boxplot.html



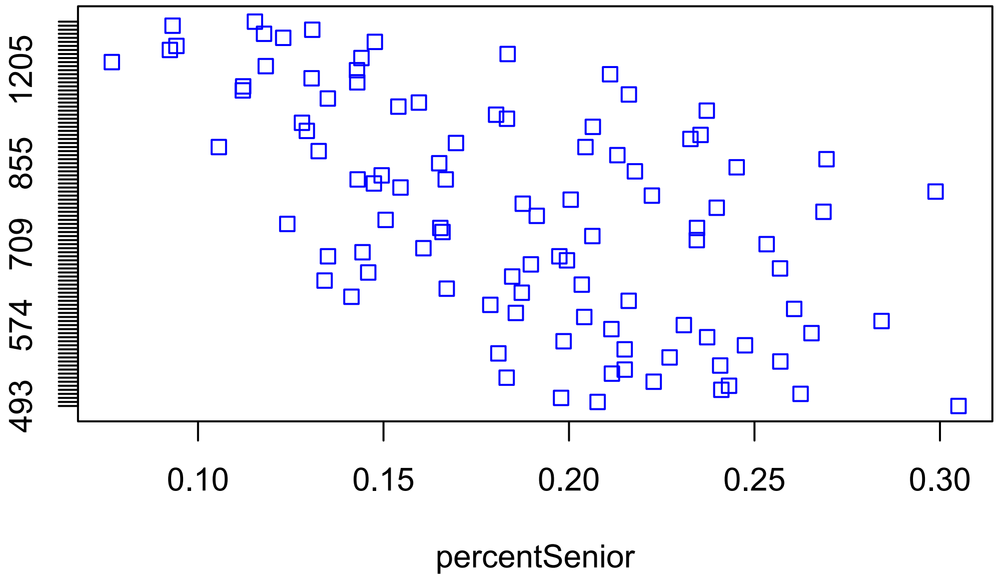
Practice describing BCN census data. Use the Demographic data of census sections in Barcelona we imported earlier and . . .

1. produce stripcharts, histograms and kernel density estimates of a variable of your choice. Be creative: define a new variable combining existing ones, combine colors, explore optional parameters of the functions.

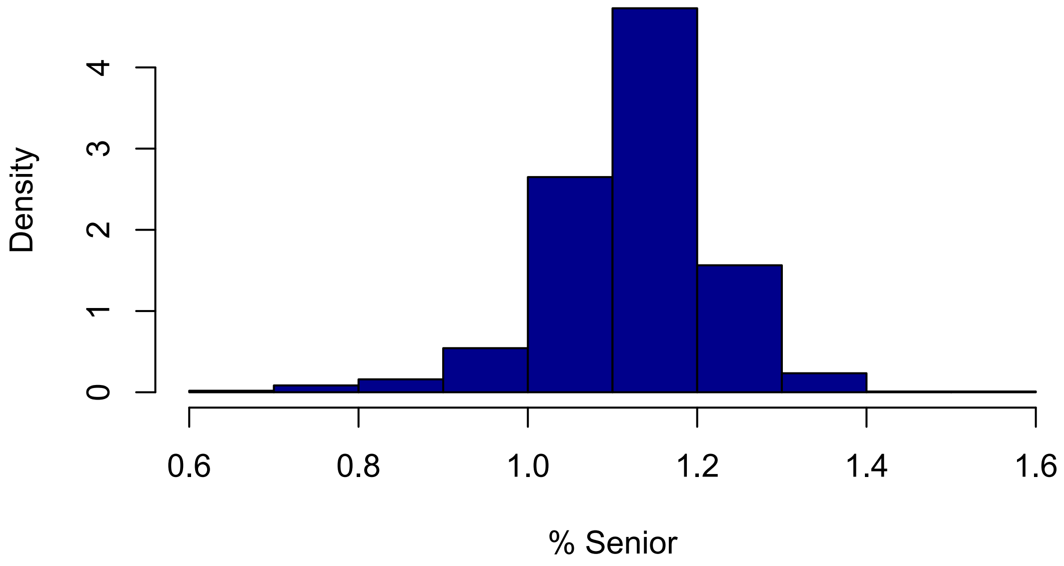
2. scatterplot two variables and add a linear regression line. Try adding a loess regression curve. Choose appropriate point characters and point dimensions (with lots of data points maybe blank-filled smaller dots are most convenient).

3. challenging bit: use the layout function to display all the univariate plots in a single matrix of plots.

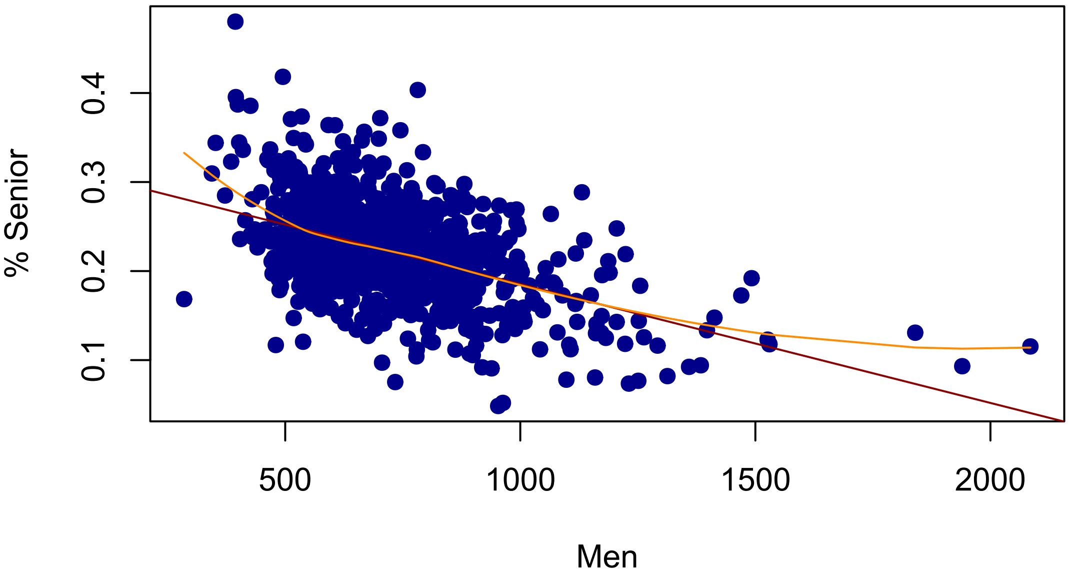
```
#1
censusBCN$women2men_ratio <- censusBCN$Women / censusBCN$Men
stripchart(percentSenior ~ Men, data=censusBCN[1:100,],col='blue')
```



```
hist(censusBCN$women2men_ratio, col='darkblue',main=NULL,xlab='% Senior',freq=F)
```



```
# 2
plot(censusBCN$percentSenior ~censusBCN$Men,
     pch=19,
     cex=1,
     col='darkblue',
     xlab='Men',
     ylab='% Senior')
abline(lm(percentSenior~Men,data=censusBCN),col='darkred')
# add local polynomial regression
x <- censusBCN$Men[order(censusBCN$Men)]
y <- loess(percentSenior~Men,data=censusBCN)$fitted[order(censusBCN$Men)]
lines(x=x, y=y, col='darkorange',pch=19,cex=1)
```

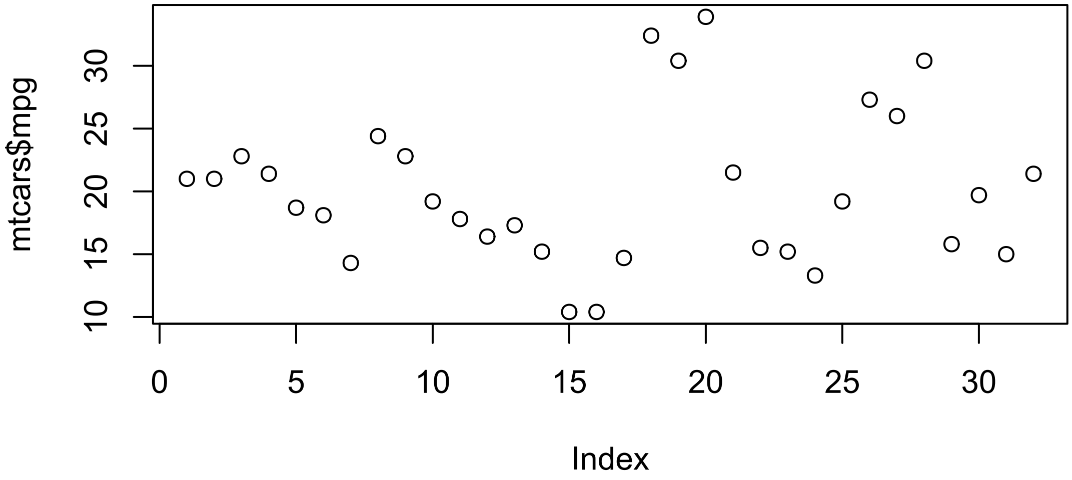


4.1 A full example of a customized plot

Plotting data in R is always easy, but obtaining the format you need almost never is. Some packages will make your life easier, but you need to learn the basics of plot personalization. This is another powerful R feature.

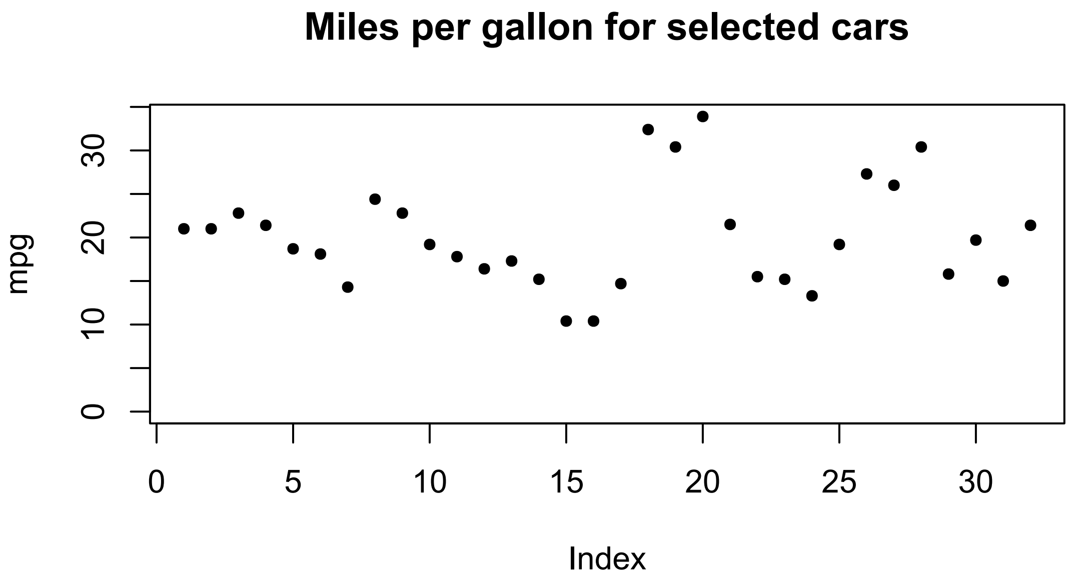
Say we want to plot the miles per gallon of the `mtcars` data set. It is pretty straightforward.

```
plot(mtcars$mpg)
```



But you will agree it is also disappointing - bad axis titles, no main title, no identification of each car etc. But these flaws are also strengths. Plots are objects as any other and all the elements can be personalized and coded. This way, they can be reproduced when data change or if you share your code.

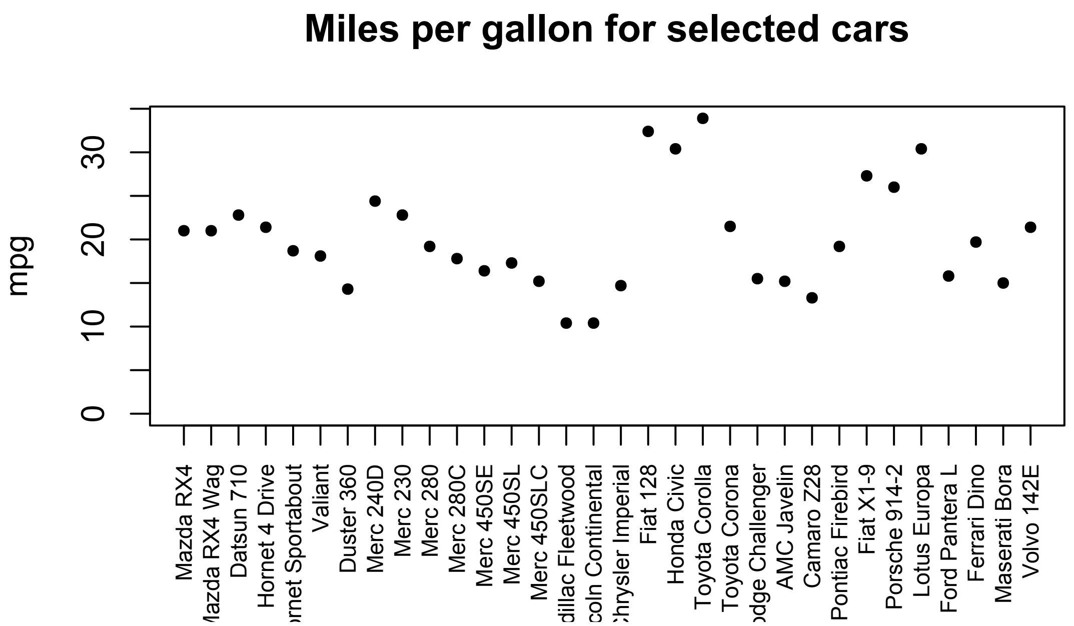
```
plot(mtcars$mpg,
     main = "Miles per gallon for selected cars",
     ylab = "mpg",
     pch = 16,
     cex = 0.8,
     ylim = c(0,max(mtcars$mpg))
    )
```



Now we have fixed some of these issues: `main` and `ylab` improve titles, `pch` and `cex` improve formats, `ylim` lets us set the axis limits. But the X axis remains without the proper car labels. Let us fix that too.

```
plot(mtcars$mpg,
     main = "Miles per gallon for selected cars",
     ylab = "mpg",
     pch = 16,
     cex = 0.8,
     ylim = c(0, max(mtcars$mpg)),
     xaxt = "n",
     xlab = "")

axis(side = 1,
     at = seq_along(mtcars$mpg),
     labels = rownames(mtcars),
     las = 2,
     cex.axis = 0.7)
```



With the axis function we can control the position and content of the axes (here the X axis, or side=1).

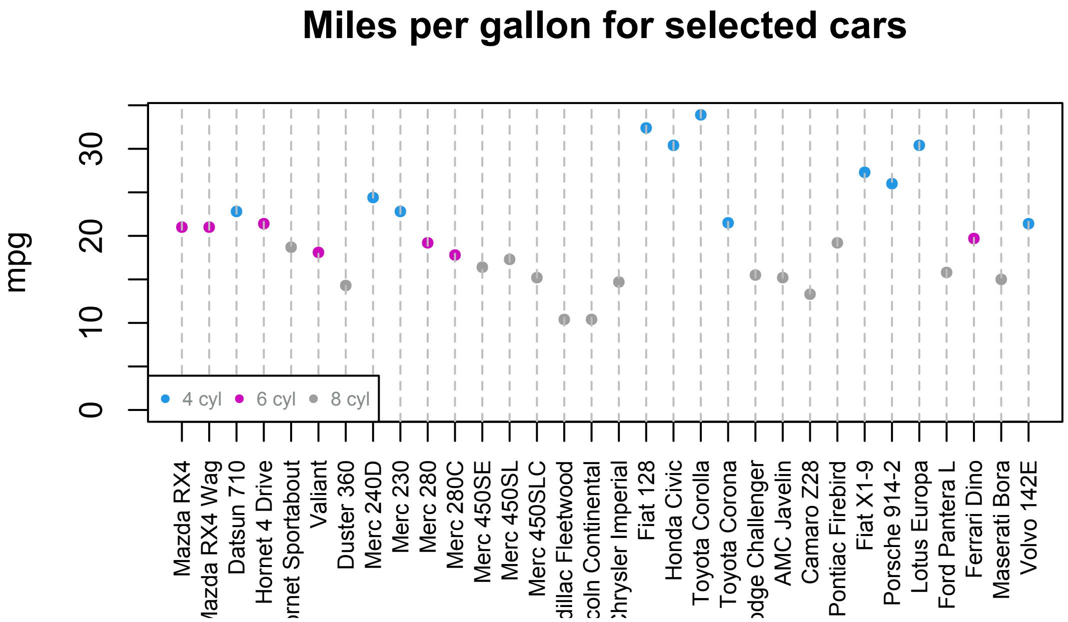
Maybe adding vertical gridlines will help identifying each car, and coloring according to the cylinders may be informative.

```
plot(mtcars$mpg,
     main = "Miles per gallon for selected cars",
     ylab = "mpg",
     pch = 16,
     cex = 0.8,
     ylim = c(0, max(mtcars$mpg)),
     xaxt = "n",
     xlab = "",
     col = mtcars$cyl)

axis(side = 1,
     at = seq_along(mtcars$mpg),
     labels = rownames(mtcars),
     las = 2,
     cex.axis = 0.7)

abline(v = seq_along(mtcars$mpg),
       col = "gray",
       lty = 2)

legend(x = "bottomleft", ncol = 3, cex = 0.6,
       bg = "white",
       legend = c("4 cyl", "6 cyl", "8 cyl"),
       text.col = "azure4",
       col = c(4, 6, 8), pch = 16)
```



Changing fonts is a bit more tricky. Moreover, R by default uses Helvetica font in figures and these fonts are not necessarily available everywhere as it is a commercial font. Hence, some pdf readers might not render correctly the figures. See `extrafont` package for some extra options with fonts.

4.2 Saving plots

By default, a plot in R opens a device in your desktop (or within RStudio). But usually you will need to save it as a high-quality image. R allows different output formats for graphics (pdf, jpg, png). Here we will set an example of pdf output.

```
## Open the device
pdf("mileage.pdf", width = 10, height = 2.5)

## Add the plot
plot(mtcars$mpg)
abline(h = mean(mtcars$mpg),
       col = "lightgray",
       lty = 2)

## Close the device
dev.off()
```

Note that no directory path is specified. By default, the file will be saved at the current working directory, optionally you can set a different output directory.

5 Basic text manipulation

In almost every analysis you need to perform operations on dates and text strings. Here we will take a look at the essential operations on these types of data.

Handling strings in R can sometimes be painful.[11] Several packages exist that ease this pain. Here we look only at R base functions.

The `paste` function is perhaps the most used in R when handling strings. It essentially concatenates strings, but in a generalized way (e.g. you can choose the character separating strings, or it converts non-string objects to characters). By default, it concatenates strings separating them with a blank space.

```
paste("Barcelona", "GSE")
```

```
## [1] "Barcelona GSE"
```

The sep parameter sets a different separator. The `paste0` function is a convenient alternative when you don't want any form of separation, so it is as if you set `sep = ""`.

```
paste("Barcelona", "GSE", sep = "-")
```

```
## [1] "Barcelona-GSE"
```

```
paste0("Barcelona", "GSE")
```

```
## [1] "BarcelonaGSE"
```

Numeric variables are coerced to strings.

```
paste("The Life of", pi)
```

```
## [1] "The Life of 3.14159265358979"
```

You can also operate with vectors.

```
paste("Class of 201", 4:7, sep = "")
```

```
## [1] "Class of 2014" "Class of 2015" "Class of 2016" "Class of 2017"
```

Count number of characters: `nchar` function works both with a single string or with a vector.

```
nchar(c("How", "many", "characters?"))
```

```
## [1]  3  4 11
```

```
nchar("How many characters?")
```

```
## [1] 20
```

[11] A classical reference for handling text is: Sanchez, G. (2013) Handling and Processing Strings in R. Trowchez Editions. Berkeley. gastonsanchez.com/Handling_and_Processing_Strings_in_R.pdf

Convert to lower/upper case with `tolower` and `toupper` functions. Again, they also work on vectors.

```
tolower("Barcelona GSE")

## [1] "barcelona gse"

toupper(c("Barcelona", "GSE"))

## [1] "BARCELONA" "GSE"
```

Obtain and replace substrings with `substr` function.

```
substr("Barcelona GSE", start = 11, stop = 13)

## [1] "GSE"

days <- c("Mond", "Tues", "Wedn")
substr(days, 4, 4) <- "."
days

## [1] "Mon." "Tue." "Wed."
```

Character translation with `chartr`.

```
chartr(old = "4", new = "a", "B4rcelon4 GSE")

## [1] "Barcelona GSE"

chartr(old = "410", new = "aio",
       "B4rcel0n4 Gr4du4te Sch00l of Ec0n0m1cs")

## [1] "Barcelona Graduate School of Economics"
```

Uniquely abbreviate strings with `abbreviate`.

```
abbreviate(c("Asset Pricing", "Corporate Finance",
             "Econometrics"), minlength = 5)

##     Asset Pricing Corporate Finance      Econometrics
##           "AsstP"           "CrprF"           "Ecnmt"
```

Practice character to numeric when importing. The Aging Population data imported earlier has some numerical columns stored as character. Use basic string manipulations to convert them back into numerical.[12]

```
agingPopulation$Total.Population <-
  as.numeric(gsub(',','',agingPopulation$Total.Population))
agingPopulation$PercentSeniors <-
  as.numeric(gsub('%','',agingPopulation$PercentSeniors))
```

[12] Hint: define new columns for trial and error.

6 Dates and times in R

Dates are represented as the number of days since 1970-01-01, with negative values for earlier dates. But R outputs them with the familiar formats (e.g. MMDDYYY). Converting to/from dates and operating with them requires some familiarity with the main R date formats and functions.[13] There are packages like `lubridate` that facilitate handling of dates and time.

System date and time are useful for many purposes (e.g. computing execution times, saving files with dynamical names).

```
# System date with default format
Sys.time()

## [1] "2020-09-10 13:15:40 CEST"

# Time with HH:MM:SS format
format(Sys.time(), "%H:%M:%S")

## [1] "13:15:40"

# Date with YYYYMMDD format
format(Sys.time(), "%Y-%m-%d")

## [1] "2020-09-10"

# using system time for measuring difference
x <- Sys.time()
y <- Sys.time()
y - x

## Time difference of 0.008023977 secs

# there is a specific function for that
system.time(
    for(i in 1:100) mad(runif(1000))
)

##    user  system elapsed
##   0.044   0.005   0.048
```

Output of `system.time` might be confusing. **User CPU time** gives the CPU time spent by the current R session, while **system CPU time** gives the CPU time spent by the operating system on behalf of the R session. The operating system might do additional other operations like opening files, doing input or output, starting other processes, and looking at the system clock, operations that involve resources that many processes must share. **Elapsed time** is the sum of the two.

Converting strings to date/time objects is the name of the game when importing data files.

[13] en.wikibooks.org/wiki/R_Programming/Times_and_Dates

Being familiar with date conversion and formatting is also crucial when reporting results. Some examples.

```
# Input date format
x <- as.Date("20140912", format = "%Y%m%d")
x

## [1] "2014-09-12"

class(x)

## [1] "Date"

typeof(x)

## [1] "double"

# Input time and date
strptime("09/12/11 17.30.00", format = "%m/%d/%y %H.%M.%S")

## [1] "2011-09-12 17:30:00 CEST"

# convert to string
as.character(Sys.time())

## [1] "2020-09-10 13:15:40"
```

Extracting information from dates.

```
# Name of weekday
weekdays(Sys.time())

## [1] "Thursday"

# Name of month
months(Sys.time())

## [1] "September"

# Number of days since beginning of epoch
julian(Sys.time())

## Time difference of 18515.47 days
```

Julian Day Number (JDN)[14] is the number of days since noon UTC on the first day of 4317 BC.

Generating sequences of dates

```
seq(from = as.Date("2014-09-12"),
    to = as.Date("2014-09-14"),
    by = "day")
```

[14] en.wikipedia.org/wiki/Julian_day

```
## [1] "2014-09-12" "2014-09-13" "2014-09-14"
```

```r
# All days between two dates
seq(from = as.Date("2014-09-12"),
    to = as.Date("2014-11-12"),
    by = "month")
```

```
## [1] "2014-09-12" "2014-10-12" "2014-11-12"
```

```r
# All months between two dates
seq(from = as.Date("2014-09-12"),
    to = as.Date("2014-09-16"),
    length.out = 3)
```

```
## [1] "2014-09-12" "2014-09-14" "2014-09-16"
```

```r
# Every other day between two dates
# Next 3 days
seq.Date(Sys.Date(), length = 3, by = "1 days")
```

```
## [1] "2020-09-10" "2020-09-11" "2020-09-12"
```

```r
# Next 3 months
seq.Date(Sys.Date(), length = 3, by = "1 months")
```

```
## [1] "2020-09-10" "2020-10-10" "2020-11-10"
```

Operations with dates.

```r
# Number of days since a given date
julian(Sys.time()) - julian(as.Date("2014-01-01"))
```

```
## Time difference of 2444.469 days
```

```r
# Adding days
as.Date("2014-09-12") + 30
```

```
## [1] "2014-10-12"
```

```r
# Adding months
seq.Date(Sys.Date(), length = 2, by = "3 months")[2]
```

```
## [1] "2020-12-10"
```

Practice proper formatting of dates in imported data. Create a new column in the BCN census data containing the day after the first column date. You must use the `as.Date` function.

7 Time Series in R

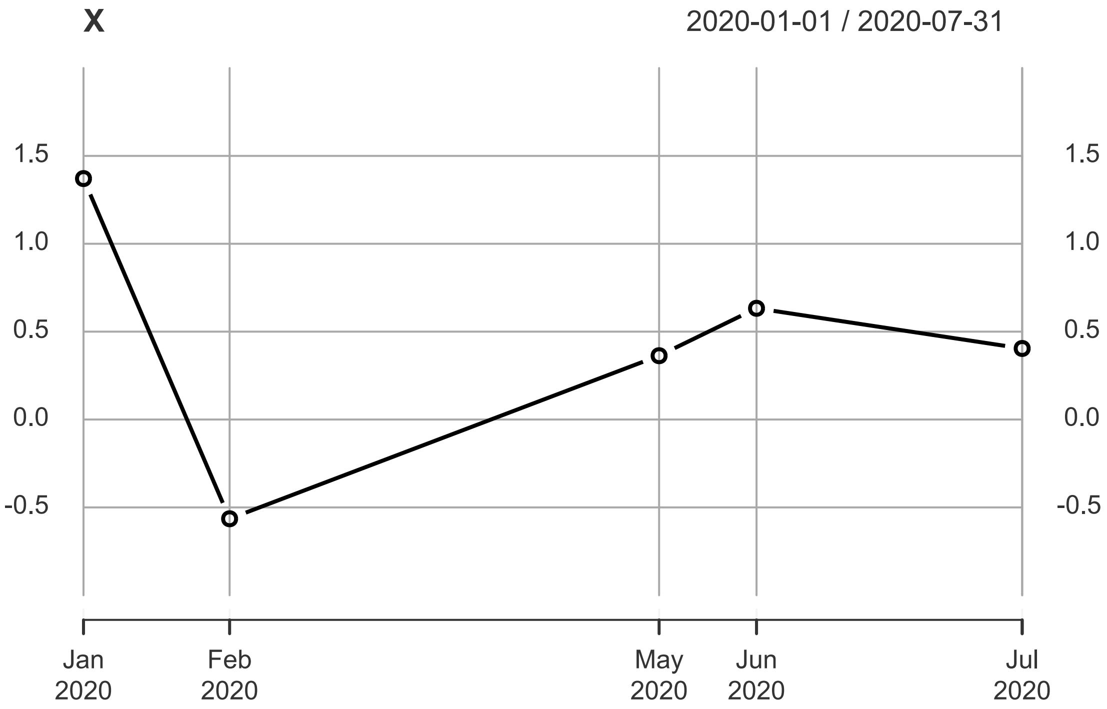
Manipulating Time Series in R with what we have learnt so far is possible. However, the powerful package **eXtensible Time Series** `xts` can make our lives a lot easier. **`xts`** objects are simple, we just have to think of them as matrices of observations combined with an index of corresponding dates and times. To create an `xts` object, we just need to have a vector/matrix of observations and an index vector (date format) with the same length:

```
# Load xts
if (!require(xts)) install.packages('xts') else library(xts)

# Generate normally distributed data
set.seed(42)
data <- data.frame(x=rnorm(5))

# Create dates as a Date class object (they can be irregularly spaced!)
dates <- as.Date(c("2020-01-01","2020-02-03",
                   "2020-05-10","2020-06-01",
                   "2020-07-31"))
# Create xts object
X <- xts(x = data, order.by = dates)

# Plot xts object
plot(X, type='b',ylim=c(-1,2))
```



```
# Given an xts, it is also easy to extract the core data and the index
# in case you want to deal with them separately
class(coredata(X))
```

```
## [1] "matrix" "array"
```

```
class(index(X))
```

```
## [1] "Date"
```

One major difference between xts and most other time series objects in R is the ability to use any one of various classes that are used to represent time. Whether POSIXct, Date, or some other class, xts will convert this into an internal form to make subsetting as natural to the user as possible.

```
# Create dates
dates <- as.Date("2016-01-01") + 0:4

# Create ts_a with a Date object
ts_a <- xts(x = 1:5, order.by = dates)

# Create ts_b with POSIXct object
ts_b <- xts(x = 1:5, order.by = as.POSIXct(dates))

# Extract the rows of ts_a using the index of ts_b
ts_a[index(ts_a)]
```

```
##            [,1]
## 2016-01-01    1
## 2016-01-02    2
## 2016-01-03    3
## 2016-01-04    4
## 2016-01-05    5
```

```
# Extract the rows of ts_b using the index of ts_a
ts_a[index(ts_b)]
```

```
##            [,1]
## 2016-01-01    1
## 2016-01-02    2
## 2016-01-03    3
## 2016-01-04    4
## 2016-01-05    5
```

7.1 Importing data to xts

```
tmp_file <- "http://s3.amazonaws.com/assets.datacamp.com/production/course_1127/datasets

# Create dat by reading tmp_file using read.csv
dat <- read.csv(tmp_file)

# Convert dat into xts
dat_xts <- xts(dat, order.by = as.Date(rownames(dat), "%m/%d/%Y"))

# Alternative: use read.zoo and convert to xts
dat_zoo <- as.xts(read.zoo(tmp_file, index.column=0, sep=',', format='%d/%m/%Y'))
```

7.2 Exporting xts

If you are working exclusively in R, the most convenient way to export/import xts objects is by use of the `saveRDS()` and `readRDS()` functions. Of course, you could use `save()` but this is different since it saves the object with the name in the current environment, whereas `saveRDS()` just saves a representation of the object.

```
f <- 'data/dat_zoo.rds'
saveRDS(object = dat_zoo, file = f)
class(readRDS(f))
```

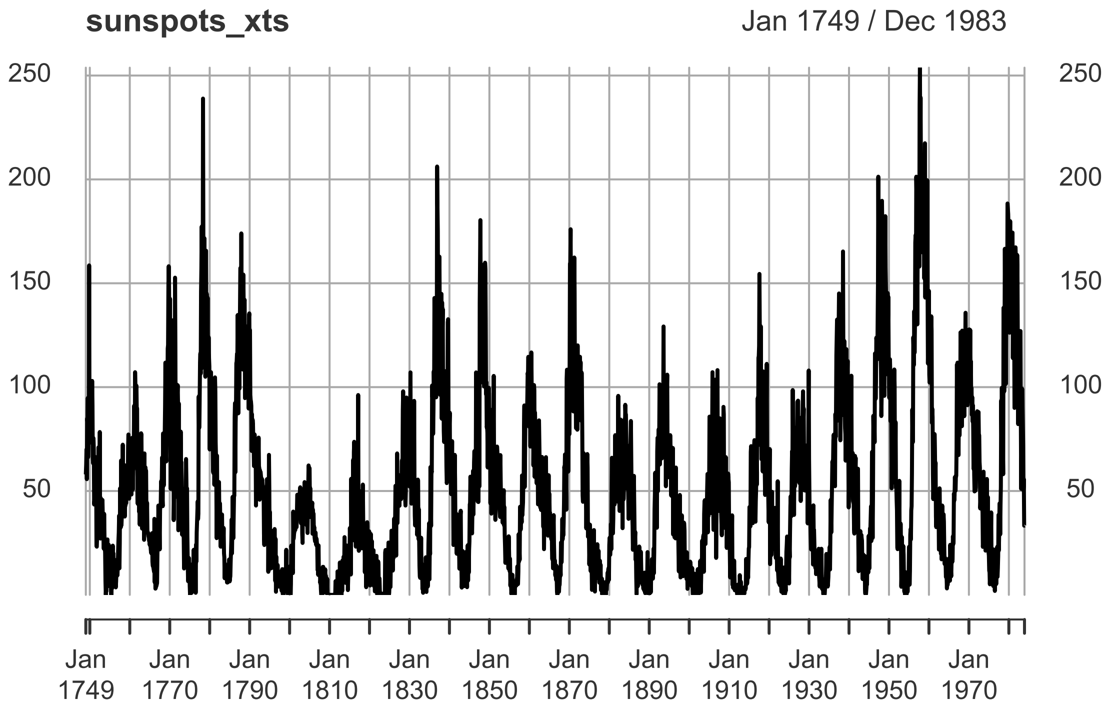
```
## [1] "xts" "zoo"
```

But sometimes you find yourself needing to share results with others, often expecting data to be consumed by processes unaware of R and xts. In that case, one of the best ways to write an xts object from R is to use the zoo function `write.zoo`. In this brief exercise we will convert sunspots to xts and save it as sunspots_xts.

```
# Get pre-loaded data and inspect the class
data(sunspots)
class(sunspots)
```

```
## [1] "ts"
```

```
# Convert sunspots to xts using as.xts().
sunspots_xts <- as.xts(sunspots)
plot(sunspots_xts)
```



```
# Get the temporary file name
tmp <- tempfile()

# Write the xts object using zoo to tmp
write.zoo(sunspots_xts, sep = ",", file = tmp)

# Read the tmp file. FUN = as.yearmon converts strings such as Jan 1749 into a proper ti
sun <- read.zoo(tmp, sep = ",", FUN = as.yearmon)

# Convert sun into xts. Save this as sun_xts
sun_xts <- as.xts(sun)
```

7.3 Extracting data

Extracting data is intuitive with xts objects. You can use the same indexing techniques as with matrices, and more. For instance, consider the following examples:

```
# Extract first two observations
X[1:2]
```

```
##                     x
## 2020-01-01  1.3709584
## 2020-02-03 -0.5646982
```

```
# Extract date '2020-06-01'
X['2020-06-01']
```

```
##                     x
## 2020-06-01 0.6328626
```

```r
# Extract year '2020'
X['2020']
```

```
##                      x
## 2020-01-01  1.3709584
## 2020-02-03 -0.5646982
## 2020-05-10  0.3631284
## 2020-06-01  0.6328626
## 2020-07-31  0.4042683
```

```r
# Extract only January and February data:
X['202001/202002']
```

```
##                      x
## 2020-01-01  1.3709584
## 2020-02-03 -0.5646982
```

```r
# Extract data from a given vector of dates
dates <- c('2020-05-10','2020-07-31')
X[dates]
```

```
##                     x
## 2020-05-10 0.3631284
## 2020-07-31 0.4042683
```

```r
# Extract first and last two months
first(X, "2 months")
```

```
##                      x
## 2020-01-01  1.3709584
## 2020-02-03 -0.5646982
```

```r
last(X, "2 months")
```

```
##                     x
## 2020-06-01 0.6328626
## 2020-07-31 0.4042683
```

Also it is possible to merge xts objects by date, combine rows/columns as with a data.frame structure, take differences in the data, lags, and many other convenient functions for time series analysis.

8 Data Wrangling with dplyr

A package which is definitely worth exploring for data manipulation is `dplyr`. To illustrate, we use the `flights` dataset from the `nycflights13` package.

```
head(flights,5)
```

```
## # A tibble: 5 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1  2013     1     1      517            515         2      830            819
## 2  2013     1     1      533            529         4      850            830
## 3  2013     1     1      542            540         2      923            850
## 4  2013     1     1      544            545        -1     1004           1022
## 5  2013     1     1      554            600        -6      812            837
## # ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

The `dplyr` package has a `filter()` function (note that when loading the package, the `filter()` function from the `stats` package is masked) which we can use as follows. Say we would like to get only the flights that happened in November or December, which arrived more than two hours late, but did not leave late. Then,

```
filter(flights,
       month %in% c(11,12),
       dep_delay <= 0,
       arr_delay > 120)
```

```
## # A tibble: 1 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1  2013    11     1      658            700        -2     1329           1015
## # ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Another useful command is the pipe operator `%>%`. It allows us to perform different manipulations sequentially, taking the output from the last operation as the input for the next one:

```
flights %>%
  filter(month %in% c(11,12),
         dep_delay <= 0,
         arr_delay > 120) %>%
  select(year,month,day,dep_delay,arr_delay)
```

```
## # A tibble: 1 x 5
##    year month   day dep_delay arr_delay
##   <int> <int> <int>     <dbl>     <dbl>
## 1  2013    11     1        -2       194
```

A lot more can be done with this package. For instance, check out dplyr's cheat sheet [15]

[15] `dplyr` cheat sheet: https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf