COMP6235 -- Foundations of Data Science

This is a tutorial on some functions in R for COMP6235 Foundations of Data Science -- part I

Getting started

Download R and install it on your laptop, e.g. from the R project. Alternatively, you can run R on your virtual machine, or use the jupyter notebook set up last week.

Change the the working directory to a folder in which you want to store your R files. R gives you and option to save the current working space when you exit (and will automatically reload it when you start it again), so --to avoid confusion-- it is advisable to generate folders for each individual R project. You can then start R by typing the command line command "R". Once in R, you can quit using the command "q()".
The instructions below have been developed for a linux system in which you use the command line version of R (but should also work fine if you use R studio, or alternatively you could use your jupyter notebook as introduced in the tutorial last week).

You can change the working directory in which R operates using the command "setwd ()" or alternatively use "getwd()" to find the current working directory.

Libraries

Often you may want to extend the basic version of R by using additional libraries/packages. You can easily install them using the install.packages command, i.e. by typing

```
> install.packages ("tidyverse")
```

You would install the so-called tidyverse package (which contains a collection of libraries such as ggplot2). If there are problems with the installation make sure that your firewall does not block https://cloud.r-project.org/.
After installing a package, you need to load the library before you can use it,

i.e.

```
> library ("tidyverse")
```

to load the tidyverse package.

Getting Help

R has an inbuilt help function, which you can use to check the documentation of R commands and exploring options you can use with these commands. The help function can be invoked either by using

```
> ?q
```

or

```
> help (q)
```

which will give show you the documentation page of the quit command.

Data Structures

R is pretty much a full programming language, designed with statistical calculations in mind (but can also be used for many other purposes). Data types provided are more or less standard what you will find in other programming languages as well.

- integer (numbers without decimals)
- double (numbers with decimals)
- character (text data)
- logical (logical)

There are also two special values -- "Null" (indicates absence of any value) and "NA" (indicates missing value)

Vectors

The most fundamental data type in R are vectors, which store ordered sets of elements of the same type. We can easily create vectors with the combine "c ()" function.

Examples

```
> subject_name <- c("Jon Doe", "Jane Doe", "Markus")
> temperature <- c(37.0,37.4,40.0)
> flu_status <- c(FALSE,FALSE,TRUE)
```

We can access elements of vectors in various ways: temperature [1], temperature[-2], temperature [c(FALSE, FALSE, TRUE)]
Note, that the assignment operator (which sometimes is "=" or ":=" in other programming languages) in R is " <-"

Factors

Are a special type of vector used for nominal (or categorical) data, i.e. data for which there is not necessarily an ordering relationship and which can only assume a set of discrete values. Examples:

```
> gender <- factor (c("MALE", "MALE", "FEMALE"))
```

i.e. factors are created by applying "factor ()" to a character vector.
To show the content of a data structure in R we can just type the name, e.g.

```
> gender
[1] MALE MALE FEMALE
Levels: FEMALE MALE
```

The "levels" variable displays information about possible categories the factor could take.
The command can also be used to define more levels than present in the actual data, e.g. if we wanted to define symptoms for each patient:

```
> symptoms <- factor (c("SEVERE", "MILD", "MODERATE"), levels =
c("MILD", "MODERATE", "SEVERE"), ordered=TRUE)
```

generates a factor symptoms in which "<" indicates an ordering, i.e.

```
> symptoms [1] SEVERE MILD MODERATE
Levels: MILD < MODERATE < SEVERE
```

For the data we could easily test of a patients symptoms are greater than mild:

```
> symptoms > "MILD"
TRUE, FALSE, TRUE
```

Lists

As vectors, a collection of elements, but not every element needs to be of the same type. Entries in lists can be addressed using a name for each field (or by number as in vectors). Example:

```
> subject1 <- list(fullname = "Markus", temperature=40.0,
symptoms=symptoms[1])
> subject1
$fullname
[1] "Markus"

$temperature
40.0

$symptoms
"SEVERE"
Levels: MILD < MODERATE < SEVERE
```

We can also access fields in a list via the "$" separator:

```
> subject1$temperature
40.0
```

We could even obtain several items in a list by passing a vector of its fields:

```
> subject1 [c("fullname", "temperature")]

$fullname
"Markus"
$temperature
40.0
```

We could now construct entire data sets by building lists and lists of lists, etc. However, R provides an inbuilt function to do this more easily: data frames.

Data frames

Can be understood as a list of vectors or factors each of which have the same number of elements. It can be generated with the "data.frame ()" function, e.g.

```
> pt_data <- data.frame (subject_name, temperature, flu_status, gender,
symptoms, stringsAsFactors=FALSE)
```

Note that we need to specify "stringsAsFactors=FALSE", otherwise R will automatically convert every character vector into a factor. When displaying a data frame it is shown in matrix format. As before, we can refer to component vectors with the "$" operator, e.g.

```
> pt_data$temperature
```

will reference the vector temperature in the data frame. Again, we can also access several vectors of a data frame, e.g.

```
> pt_data [c("temperature", "subject_name")].
```

If we don't want to type the name of "pt_data" in front of the record we are addressing each time, we can also use the R-command "append" to make R remember that we refer to pt_data. In this case we'd first type:

```
> attach(pt_data)
```

and could then directly reference the fields temperature, fullname, etc., i.e.: >temperature

now outputs contents of pt_data$temperature without first specifying pt_data.
Individual entries can also be referenced by specifying rows and columns, e.g.

```
> pt_data [2, 1]
```

or combinations thereof

```
>pt_data [c(1, 3), c(2, 4)]
```

will display data from the 1st and 3rd row and 2nd and 4th column. All rows or columns can be extracted by leaving the other entry empty, i.e.

```
>pt_data [, 1]
```

will extract all rows from the first column.

We can add records to data frames using the function rbind (). For example,

let's construct an entry for a new patient:
```
> new <- c("Mike", 40.5, TRUE, "MALE", "SEVERE")
```

to add it to the data frame pt_data:

```
> pt_data <- rbind (pt_data, new)
```

What about introducing a new field into pt_data, e.g. marital status? This can simply be done by defining a new vector in pt_data
```
> pt_data$married <- c("TRUE", "TRUE", "FALSE", "FALSE")
```

Matrices

R also provides another data structure to store values in tabular form. Entries can be of any type, but are most often used for numerical data. Matrices can be generated with the "matrix ()" function with a parameter specifying the number of rows ("nrow") or the number of columns ("ncol"). E.g.

```
> m <- matrix ( c(1, 2, 3, 4), nrow = 2)
```

generates a matrix with two rows, i.e. it divides the input vector c(1, 2, 3, 4) into two rows.

Exercises for part I.

Managing Data

Saving, loading, and removing data structures

Data can be saved using the "save ()" function. R-files typically have an ".RData" file extension, i.e. if we want to save the data structures x,y, and z into a file we'd use the command

```
> save (x, y, z, file = "mydata.RData")
```

Analogously, use "load ()" to retrieve data from a file (but pay attention, it might overwrite existing data frames in your workspace)

```
> load ("mydata.RData")
```

will automatically generate the data structures x, y, and z. There is also a command "save.image ()" which will save your entire workspace to a file call ".RData". Upon startup, R will look for this file and automatically load it when you start R again.
When you handle very large data structures, you might sometimes want to remove some of them from memory. This can be done using the "rm ()" command, e.g.

```
> rm (m, subject1)
```

Will remove the objects m and subject1 from memory. To clear the entire workspace use "rm (list=ls())".

Handling CSV files

R has functions for reading and writing to CSV files. For instance, to read a file use:

```
> mydata <- read.csv ("mydata.csv", stringsAsFactors=FALSE)
```

Note that without path specification R attempts to read the file from the current working directory (the directory from which you started R). By default, R will also assume that the CSV file has a header line to assign names to the various vectors. If you the file does not have a header, use specify "header = FALSE" in the read.csv function. Then, features will be named "V1", "V2", etc.
read.csv is a special case of the more general "read.table ()" function which can read data in other formats. Try to use the help function to explore further ("?read.table").
To write data into a CSV file, simply use "write.csv ()", e.g.:

```
>write.csv(mydata, file = "mydata.csv", row.names = FALSE)
```

(the option "row.names = FALSE" overwrites R's standard mode in which it also adds row names to the file).

More general than read.csv () is the read.table () command which reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file. Read.table () has also options to change the separator of data entries, explore this with the help function if needed.

Exploring data

In this section I'll briefly mention some R function that can be used to explore data. As an example data set I'll use the file usedcars.csv . Let's first read the file into memory:

```
> usedcars <- read.csv ("usedcars.csv", stringsAsFactors = FALSE)
```

To explore the structure of the data, we can use the "str ()" function which will give us some information about various fields/vectors and data types in the data set.

```
> usedcars <- read.csv ("usedcars.csv", stringsAsFactors = FALSE)
> str (usedcars)
'data.frame':150 obs. of 6 variables:
$ year : int 2011 2011 2011 2011 2012 2010 2011 2010 2011 2010 ...
$ model : chr "SEL" "SEL" "SEL" "SEL" ...
$ price : int 21992 20995 19995 17809 17500 17495 17000 16995 16995
16995 ...
$ mileage : int 7413 10926 7351 11613 8367 25125 27393 21026 32655
36116 ...
$ color : chr "Yellow" "Gray" "Silver" "Gray" ...
$ transmission: chr "AUTO" "AUTO" "AUTO" "AUTO" ...
```

We can gather the following:
(1) 150 obs -> the data set contains 150 records (one would often say "n=150", i.e. we have 150 observations).
(2) There are 6 features in the data set (year, model, mileage, color, transmission)
(3) We see of which data type each feature is.
(4) The following entries are the first 4 data entries for each feature.

Exploring numeric variables

We can use the summary command to get a quick overview of some summary stats, e.g.:

```
> summary (usedcars$year)
```

```
Min. 1st Qu. Median Mean 3rd Qu. Max.
2000 2008 2009 2009 2010 2012
```

which gives some quick information about min/max/mean/median values and interquartile ranges (see lectures on this topic for more information LEC???). The information provided can also be explored with separate commands, e.g.

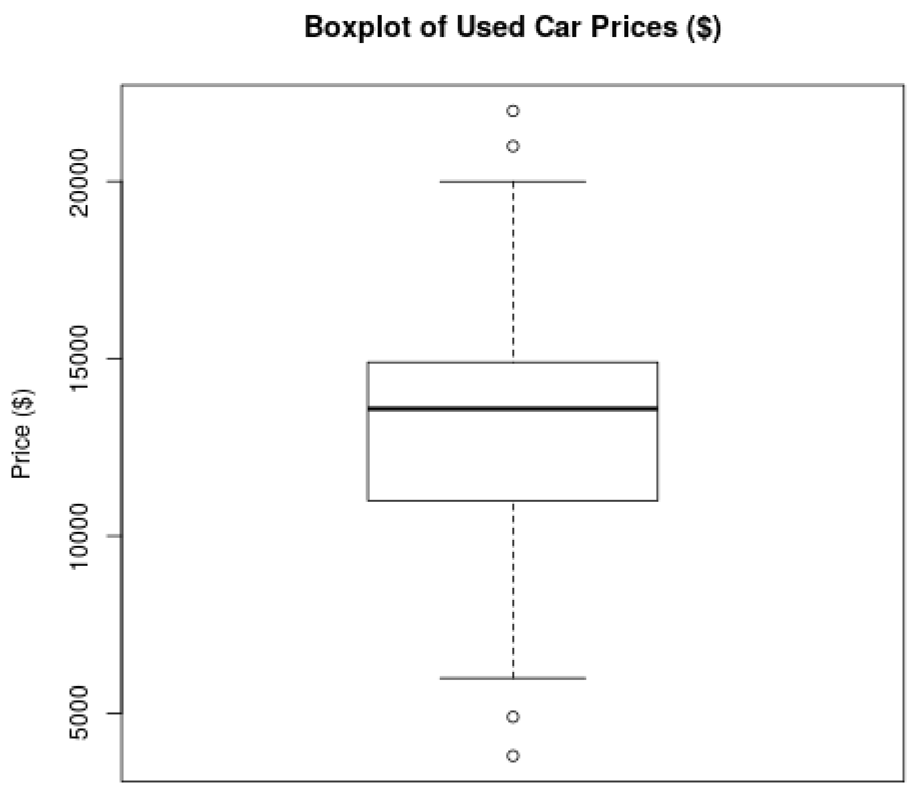
```
> mean (usedcars$year)
> median (usedcars$year)
```

Similarly, functions like "min (), max (), range (), IQR ()" etc. can be used. Explore some of them yourself.

Visualizing numeric variables

A typical way of visualizing the distribution of a numerical variable is a boxplot which gives information about the distribution of this variable in the form of indicating Q1, Q2 (the median), Q3 (for the drawn box in the middle) and the "whiskers" to either indicate min/max or 1.5 times IQR below Q1/above Q3. Values outside of this range are considered outliers and drawn as dots. For example:

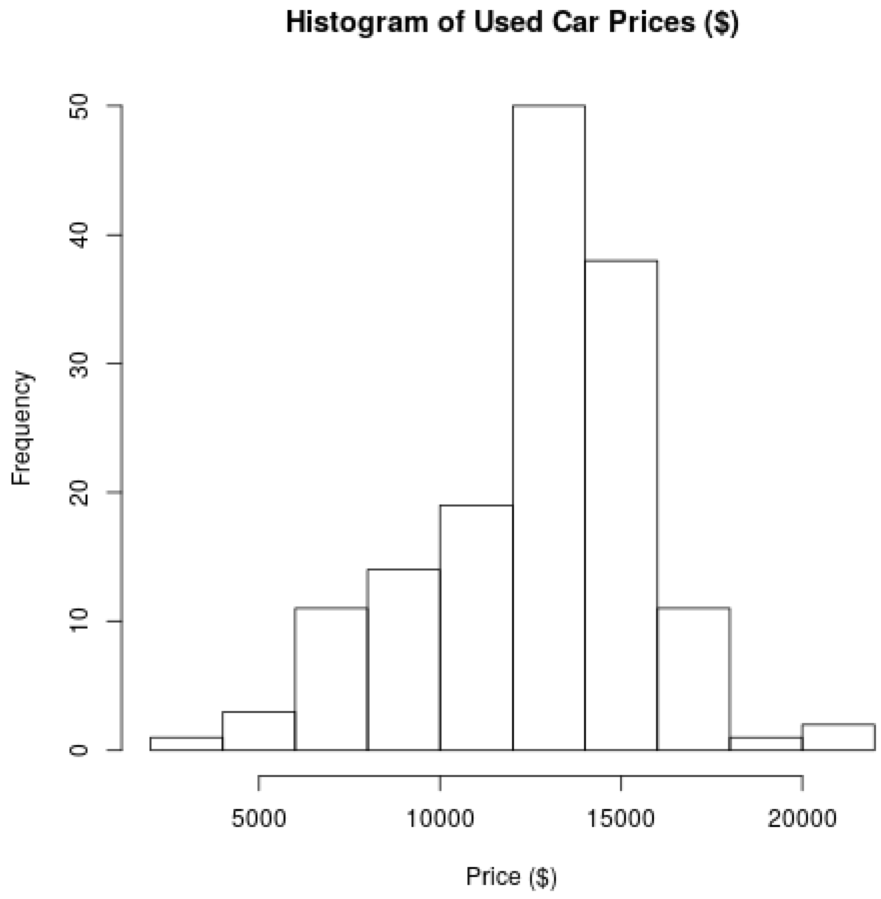
```
> boxplot(usedcars$price, mean="Boxplot of Used Car Prices", ylab="Price
($)")
```



Explore "?boxplot" to see how you can further customize the appearance of the plot.

Another way of visualizing data is via histograms -- see LEC???. In R we can generate histograms via the "hist ()" function, e.g.

```
> hist (usedcars$price, main="Histogram of Used Car Prices", xlab="Price ($)")
```



R also provides a number of functions to calculate the spread of distributions, e.g. "var (), sd (), or IQR ()".

Generating plots in pdf/png format

You can also use R to produce figures in pdf (or other formats) for reports/papers etc. The way to do this is to first set a graphics device, e.g.

```
> pdf ("boxpot.pdf")
```

where you set the output for the next plot command. You then simply repeat the plot command

```
> boxplot(usedcars$price, mean="Boxplot of Used Car Prices", ylab="Price ($)")
```

R will have generated a file "boxpot.pdf" but may not yet have written the contents of this file (which happens when buffers are flushed). To enforce writing contents to the file, you can use the command "dev.off ()". Graphics devices are available for a number of other formats, e.g. "png (filename)", etc. Explore how to customize them yourself.

IMPORTANT: in reports/papers etc. avoid inserting screenshots. When you use screenshots your files become huge, graphics don't scale very well (so always use vector graphics whenever possible) and the output does not look very professional.