Part1-- 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.

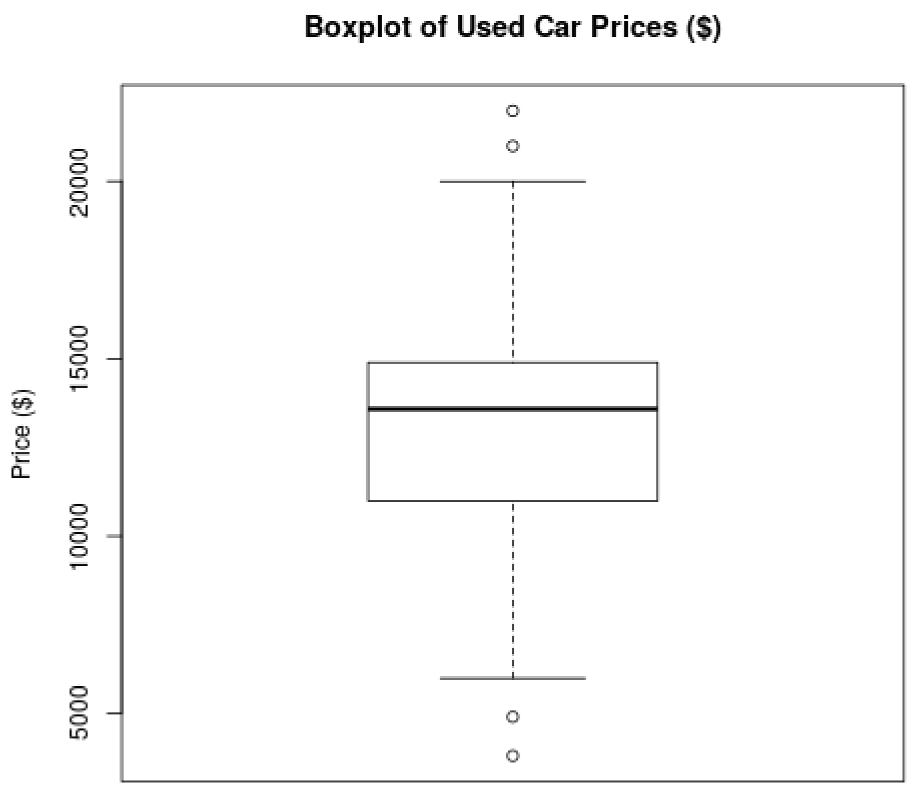
```
> mean (usedcars$year)
> median (usedcars$year)
```

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:

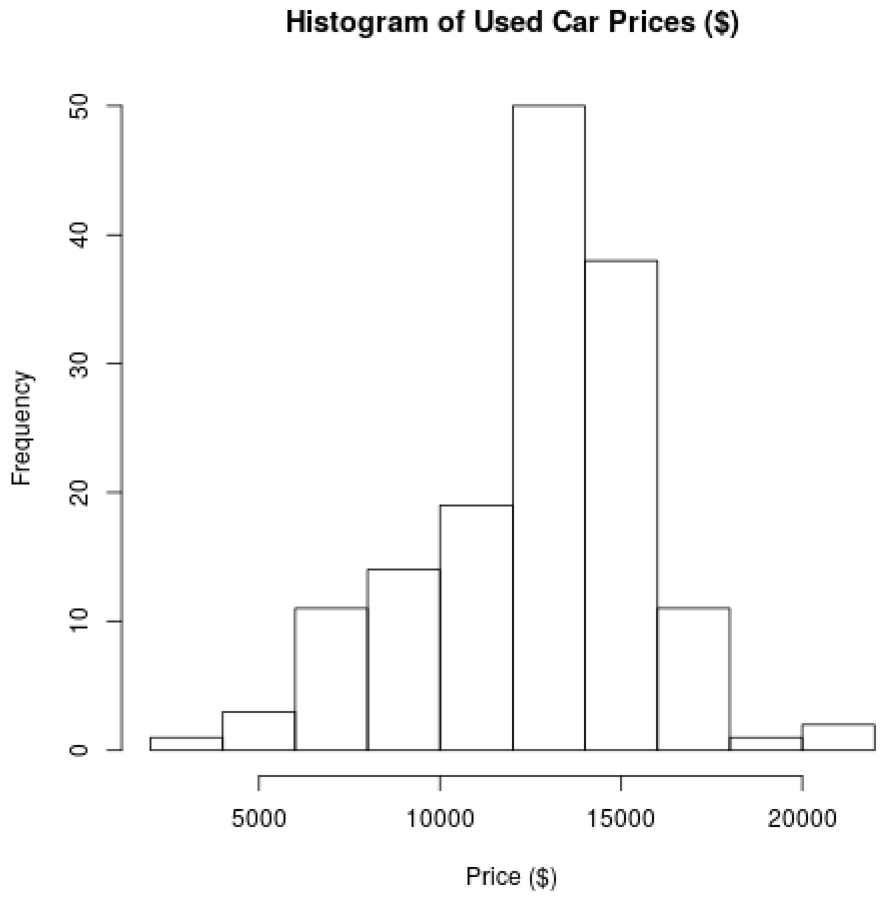
```
> boxplot(usedcars$price, mean="Boxplot of Used Car Prices", ylab="Price
($)")
```



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.

Part2 -- Foundations of Data Science

This is a tutorial on some functions in R for COMP6235 Foundations of Data Science -- part II. This follows on from part I of the tutorial.

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 my lecture for more information). The information provided can also be explored with separate commands, e.g.

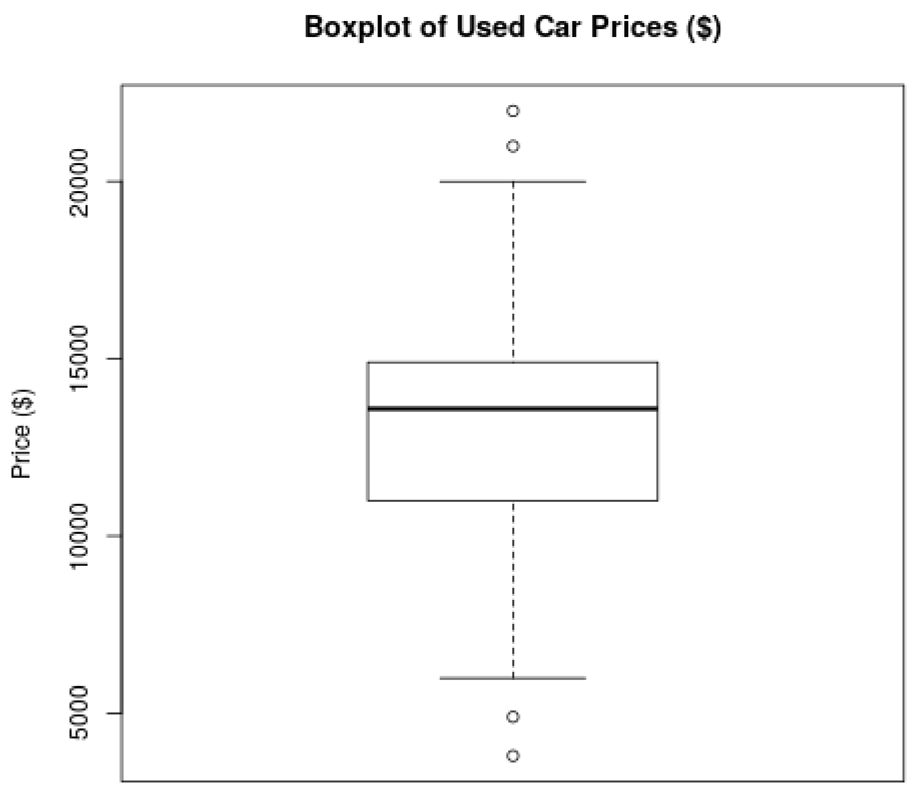
```
> mean (usedcars$year)
> median (usedcars$year)
```

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:

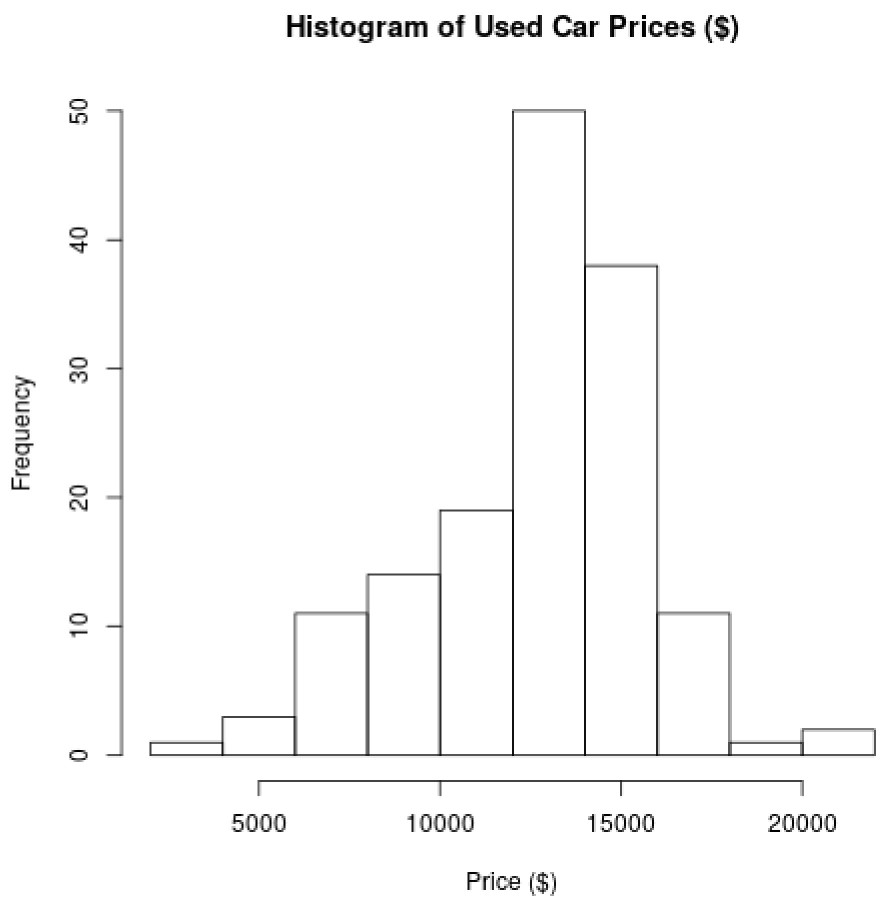
```
> boxplot(usedcars$price, mean="Boxplot of Used Car Prices", ylab="Price
($)")
```



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

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

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

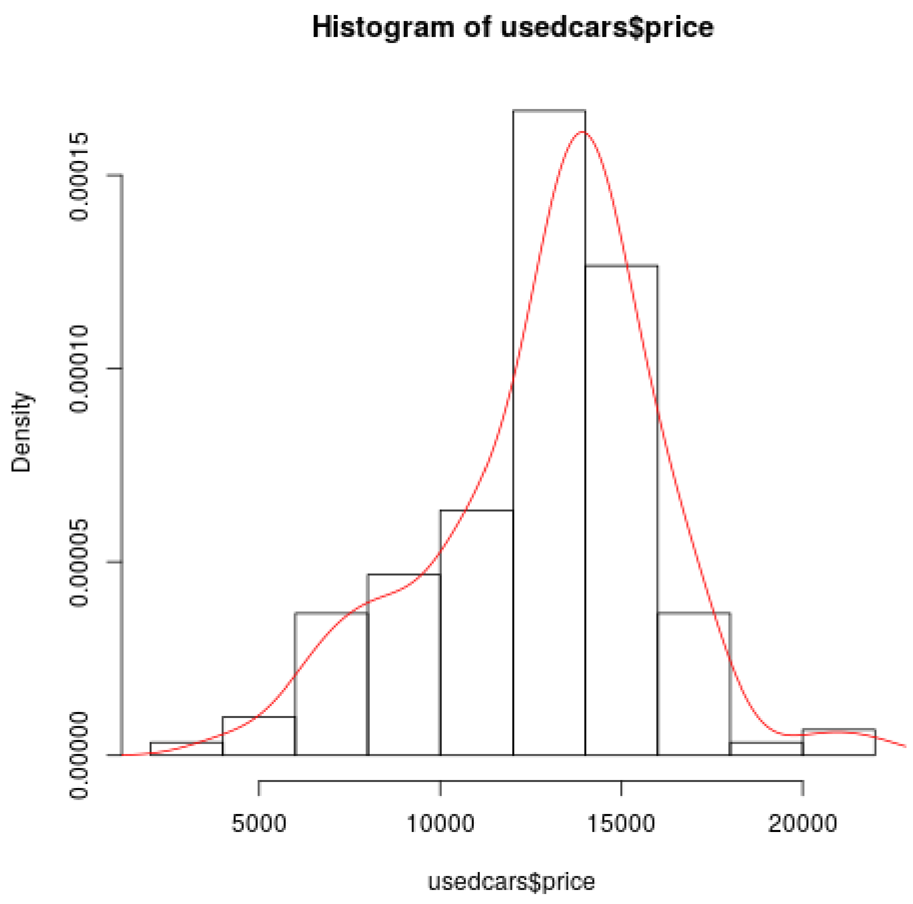


What about a comparison of the histogram to a kernel density estimate of the usedcars$price distribution? For this purpose, we'd first normalize the distribution:

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

and then add another plot using the lines command:

```
> lines (density (usedcars$price), col="red")
```



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 "graphics.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.

Exploring categorical variables

Categorical data are often examined using tables -- these are basically frequency counts for the occurence of the categories. For instance, to examine the "color" field of the usedcars dataset, we could use:

```
> table (usedcars$color)
```
which produces the output:

```
Black Blue Gold Gray Green Red Silver White Yellow
35 17 1 16 5 25 32 16 3
```

i.e. we have 35 black cars, 17 blue cars, etc. We could also get proportions by using the prop.table () command and applying it to a table, i.e.:

```
> mytable <- table (usedcars$color)
> prop.table (mytable)
```
will now output proportions of cars belonging to the respective category

```
Black Blue Gold Gray Green Red Silver White Yellow
0.233333333 0.113333333 0.006666667 0.106666667 0.033333333
0.166666667 0.213333333 0.106666667 0.020000000
```

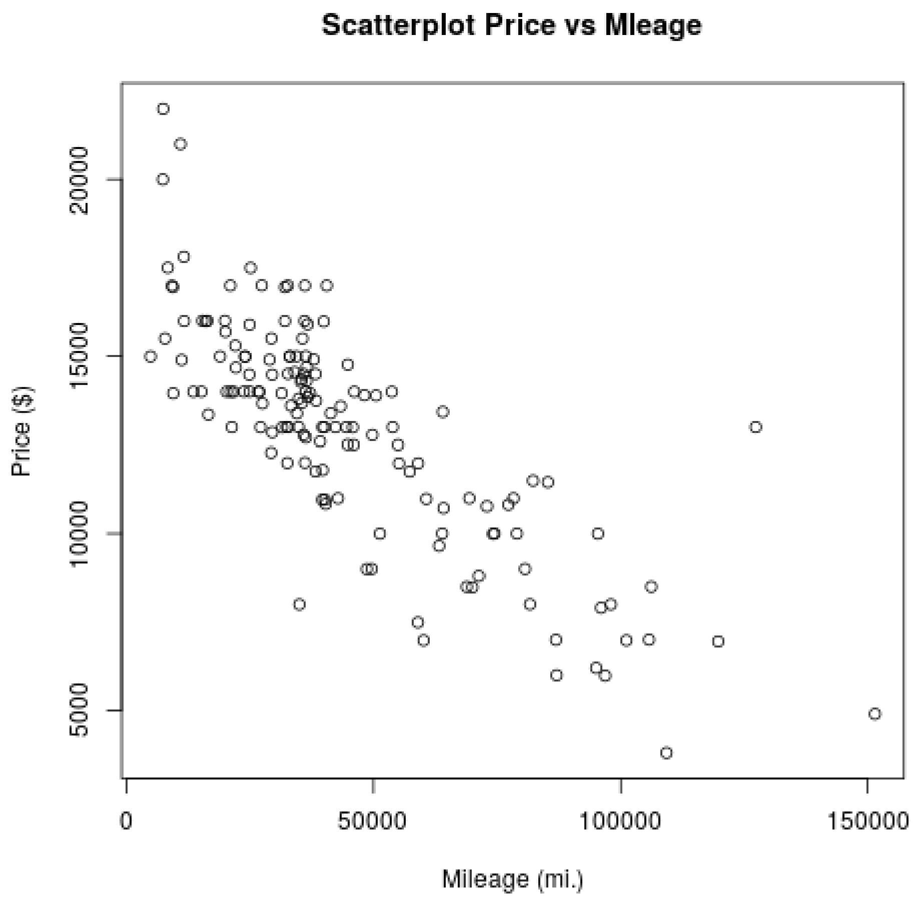
Relationships between variables

So far we have been examining one variable at a time, i.e. we have only analyzed univariate statistics. Often we are also interested in questions regarding the relationship between variables. In case of two variables, we'd have a look at bivariate relationships, and so on.

Scatterplots

are diagrams that visualize bivariate relationships, we basically plot one feature of the data set vs. another feature. For instance, we might be interested in possible relationships between price and mileage in the usedcars data set. Typically, when plotting such relationships, one will assign the y-variable to the dependent variable, e.g.:

```
> plot (x=usedcars$mileage, y=usedcars$price, main="Scatterplot Price vs Mleage", xlab="Mileage (mi.)", ylab="Price ($)")
```



Note, that alternatively, we could have used the "~" sign in the plot function to plot y vs. x, i.e.

```
> plot (usedcars$price~usedcars$mileage, main="Scatterplot Price vs Mleage", xlab="Mileage (mi.)", ylab="Price ($)", type="p")
```

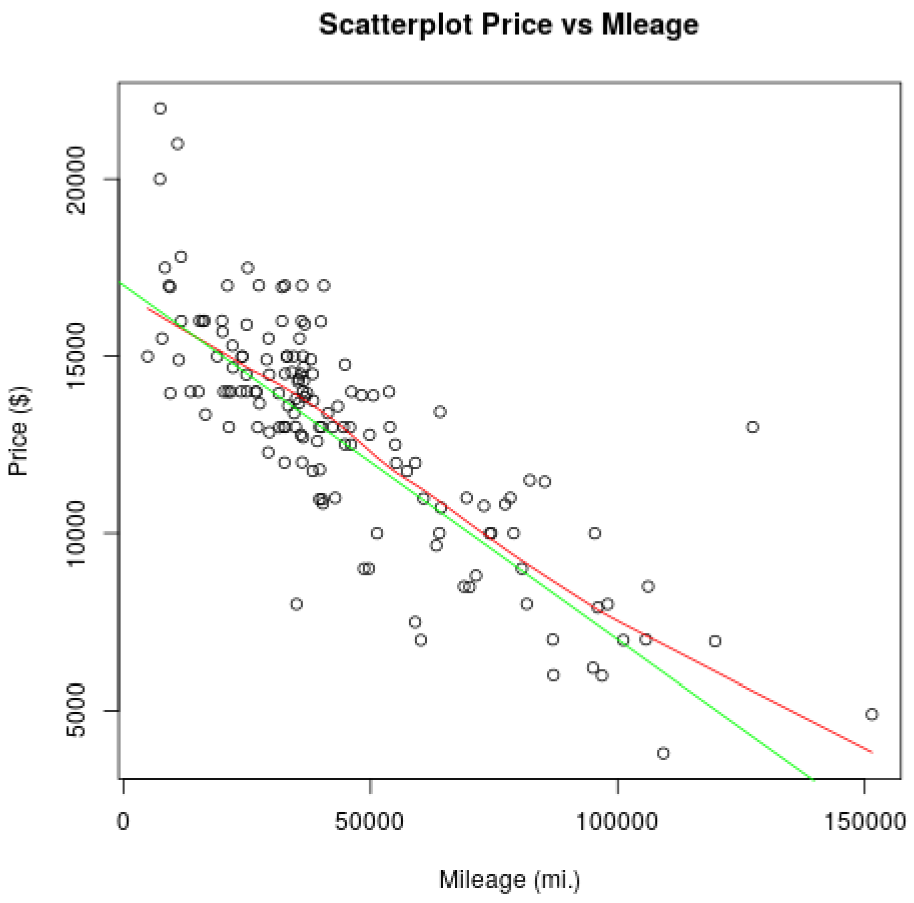
would have produced the same plot with a bit less typing. I also specified that data points should be plotted as points by giving the option type="p" -- one could have chosen type="l" for lines, type="b" for both are some others (used ?plot to explore).
The plot seems to indicate a negative relationship between mileage and price

of car, not very surprising. Suppose I want to indicate this relationship in the figure? I could, e.g., do this by adding a straight line:

```
> abline (17000, -.1, col="green")
```

which will add a green line with intercept \$17000 and slope -.1\$/mile to the graph. This is probably not the best indicator of the relationship between price and mileage. Alternatively, we could also use the "lines" command to add all sorts of different lines to the graph. For instance, we could combine this with the "lowess" command (technically: lowess produces a locally weighted regression line) to generate a curved line that indicates the relationship:

```
> lines (lowess (usedcars$price ~ usedcars$mileage), col="red")
```



Using the lines command, or the similar points command, it is possible to add data from additional variables or even from different data sets to the same graph. If you are planning to do this, you might need to use the xlim and ylim options to the plot command to set an appropriate x or y range for the initial plot. For example, plot (usedcars$price~usedcars$mileage,ylim=range(0:5000)) would force the y-axis to a range of zero to 5000.

Let us come back to the used cars data set and have a look at relationships between categorical variables. For instance, we might be interested in questions like whether the type of transmission (manual or automatic) has an effect on price. To test this, we could simply compare mean values of prices of cars with automatic or manual transmission. One way to approach this question is to first select cars with a specific type of transmission and than investigate summary statistics, e.g. we could use:

```
> summary(usedcars[usedcars$transmission=="MANUAL",])
> summary(usedcars[usedcars$transmission=="AUTOMATIC",])
```

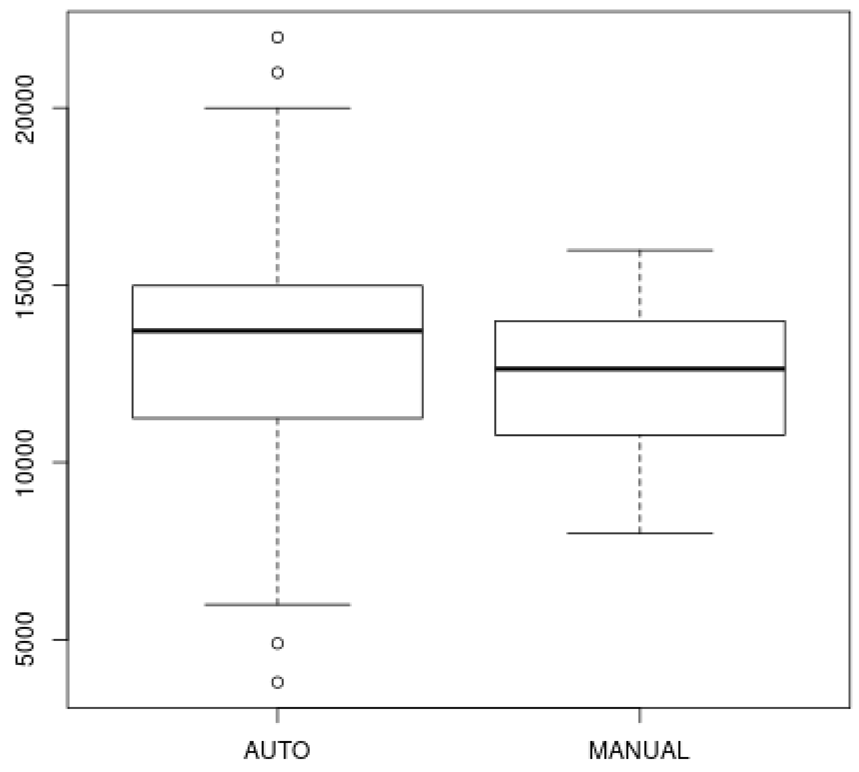
and would find values of $12200 and $13093, respectively. This can also be done much quicker using the tapply command:

```
> tapply (usedcars$price, usedcars$transmission, summary)
```

which tells R to split the field usedcars$price of the usedcars data frame by the variable transmission and then apply the command summary to the result.

We might now wonder if this allows us to conclude that cars with manual transmission are on average cheaper than cars with automatic transmission? We will discuss this type of problem in more detail in the lectures. For the moment, we would hypothesize that the distribution of car prices around the mean values would surely affect our verdict. To have a look at this, we should produce suitable boxplots that visualize both distributions in the same graph. We can do this via

```
>boxplot (usedcars$price ~ usedcars$transmission)
```



We see that there is quite a bit of overlap between both distributions, i.e. there are many cars with manual transmission that are more expensive than cars with automatic transmission. We might have to be a bit more careful with our statistics here -- which factors would we have to consider?

Heat maps and contour plots

Sometimes you want to look at how one variable is related to two others simultaneously. For example, in a demographic data set you might want to explore income as a function of both IQ and education. Contour plots, heatmaps, and 3D surface plots are the appropriate tools for this, and they are all possible in R.
These types of plots expect the data to be in a slightly different format to what we have seen so far. Instead of a simple rectangular file, we expact the data to be in the form of a vector of x values, a vector of y-values, and a matrix of the z values we are going to plot. If you have your own data set which is arranged in rows and columns already, you can have R read it in as a matrix using the following:

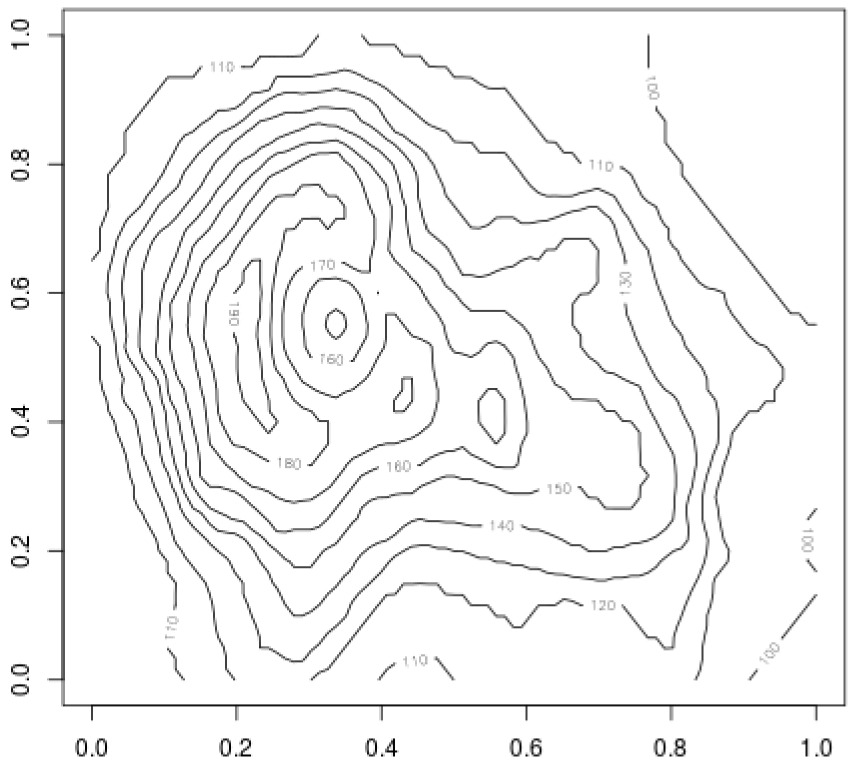
```
> matrixData <- as.matrix(read.table ("file.dat"))
```

For now we will just use one of R's built-in data sets, the "volcano" data set, which is already in the correct format. We can load it using:

```
> data (volcano)
```

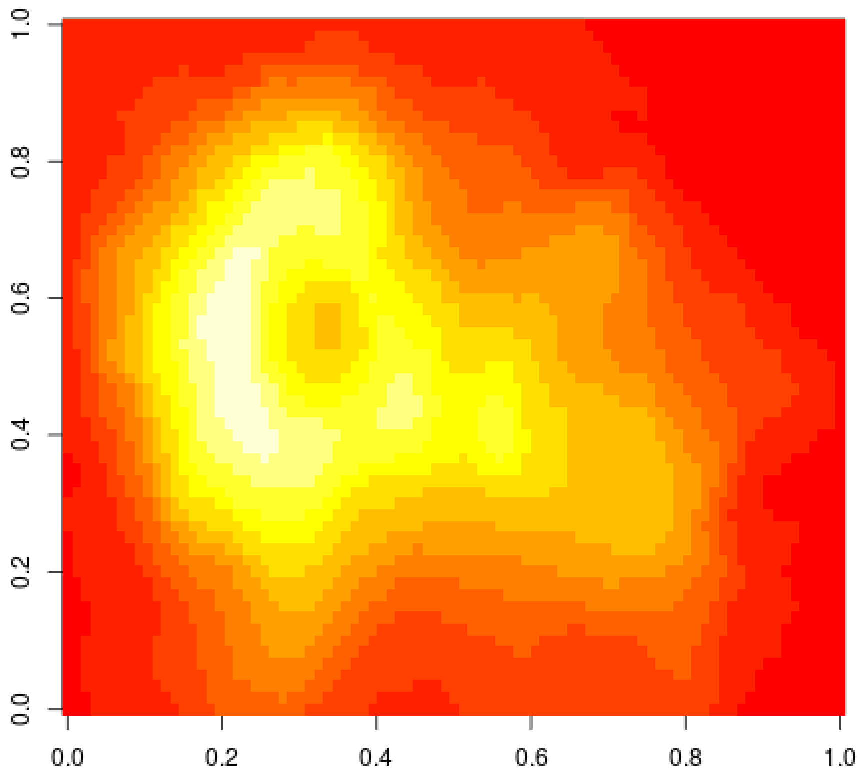
For a contour plot use:

```
>contour (volcano)
```



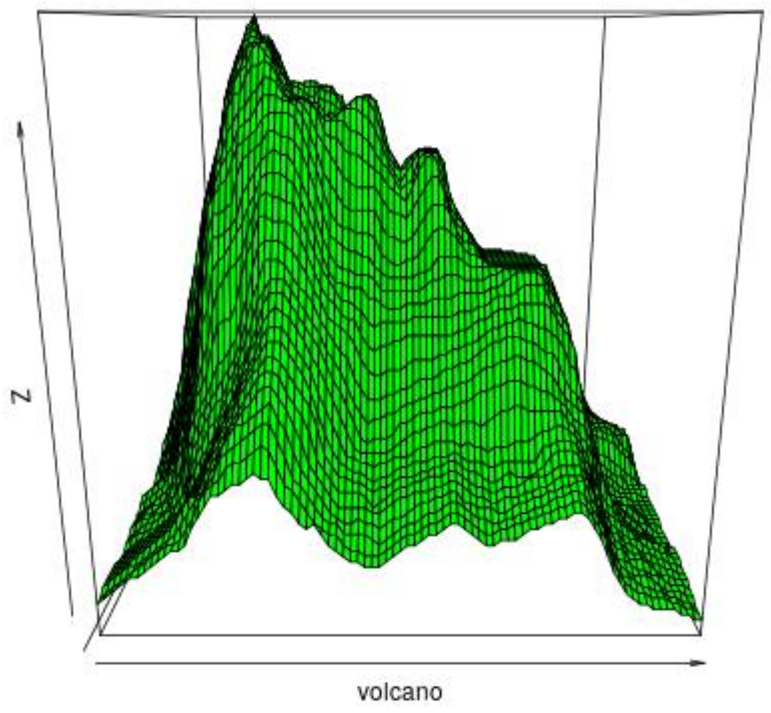
For a heatmap use:

```
>image (volcano)
```



For a 3D plot use:

```
>persp (volcano, col="green")
```



where I have set the color option to green in the last one. All of these commands come with a plethora of options, explore at your own convenience using the help function. If you use latex as a text processing system, pdf files should import best. For word-based processors, png-files should be most convenient to import. For more information and to explore other options, try help(Devices), help(postscript), help(pdf), help(png), etc.

Quantifying relationships between variables

Correlation coefficients are a basic statistical tools for describing relationships between variables -- we will talk about the various types of correlation coefficients in more detail in the lectures later. For now, a correlation coefficient is a number that ranges between -1 and +1, and describes the strength and direction of the linear relationship between two variables.

- A positive value correlation means a linear relationship in the positive direction, i.e. as the first variable gets bigger, the second gets bigger as well. Think of the relationship between height and weight -- we expect that, on average, somebody of large height also has larger weight, but this is not always the case. The result would be a positive correlation coefficient between zero and one.

- A correlation of zero, or close to zero, indicates the absence of a (linear) relationship. This is another way of saying knowledge of variable one tells us nothing about variable two. E.g. the correlation between peoples IQ and their house numbers is surely close to zero. (Something to be aware of is that you can have close to zero correlation even though there might be a strong non-linear correlation in your data!)
- A negative correlation indicates a linear relationship for which one variable decreases as the other increases. An example of a moderate negative correlation might be the relationship between the age of a driver and the speed of vehicle on the motorway.

Although there are no hard-and-fast rules about what counts as a high or low correlation coefficient, a very approximate guide to these numbers might be as follows:

- 0...0.3: Weak relationship; may be an artefact of the data set and in fact there is no significant relationship at all
- 0.3-0.6: Moderate relationship; you might be on to something, or you might not
- 0.6-0.9: Strong relationship; you can be confident that the two variables are connected in some way.
- 0.9-1.0: Very strong relationship, these two variables are virtually measuring the same thing.

We will talk about the significance of correlations in a bit more detail in a later lecture.
Let's come back to our used car example. Our previous plots indicated the presence of a negative relationship between price and mileage, i.e. as mileage gets larger, we typically find lower prices. We can quantify this with a correlation coefficient using the R-command "cor". However, cor requires a numeric data frame as input, so I first need to transform the usedcars data:

```
> df <- data.frame (usedcars$mileage, usedcars$price)
> cor (df)

usedcars.mileage usedcars.price
usedcars.mileage 1.0000000 -0.8061494
usedcars.price -0.8061494 1.0000000
```

gives a matrix of correlations between all columns of df. Here we note the presence of a strong negative relationship. However, when plotting the relationship we noticed that it is not exactly a linear relationship, we'll see how to quantify this in a bit. By default cor uses a Pearson correlation coefficient, if relationships are monotonic, but not exactly linear, we might try spearman's rho or kendall's correlation coefficient instead.

When exploring the links between variables in a data set that has many variables, the matrix of correlations can be an excellent place to start. To get some experience of this, let us use one of R's inbuilt data sets, describing the relationship between various social measures across different regions of Switzerland. Type:

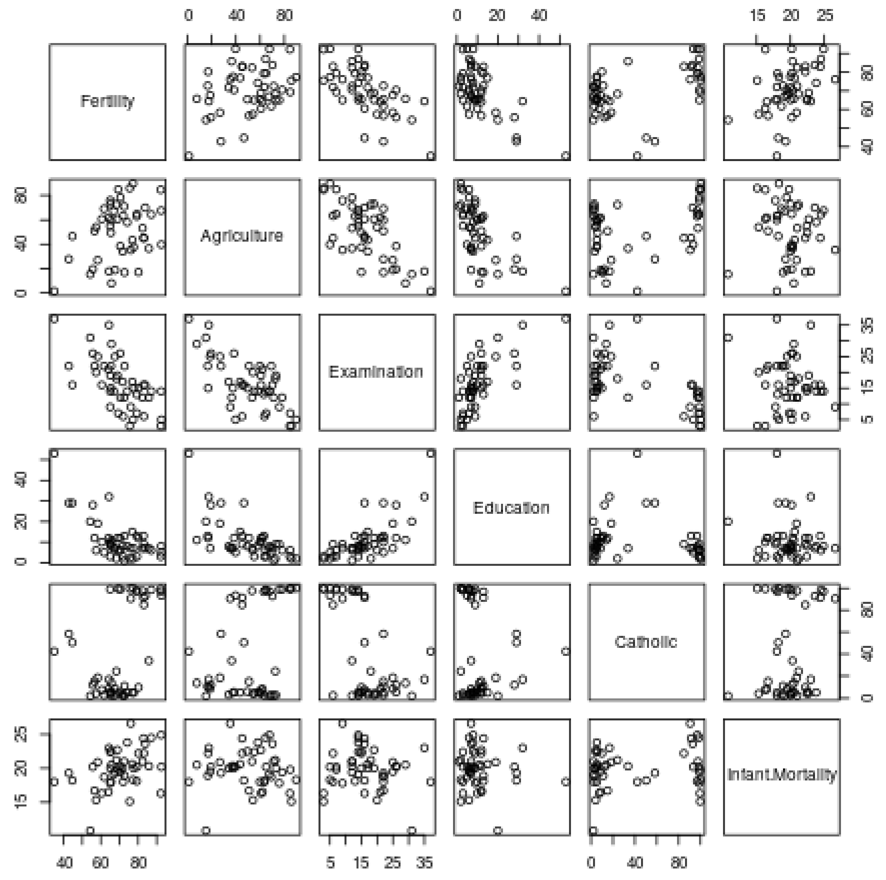
```
> data (swiss)
```

to load the data file. (Try data () to see which other built-in data sets are available if you would like to experiment with them.)

Typing cor (swiss) will give you a matrix of correlations between the six regional variables in the swiss data set: fertility (i.e. birth rate), level of agriculture, average exam results, educational level, prop. of Catholics in the local population, and infant mortality. Have a look for strong and weak correlations. Do the strong correlations make sense?

R provides a visual analogue of the correlation matrix. Try the command pairs (swiss):

```
> pairs (swiss)
```



This plots a scatterplot for the relationship between each pair of variables in the data. Look for a graph that shows a strong linear relationship and then check the corresponding correlation in the output of cor (swiss). This is a good way to get a feel for what various levels of correlation actually mean in practice.

Part3 -- Foundations of Data Science

This is a tutorial on some functions in R for COMP6235 Foundations of Data Science -- part III. This follows on from part II of the tutorial.

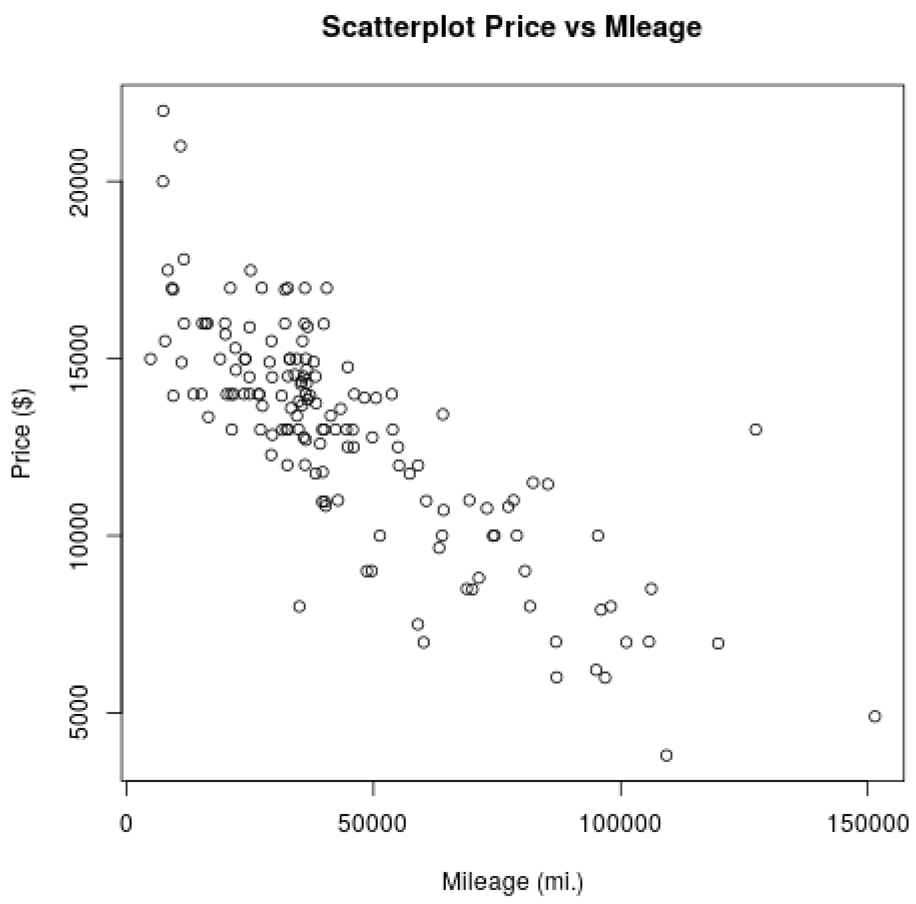
Fitting statistical models

In this section we are moving onto a more advanced statistical concept: the idea of fitting a model. Fitting a model generally means asking the question: "How well can I explain a given set of data with a simple model that relates the variable I am interested in to the other variables?". We'll start with simple models and make build on model complexity as we go.

Fitting a simple linear model

To illustrate the concept, let's return to the used cars data set we explored before. You might remember that we found a negative relationship/correlation between the price of the car and mileage.

```
> plot (x=usedcars$mileage, y=usedcars$price, main="Scatterplot Price vs Mleage", xlab="Mileage (mi.)", ylab="Price ($)")
```



which we had before approximated by a linear relationship (which I estimated by just looking at the cloud of points). The underlying model behind this reasoning was the hypothesis that on average the price of a car decreases by -0.10$ per mile it has been driven. Can we be more systematic about this? R makes it easy to look at the degree to which a theory like this is supported by the data. The relevant command is "lm" which means fitting a linear model to the data. The easiest way to understand the fitted linear model is to think about trying to draw a straight line on a graph that stays as close as possible to all of the data points of mileage/price of car points in our data set. In fact, R minimizes the square distance of the model from the data points or -- in other words -- carries out linear regression.

In the following, we will give the fitted model a name (lm1) (for "linear model #1"). We build it as follows:

```
> lm1 <- lm (price ~ mileage)
> summary (lm1)
```

Our output should look similar to the following

```
Call:
lm(formula = price ~ mileage)

Residuals:
Min 1Q Median 3Q Max
-5830.9 -1113.7 40.5 992.6 7782.4

Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.709e+04 2.915e+02 58.63 <2e-16 ***
mileage -9.329e-02 5.629e-03 -16.57 <2e-16 ***
---
Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1854 on 148 degrees of freedom
Multiple R-squared: 0.6499,Adjusted R-squared: 0.6475
F-statistic: 274.7 on 1 and 148 DF, p-value: < 2.2e-16
```

The output from the summary of this linear model might look confusing at first. We don't have the time to make this a full course in statistics, so we'll have to skip many of the details. Here are the elements we need to consider:

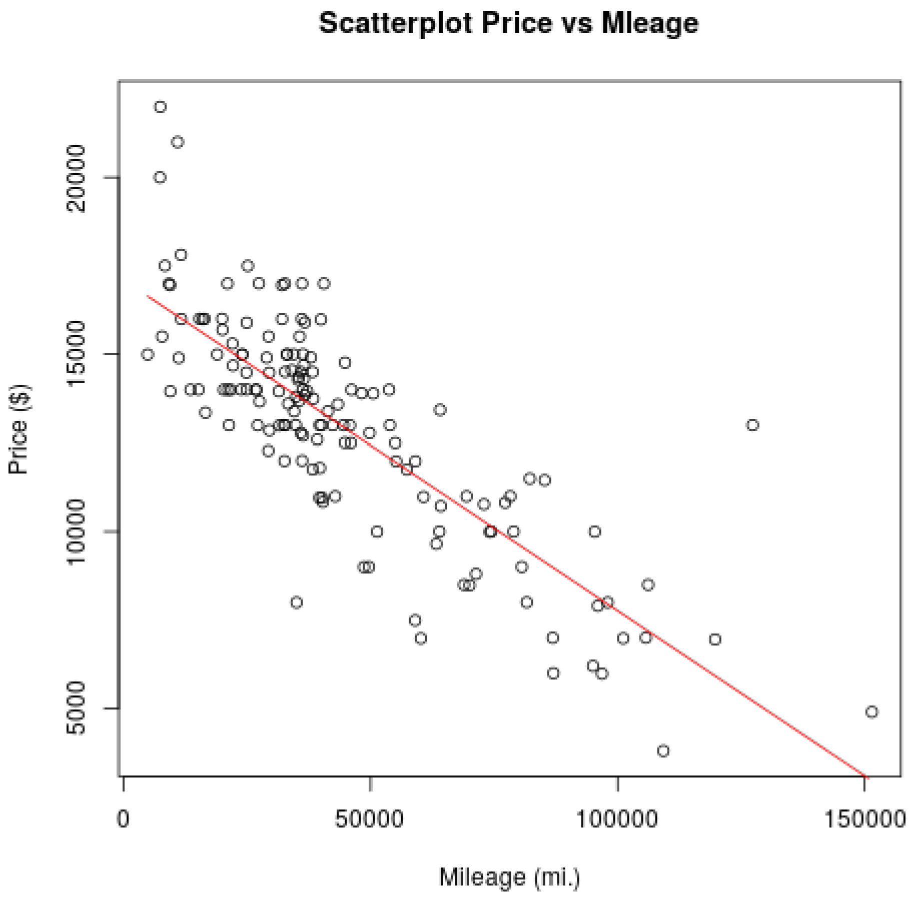
- The estimated intercept and mileage coefficients in the middle of the output. These numbers describe the equation for the regression line we have just fitted. The value of "1.709e+04" for the intercept can basically be interpreted as the average price of a new car (i.e. a car that has driven zero miles), roughly $17000. The mileage coefficient "-9.329e-02" gives the decrease in price of an average car per mile driven, i.e. roughly -0.10$ per mile, close to what I estimated by just looking at the data before.
- The "Adjusted R-squared" value on the right of the second last line. This number is always between 0 and 1 and gives the proportion of the variation in price that can be accounted for by mileage, i.e. around 65% in our case. To get some idea of the scale of these values, they are equivalent to taking a correlation coefficient and squaring it. So the value of 0.65 should be interpreted much like a correlation coefficient of 0.80: a fairly strong relationship.
- The "p=value" associated with the F-statistic, on the right side of the last line. P-values are probably the most misunderstood concept in statistics. They are best explained by considering that most statistical tests ask the question "If in fact the variables I am looking at are unrelated, how much of a fluke would it be to get results as weird as

these?". In other words, what is the probability of something that supports the model at least as much as the current data set does, if there is actually no relationship between the variables in the model? Very small p-values mean that the data would be unlikely to be observed in the case of no relationship, and thus they are generally taken as evidence in support of the model. Larger p-values (i.e. larger than about 0.01 or 0.05 in practice) are often taken as evidence against the model, as the data are not giving us strong reasons to doubt the no-relationship hypothesis.

In this case we find a p-value: < 2.2e-16, i.e. the linear model assumption works very well.

Because the fitted linear model has been stored in the R-object "lm1" we can use this object ot obtain a plot of the regression line and judge for ourselves what sort of fit has been achieved. The command to do this is as follows:

```
> lines (predict(lm1) ~ mileage, col="red")
```



Fitting polynomials

The fit achieved by the linear model of the price-mileage relationship is good, but does not explain all of the variation in the data. We could probably do much better if we were not limited to straight lines. R makes it simple to fit a polynomial model to the data, i.e. a model that employs a prabola (or higher order polynomial relationship) instead of the straight line. We can use the same "lm" command. (Of course, if this were real analysis, it would be strongly desirable that we have some theoretical reason for believing that a polynomial model might apply here before going ahead.) The command is:

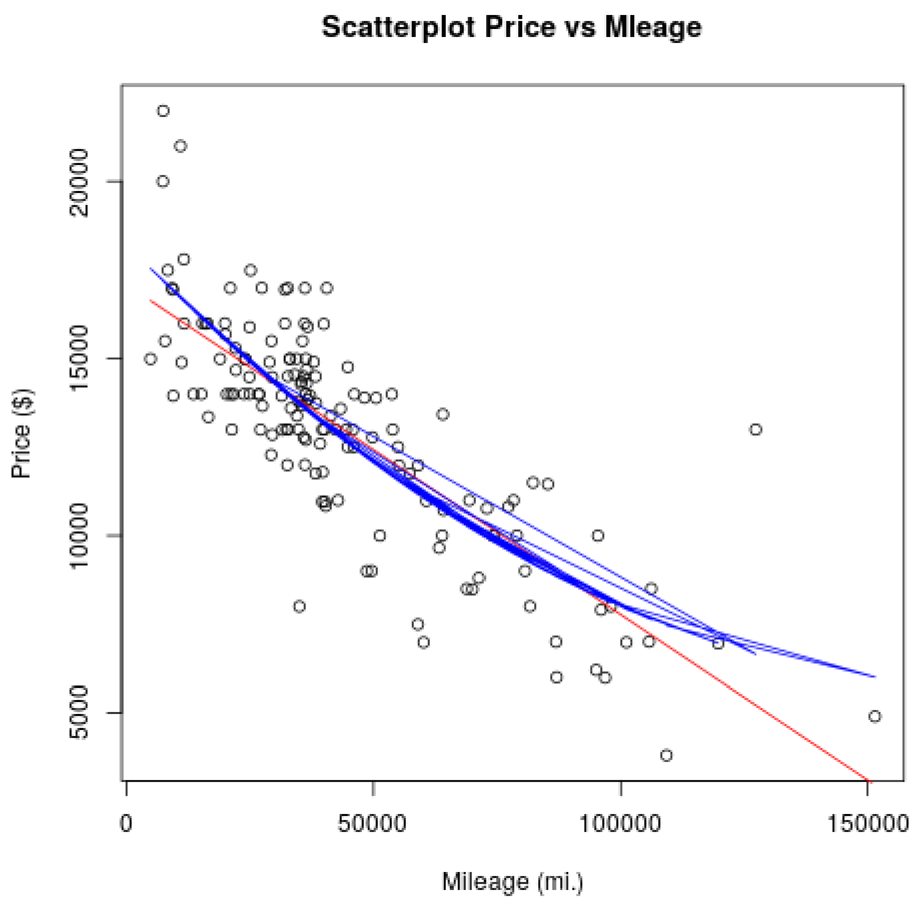
```
> lm2 <- lm ( price ~ poly (mileage, 2))
```

Note that we are fitting a polynomial model of degree 2, which just means that we are looking at both mileage and mileage^2 as predictors of price. You can try "summary (lm2)" to see the details of the fitted model and you will notice that we have slightly improved our adjusted R-squared value in this case (0.6649 for the polynomial model vs. 0.6475 for the linear model).

If the previous plot window is still open, we can add another relationship:

```
> lines (predict (lm2) ~ mileage, col="blue")
```

and obtain the following plot:



What went wrong here?

Actually, this is quite a pain and has something to do with the way how R plots the predicted line and how our data set is sorted. Namely, our data set is not sorted by mileage and so R just steps through the data points in the order in which they are in the data set, predicts, and connects the dots. This did not matter for a straight line (because the many lines effectively drawn by R all overlap), but it matters for any "curved" relationship. How to get around it?

Well, my best guess is to start from the beginning and build a new sorted data frame:

```
> df <- data.frame (x=sort(mileage), y=price[order(mileage)])
```

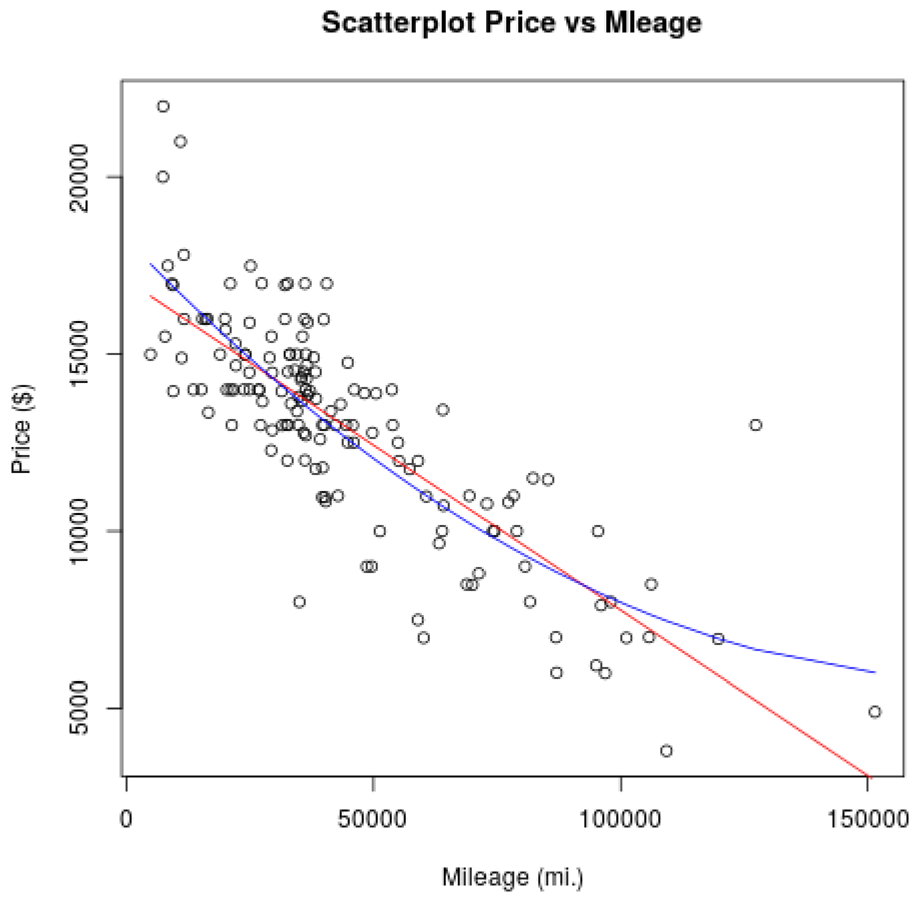
which now contains the sorted mileage variables and the corresponding prices. I then again build our linear models

```
> lm1 <- lm ( df$y ~ df$x)
> lm2 <- lm ( df$y ~ poly (df$x, 2))
```

and plot everything:

```
> plot (df$y ~ df$x, main="Scatterplot Price vs Mleage", xlab="Mileage
(mi.)", ylab="Price ($)")
> lines (predict (lm2) ~ df$x, col="blue")
> lines (predict (lm1) ~ df$x, col="red")
```

with a slightly more pleasing result:



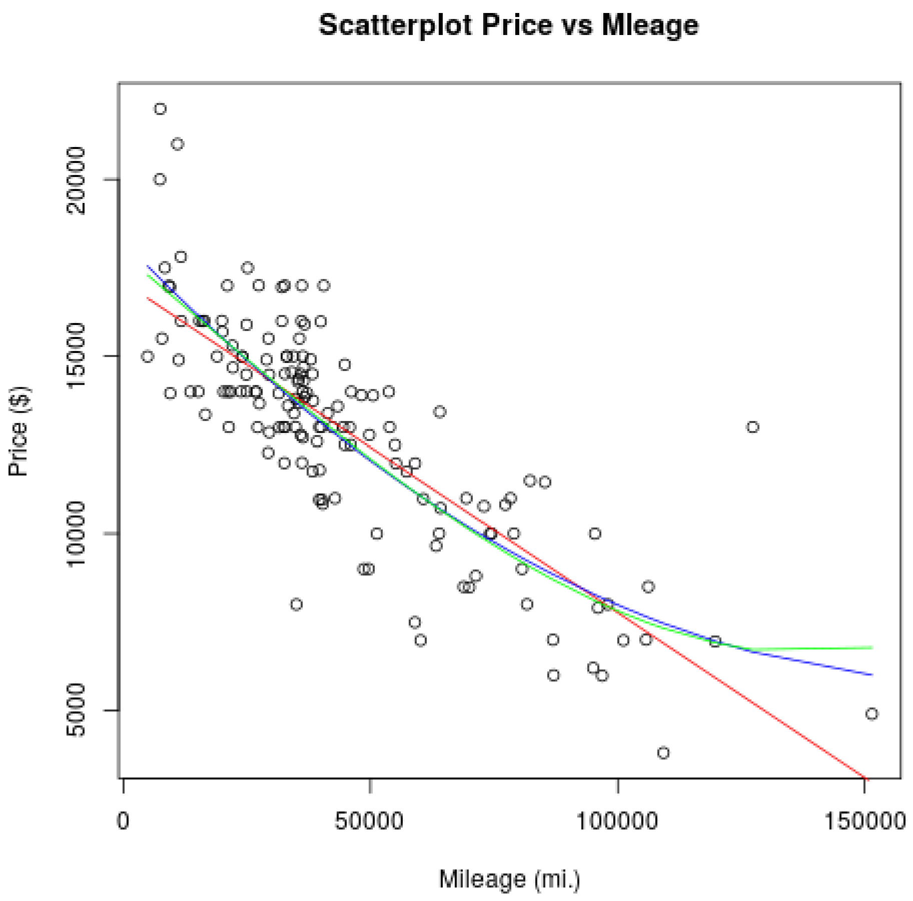
Looking hard we can see that the blue line on the graph gives a slightly better fit to the data than the red line. We should be sceptical, however, about the idea that we've found the "true story" behind these data. The blue line gives a parabola and would (if we extended the x-range for long enough) predict that the trend that car prices decrease as mileage increases reverts at some stage.

No need to stop at degree 2 polynomials. With the following commands, we can fit an even more flexible curve and add it to the plot:

```
> lm3 <- lm ( df$y ~ poly (df$x, 3))
```

which we can add to our plot as a green line via:

```
> lines (predict (lm3) ~ df$x, col="green")
```

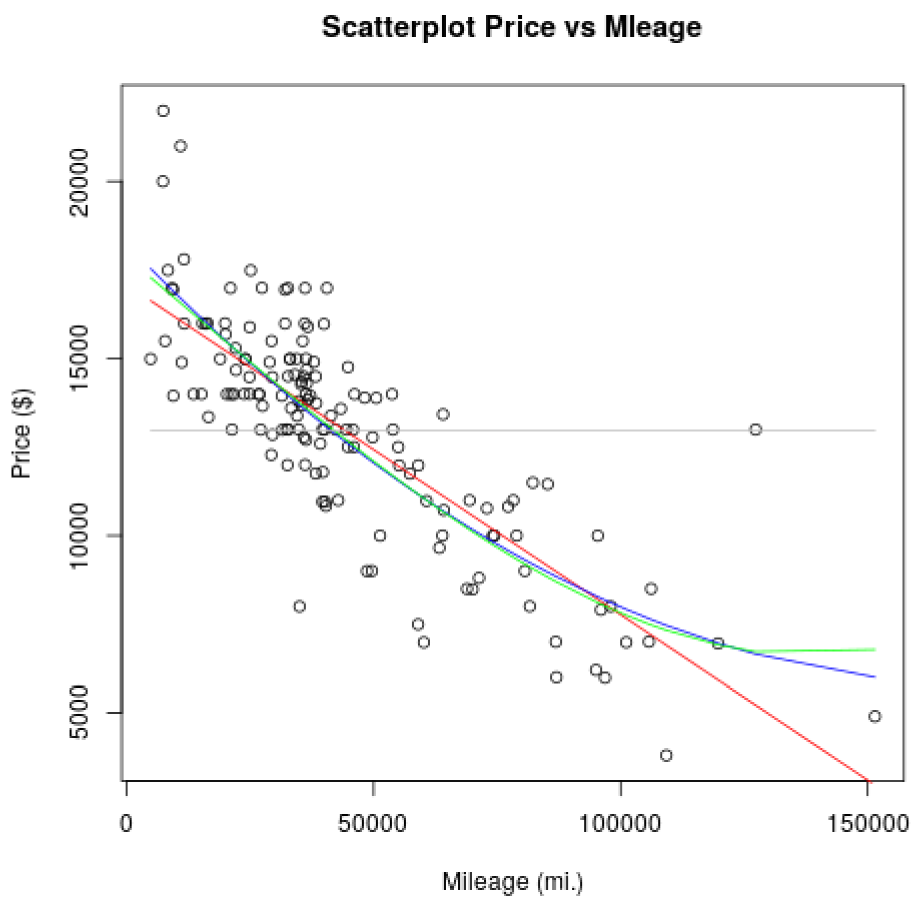


Notice that the green line doesn't seem to be much of an improvement over the blue line, however.

Choosing between alternative models

Now that we have lm1, lm2, and lm3, a natural question arises as to which one is the best model. And in order to settle this question, we actually need to introduce one more model into the contest

```
> lm0 <- lm ( df$y ~ 1)
```



This final, rather boring, model is shown in gray in the graph. The call to lm asks R to build a model of price (aka df$y) with nothing but the integer one as a predictor. Thus the model fitting algorithm has nothing to go on, and all it can do is predict that the value of price will always be equal to its mean: that's why the gray line is flat. This sort of model is often called the null model, for obvious reasons.

There are various ways of determining which model is best, e.g. stepwise model reduction . A statistician named Akaike devised a very elegant way of determining which model should be preferred in cases like ours. He developed a measure, known as Akaike's Information Criterion , or AIC for short, that can be used to rank alternative models. AIC rewards a model for the amount of variation it explains, so that the blue and green curves in our case should score highly. However, AIC also penalizes a model for including too many variables, so that especially the green curve will suffer here, whereas the gray curve might do well. So you see that AIC embodies a trade-off between prediction and simplicity. You don't need to know how to calculate AIC, as R will do it for you. All you need to remember at this stage is that smaller is better. The R command to compare our four models is as follows:

```
> AIC (lm0, lm1, lm2, lm3)
```

and we find that the lowest score is obtained for the model with polynomial degree 2 (i.e. 2680.587), hence the polynomial model is the most reasonable of the four models considered.

(Note: If you have done stats before, you might wonder why the more long-established criterion of analysis of variance was not used here. You can use this in R via the anova () command, however, the AIC method is more universal and does a better job of capturing scientific parsimony and is probably easier to learn to use.)

A note on linear models for categorical data

The lm command in R is rather powerful and can be used to fit models to categorical data as well. Let us return to the used cars data set which contains various categorical fields. For the moment, we might be interested in the field "model" which has the three levels "SE", "SES" and "SEL". We might be interested in figuring out whether the car model has an effect on price and would also want to consider an appropriate null model to check of our model fit is meaningful:

```
> flm0 <- lm ( price ~ 1)
> flm1 <- lm ( price ~ model)
> summary (flm1)
```

Amongst other stuff, we'll obtain the following output:

```
Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 12202.1 320.3 38.095 < 2e-16 ***
modelSEL 3907.1 671.2 5.821 3.53e-08 ***
modelSES 492.0 515.7 0.954 0.342
---
Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2829 on 147 degrees of freedom
Multiple R-squared: 0.1902,Adjusted R-squared: 0.1792
F-statistic: 17.27 on 2 and 147 DF, p-value: 1.837e-07
```

Note, that the adjusted R-squared value is not very impressive, i.e. only a very small portion of the variation can be explained by car model type, but the p-value indicates that this model might be significant. Of primary interest in the above are the coefficients. The first coefficient corresponds to the reference level -- in this case R has chosen "SE" and gives the average price for a car of that type. The other coefficients are relative to the reference level, i.e. have to be added to the reference level. Thus, a "SEL" car has an average price of 12202+3907 and an SES car 12202+492 $. R also automatically carries out t-tests if the coefficients are significant and here we'd notice that the coefficient for SES is not significant -- hence we could not be sure is SES cars are indeed more expensive than SE cars. In contrast, our confidence that SEL cars are more expensive than SES cars would be high.

As a last step we might want to figure out if our model is sensible at all, by carrying out an AIC analysis. Using

```
> AIC (flm0, flm1)
```

we find that the AIC score for flm1 is indeed lower than the score for flm0, hence the preferred model is flm1.
We could also reverse the logic here and attempt to predict a categorical variable using a continuous(ordinal) variable as a predictor. In this case we end up in the territory of logistic regression (and related models). To learn more about regression models for categorical variables, you might also want to have a look at the corresponding R-command "glm ()".

Scripts in R

If you find yourself doing the same thing in R over and over you should switch from interactive mode to batch mode, i.e. you should write a script (effectively a short R program) in order to do the repetitive operations in R for you.Here is an example script that loads the usedcars data set and plots the relationship between price and mileage into a pdf called example.pdf. In order to run scripts in R you can do either of two things:

- The first is that you can run R from a command line environment (e.g. from a linux system) using:

  ```
  R --vanilla < script.txt
  ```

  The "vanilla" option is to specify that R should not save the workspace, etc. and the "<" symbol (in a linux environment) means that input to R should come from the named text file. You could use "R --vanilla < script.txt > output.txt" if you want the output of R to be redirected to a particular file (here output.txt) instead of being shown on the terminal.

- The other way of executin R scripts is within the R environment by using the R-command "source ()", i.e.:

  ```
  >source ("script.txt")
  ```

Programming in R

Functions

R has the flexibility to allow for user-defined functions. The typical syntax for defining a function is given below:

```
myfunction <- function(arg1, arg2, ... ){
statements
return(object)
}
```

where "myfunction" is the name of the function, "arg1, arg2, ..." is a list of inputs to the function which is enclosed in brackets. The body of the function starts with "{" and ends with "}". Then a list of statements or operations follows, ending by a return statement specifying which object is to be returned (Actually: return is not needed, R will return whichever variable is in the last line of the function.)

Let's say we want to build a function that returns the mean and the standard deviation of a sample in one go. We could do this as follows:

```
> meansd <- function (sample) {
+ df <- list (center=mean(sample), spread=sd (sample))
+ return (df)
+ }
```

Note that R continues lines with the "+" sign once you press enter after an opening brace "{" and will change back to standard input mode after the next closing brace "}". In our function we assign the name "meansd". You can later retrieve the source code of this function by typing the name of the function without "()", i.e. if you type "meansd" R will display the corresponding source code. Next, we define a data structure which will enclose all of the results to be returned. In our case, a list seems appropriate and we create two fields called "center" and "spread" within it. The last statement in the function's body returns our list of results. Let's test it:

```
> sample1 <- rnorm (100, 0, 1)
> sample1
```

i.e. I create 100 random variates from a normal distribution with mean 0 and standard deviation 1. Now I can use our convenient way of calculating mean and standard deviation in one go:

```
> meansd (sample1)
```

and obtain the following output

```
$center
[1] -0.01405578

$spread
[1] 0.9821303
```

i.e. not surprisingly fairly close to the parameters I used to generate the random variates.

Control structures

R allows for all the usual types of control structures typically present in programming languages. For example, you can use conditional expressions following the syntax:

```
if (cond) expr
if (cond) expr1 else expr2
```

Use "for loops" following

```
for (var in seq) expr
```

Use "while loops" following

```
while (cond) expr
```

and others like "switch" or "ifelse" etc. Check the R documentation for more detail.

Let's suppose we want to build a function which first checks the input data structure if it is a vector and then calculates the population version of the standard deviation of our sample. We could do it as follows:

```
mysd <- function(x) {
if (!is.vector(x)) {
```

```
warning("argument is not a vector!")
return (NA)
}
s=0.
s1=0.
for (i in 1:length(x)) {
s=s+x[i]
s1=s1+x[i]*x[i]
}
s=s/length(x)
s1=s1/length(x)
s=sqrt(s1-s*s)
return(s)
}
```

As a last step, let's test this function. For this purpose I generate 1000 normally distributed random variates from a normal distribution with mean zero and variance 1.

```
> x<- rnorm (1000,0,1)
> mysd(x)
```

for my particular random seed I get a value of 1.037462 which is not too bad.