Statistical Computing

Master Data Science

Winter Semester 2017/18

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Prof. Tim Downie

Workshop 4
Basic statistics and exploratory data analysis II

1 Introduction

This workshop is a continuation of Workshop 3. The main focus is on manipulating data frames so that you can shape the data set into one which is convenient to analyse. The second main theme is an introduction to statistical hypothesis tests used in statistical data analysis. The general ideas will be presented using one specific test called the t-Test.

1.1 Preliminaries

Download the data files from the course Moodle page into your `H:\\StatComp` directory. The files are: `Wkshp4test.dat`, `WideData.dat` and `presidential`. You will also need `Medication.csv` which you used last week. Check that you can find this file and download it if you can't find it.

Start RStudio using the Icon on the desktop

Strg+Shift+N opens a new R script.

Type the comment

```
#Statistical Computing: Workshop 4
#Basic Statistics and exploratory data II
```

as the title.

Save the file in your `H:\\StatComp` folder with the name `Workshop4.R`.

Set your *working directory* to be `H:\\StatComp`. The code to do this is

```
> setwd(H://StatComp)
```

By now it is possible that you have accidentally saved your workspace when leaving RStudio. If your Environment Window is displaying any objects then clear the workspace using *Session > clear workspace*.

As always, work through the examples making sure you understand the code and the output and don't forget to add plenty of comments.

1.2 Installing R packages

There are many external packages in R which can easily be downloaded and installed. One of the aspects of R, which makes it very useful for statistical analysis, is that so many contributors have provided methods in very many different areas of statistics and data science.

You only need to install each package once. After you have installed it use the command `require` or `library` once per R-session. Today you will be using the package `dplyr`, a not very helpful name for the package "a grammar of data manipulation".

If you are using one of the lab computers, this package has already been installed on the network for you. Start the package using:

```
> require(dplyr)
```

and jump to Section 2.

If you are using your own computer:
First of all check if this package is already loaded:

```
> find.package("dplyr")
```

If you get a directory name then it is installed. Start the package

```
> require(dplyr)
```

and jump to Section 2.
If you get an error then you need to install the package, which you do from within RStudio.

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

Follow the instructions, which may take a minute or so. If a dialogue box asks you if you want to install this as *administrator* then click yes.

Occasionally there is a problem with the firewall and the user has to quit and RStudio *restart* as administrator. If this seems to be the case, then quit RStudio and "run as administrator" (Windows). If that still doesn't work (or you have a computer where you can't do this) try

```
> debug(utils:::unpackPkgZip)
> install.packages("dplyr")
```

Ask if you need help. Start the package: `> require(dplyr)`

2 Manipulation of data frames: Data wrangling

Last week you learnt how to read data into R, and to calculate basic summary statistics. Often however, it is necessary to rearrange the data in some way prior to analysis. In this section, we'll look in a bit more detail at ways of accessing the columns of a data frame, and explore some ways of manipulating and rearranging them. The reason being that, almost invariably, the first step in any analysis (outside the classroom) is get the data in the right form to analyse.

The language available to manipulate data in R has progressed considerably in the last 10 years, much of it led by data storage and data base methods. The `dplyr` Package is an extension which provides "modern commands" to do the same things as the "traditional R commands" but using different language. It is important to be able to use the traditional R commands even if you plan to just use the modern approach. There are many reasons for this but two important ones are: the concepts are important to understanding R in general not just "data wrangling" and you need to be able to understand other peoples code.

In this Section you will be using a slightly larger version of `test.data` from Workshop 3. This is of course meant to be for simple illustration. Read in the data using
`> test.data <- read.table("Wkshp4test.dat",header=TRUE)`
and view the data frame by clicking on `test.data` in the environment window.

2.1 Accessing one column of a data frame

Last week we found the mean response in our `test.data` example by typing
`> mean(test.data$Response)`. Here are some alternative ways of doing the same thing:

1. `> mean(test.data[,3])`
   This reads as 'calculate the mean of the 3rd column of `test.data`'. Although it's a perfectly valid command, for a human it's not as nice as `mean(test.data$Response)` because it isn't so easy to read.

2. `> mean(test.data[,"Response"])`
   This reads as 'calculate the mean of the column called `Response` in `test.data`.' This notation is easy to read but not as succinct as `mean(test.data$Response)`.

3. `> attach(test.data)`
   `> mean(Response)`
   The **`attach`** command means the variables in `test.data` can be accessed without needing to name the data frame.

   What does attach actually do? Type

```
> search()
[1] ".GlobalEnv"  "test.data" "package:dplyr" "tools:rstudio"
[5] "package:stats" "package:graphics" "package:grDevices" "package:utils"
[9] "package:datasets" "package:methods" "Autoloads" "package:base"
```

Your list of packages will be similar but perhaps not exactly the same. Notice that the second element in the *search path* is called `test.data` the name of the data frame we have just attached. In first place is `.GlobalEnv` which contains all the objects in your Environment Window (notice that it says `Global Environment` near the top of this window)

When you type a command with `Response` in it, R searches for an object with this name within `.GlobalEnv`. If none is found R then searches for `Response` object within `test.data`, and in this case it is found.

```
> pi
[1] 3.141593
> find("pi")
[1] "package:base"
```

This tells us that the constant `pi` is stored in the very last position, in R's Base Package. We can overwrite the value of `pi`!

```
> pi<-3.0
> find("pi")
[1] ".GlobalEnv"   "package:base"
> 2*pi
[1] 6
```

So now objects called `pi` are in two places, in the Global Environment and in the Base Package. When we access `pi` the first is used, so the user defined version is taken in preference.

Please now remove the fool's version of pi!

```
> rm(pi)
> pi
[1] 3.141593
> find("pi")
[1] "package:base"
```

The `attach` command provides a very convenient way to access columns of a data frame. **HOWEVER**, indiscriminate use of `attach` can lead to problems. For example, suppose you have a data frame with a column called `Response` *and* an object called `Response` in your workspace. Every time you ask R for `Response`, it will give you the first one it finds

which may not be what you think you're getting! The problem gets even worse if you've `attached` several data frames simultaneously. As a rule, only `attach` one data frame at a time and, when you've finished with it, use

```
> detach(data.frame.name)
```

which un-attaches that object. Detach does not delete the data frame itself so the data are still available.

If you get confused over what is and isn't attached, type

```
> search()
```

to list all the locations that R searches.

Detach the `test.data` data frame from the search path now:

```
> detach(test.data)
```

2.2 Accessing multiple columns of a data frame

The `$` notation can only be used to access one variable, and in all the examples above a vector war returned. Often we want to access a subset of columns in a data frame.

1. ```
   > test.data[,c(1,3)]
   > test.data[,2:3]
   ```
   This reads as 'return the 1st and 3rd column (or 2nd and 3rd column) of `test.data`'.

2. ```
   > test.data[,c("Treatment","Response")]
   ```
   You should be able to work out what this does

3. The `dplyr` (modern) command to access columns of a data frame is `select`
   ```
   > select(test.data,Treatment,Response)
   ```
   Notice that all the variables are just listed as arguments separated by commas. `select` always returns a data frame, which is why it was not listed in the "Acessing one column" section. To see what the problem is try
   ```
   > select(test.data,Response)
   > mean(select(test.data,Response))
   ```

2.3 Accessing rows of a data frame

You can access one row of a data frame using `test.data[4,]`. Sometimes a data frame has "row names" in which case you can use the row name in quotes. To access multiple rows

1. ```
   > test.data[1:4,]
   ```
   returns a data frame with the rows 1 to 4.
   ```

2. `> test.data[test.data$Sex=="F",]` or
   `> test.data[test.data$Sex=="F" & test.data$Treatment==0,]`
   Note the `==` symbol for testing which elements take a specific value and `&` for the "logical and" operator.

3. The `dplyr` version are somewhat nicer. As the command `filter` expects that the variable names are in the data frame, it is not necessary to repeat `test.data`.
   `> filter(test.data,Sex=="F")`
   `> filter(test.data,Sex=="F",Treatment==0)`

2.4 Adding a new column

The standard way to do this is simply assign something to a new variable of a data frame using `data.frame.name$new.var.name<-`. For example a common calculation in statistics is to calculate the *standardised residual* by subtracting the mean and dividing by the standard deviation.

```
> test.data$StdRsd<-(test.data$Response-
+ mean(test.data$Response))/sd(test.data$Response)
> test.data
```

The `dplyr` function is called `mutate`. The new variable name is given as an argument followed by the calculation formula whilst omitting the data frame name. `mutate` allows you to add more than one column in one go.

```
> mutate(test.data, StdRsd2=(Response-mean(Response))/sd(Response))
```

2.5 Collapsing over subgroups of data

Particularly in the early stages of an analysis, it can be useful to reduce the size of a dataset by examining summary statistics for subgroups of data. For example, suppose you have to analyse daily rainfall data from 100 locations, covering the period 1950 to 2015. The size of such a dataset ensures that even simple analyses are likely to be time-consuming. However, for exploratory purposes we can speed things up by, for example, working with monthly rather than daily data — perhaps monthly means or totals at each site.

The simplest way to do this kind of thing is using the `aggregate` command. For example, suppose we wanted to create a data frame containing the mean response for each treatment in our `test.data` example:

```
> treat.mean <- aggregate(test.data$Response,
+ by=list(Treat=test.data$Treatment),FUN=mean)
```

The data frame `treat.mean` contains a row containing the mean response for each Treatment value. We can compute the sum of the responses, the number of missing values, or whatever, by changing the value of `FUN`. We could also aggregate over each Sex/Treatment combination using

```
> treat.mean <- aggregate(test.data$Response,
+ by=list(Treat=test.data$Treatment,Sex=test.data$Sex),FUN=mean)
```

N.B: if you want more than one summary variable in the output, this is possible, but you need to specify the `FUN=` argument as a user-defined function which is a subject for another workshop.

In some ways the `aggregate` command is similar to `tapply`, which can be used to compute summary statistics over subgroups of data. The basic difference is that `aggregate` is designed to create data frames for further analysis; `tapply` is more often used simply to create tables of summary statistics it creates *arrays* rather than data frames. For example:

```
> tapply(test.data$Response,
+ INDEX=list(Treat=test.data$Treatment,Sex=test.data$Sex),
+ FUN=sum)
```

The `dplyr` approach is to define the group levels in the data frame with `group_by` and then to apply functions to each level using `summarize`. The following command specifies the groups as each level in treatment (0 and 1) and then calculates the number of elements in each level and the mean `Response` in each level.

```
> summarize(group_by(test.data,Treatment),Nobs=n(),MR=mean(Response))
```

Exercise: Use `summarize` so that the levels are defined by `Sex` and `Treatment` in combination. The output should include for each level: the number of observations, the sum of `Response` and mean of `Response`.

2.6 Reshaping data frames and stacking columns

Consider the following situation, which isn't too implausible: a clinical trial is run in which patients are allocated to one of two treatment groups, and three measurements of the same clinical response are taken for each patient. The data are provided in the ASCII file `WideData.dat`

```
> wide.data<-read.table("WideData.dat",header=TRUE)
```

The data have the following format:

```
ID Group Responsel Response2 Response3
AA 0     15.4      9.7       6.2
AB 1     17.6      13.7      12.1
AC 0     10.9      9.6       7.0
```

Each row gives the data for an individual patient, identified by a code `AA`, `AB` etc. The Response columns are *repeated measurements* of the same underlying variable (e.g. blood pressure), rather than measurements of different variables (e.g. blood pressure, height and weight). For analysis, it may be more convenient to rearrange these data into 3 columns: `ID`, `Group` and `Response` (with 3 rows for each patient).

One way to do this in R is via the reshape command. The basic idea is that data such as those above can be represented either in a *wide* table or a *long* table. In this example, we have a wide table. Suppose the data have been read into a data frame called `wide.data`. To create the corresponding long data frame:

```
> long.data <- reshape (wide.data,direction="long" ,idvar="ID" ,
+ varying=list (names (wide.data)[3 :5]))
```

The `varying` argument is used to specify the columns that correspond to repeated measurements (here the 3rd to 5th). The result is a data frame containing columns `ID`, `Group`, `time` and `Responsel`. The `time` variable takes the values 1, 2 or 3 and indicates, for each row, the corresponding column (`Responsel`, `Response2` or `Response3`) of the original data frame.

The `reshape` command can also be used to convert a 'long' data frame into a 'wide' one. If you find yourself baffled by the syntax, you may like to consider the more basic basic but longer approach of creating vectors by stacking columns of `wide.data` on top of each other (the `stack` command does this), and then joining all these vectors together into a single data frame. In this case you need to know how to join vectors (and data frames) together ...

2.7 Combining data frames

The basic commands are `cbind` ('bind two objects together column-wise' i.e. create a single object by placing them side by side) and `rbind` ('bind two objects together row-wise' i.e. create a single object by placing one above the other). With `cbind`, the objects must have the same number of rows; with `rbind` they must have the same number of columns. An alternative to the `reshape` command above is:

```
> responses <- stack(wide.data[,3:5]) # Stack 'Response' columns
> patients <- rep(wide.data$ID,3) # Replicate patient IDs 3 times
> groups <- rep(wide.data$Group,3) # Ditto patient groups
> long.data <- cbind(patients,groups,responses) # Tie them all together
```

`cbind` and `rbind` are very useful, but can only be used to join objects that have the same number of (respectively) rows or columns.

Another thing you might want to do is to *merge* two data frames, based on variables that they both share. For example, in our clinical trial we may have some patient information that isn't present in the data frame `long.data`, but is stored in another data frame with a single row for each patient. The data frames could be combined using the `merge` command, or `inner_join` using `dplyr`. I'll leave examples of this until another time.

2.8 Ordering, ranking and sorting

`sort`: sorts a vector into ascending or descending order.

`order`: if x is a vector, order (x) gives the order in which you'd have to write the elements to sort them. For example, to sort the sequence 5, 12, 7, 6, 2, 3 you'd take the 5th, 6th, 1st, 4th, 3rd and then 2nd values.

`rank`: ranks the elements of a numeric vector. For the 5, 12, 7, 6, 2, 3 example the ranks are 3, 6, 5, 4, 1, 2 (i.e. 2 is the smallest value so it gets a rank of 1; 3 is the next smallest so it gets a rank of 2 and so on). Ranks are often used in non-parametric test procedures. Don't confuse `rank` with `order`!

Exercise: what does `test.data[order(test.data$Response),]` do?
Think about it and try and work it out, before typing it to see if you're right.

There is an easier way to do this using a `dplyr` command `arrange`:

```
> arrange(test.data,Response)
```

2.9 Data Wrangling Exercise: US President Data

You have downloaded the dataset `presidents`, which contains the data on presidents of the USA.

Enter the date using the `load()` command.
Inspect the data in the data frame.
Notice that not all US presidents are in the dataset.
Check that you understand all the variables.
What is the problem with using the name variable to identify the presidents?
Alter the name variable so that this problem is dealt with.

The variables `start`, `end` and `DOB` have class `Date`. This means that they are stored internally as days since 1st January 1970, but are output on the screen in a useful format. The time in days between two `Date` objects can be calculated by subtracting one from the other.

Use the methods above to do the following tasks. You can choose if you want to use the traditional commands or the modern `dplyr` commands.

(a) Produce a frequency table of `Party`

(b) Obtain a data frame containing just the Republican presidents

(c) Add two new variables to the original data frame, one containing the length of the presidency and one containing the age at inauguration.

(d) Sort the data frame according to age at inauguration.

(e) Create a data frame with two rows with the total length of years in office for the Republican party and for the Democrat party.

3 Introduction to Hypothesis Tests

In the short presentation at the beginning of the workshop there was an example of a so called t-test. This is a example of a *hypothesis test*, which occur everywhere in data analysis and statistics. A hypothesis test is a formalised way of making a decision between two options, such as "is Drug A better than Drug B" or "does mean global temperature have an influence on storm strength in the Caribbean."

You will learn the theory of hypothesis testing in Mathematical Modelling. But the broad principles of a hypothesis test are:

- Specify two hypothesis, exactly one of which can and must be correct.
- The *null hypothesis* has specific statistical properties which we can model.
- The second hypothesis is called *alternative hypothesis*.
- We compare our data with the what we expect if the *null hypothesis* were true. If our data are very "unexpected" we don't believe the null hypothesis, we reject it and "find for" the alternative hypothesis.
- If we don't reject the null hypothesis, either the *null is true or it is untrue but we have not collected enough evidence yet to reject it*.

Because we are analysing data, which has some level of randomness, we can never be totally sure that we come to the right conclusion.

Today we will just concentrate on one "test of location", called the t-Test, but consider two variants of it.

3.1 One sample t-test

In the talk you saw some hypothesis tests based on simulation. The scenario is:
A sample of 20 Berlin residents is obtained and the heights in cm of each person is measured. The population mean is assumed to be 170 cm.

The two hypotheses are:

Null hypothesis H_0	The mean height of Berliners is 170cm
Alternative hypothesis H_1	The mean height of Berliners is not 170cm

The code for the first test is given below. Because the `rnorm` command gives a different sample of random numbers each time, your results will be different from in the talk. This is equivalent to you taking a survey of different Berliners than the Lecturer did.
Try it out.

```
> x<-rnorm(20,170,12)
> x
> summary(x)
> plot(x)
> abline(h=mean(x)) #observed sample mean
> abline(h=170,col=2) #true population mean
> t.test(x,mu=170)
```

Ignoring all the background information, look directly at the p-value. Do you have a p-value under 0.05? If yes then you were unlucky! When $p < 0.05$ then we no longer believe the null hypothesis and we reject it. This example is controlled so that the null *is* true and the chances that we will reject the null is one in twenty (when using the standard threshold of $p = 0.05$). With simulated data we know what the truth is. In practise we do not know the truth.

Repeat the above commands. Do you have a p-value under 0.05? It is extremely unlikely that you had bad luck twice.

Now alter the population mean from 170 to 180 in the `rnorm` command. This would be equivalent to you taking sample of 20 players in the Berlin Adlers (an American football club in Wedding). You probably, but not definitely, now have a p-value under 0.05 which means you reject that the sample comes from a population with mean 170.

Play around with the `rnorm` command, choosing your own values of the sample mean (170) the standard deviation (12) and sample size (20). A larger sample size means that you are more likely to reject the null hypothesis when the population mean is not 170.

3.2 Two sample t-test

In the last subsection we assumed that the population mean (average height of all Berlin residents) is equal to 170 cm. In practice we rarely know what the population mean is. A more likely scenario is that we are interested in the heights of the Adler players and compare them with a reference sample taken from the general Berliner population.

The two hypotheses are:

Null hypothesis H_0	The mean height of the Adler players is equal to the mean height of Berliners
Alternative hypothesis H_1	The mean height of the Adler players is not equal to the mean height of Berliners

```
> Berliner<-rnorm(20,170,12)
> Adler<-rnorm(20,180,12)
> plot(Berliner,ylim=range(c(Berliner,Adler)),pch=16)
> points(Adler,col=2,pch=16)
> abline(h=mean(Berliner))
> abline(h=mean(Adler),col=2)
> t.test(Berliner,Adler)
```

A p-Value under 0.05 means that we reject the null hypothesis, and conclude that the sample mean of the two populations are different.

There's an alternative syntax for `t.test` that allows you to compare 2 groups of measurements stored in a single variable in a data frame. For example, in our `test.data` example above we might want to compare the values of Response for the treatment and control groups.

```
t.test(Response ~ Treatment,data=test.data,var.equal=TRUE)
```

Notice the use of the tilde `~` notation.

3.3 T-Test: Exercise

In the main Exercise (Section 4.1) from Workshop 3 you started to analyse the `medicament` dataset. We will now use this for a hypothesis Test

- Re-load this dataset `medication<-read.csv(file="Medication.csv")`
- Last week you used the traditional method of renaming the column names of this data frame using `names()<-`. There is a `dplyr` function called rename. This command is much nicer when the data frame has many columns and only one or two need to be renamed

  ```
  medication<-rename(medication,Time=Zeit,Drug_Type=Medikament)
  ```

- Re-create a box plot for `Time` grouped by `Drug_Type`. Does it seem likely that the mean time to blood clotting is the same in both groups?
- Write the `t.test` command to test whether the mean time to blood clotting is the same/different in both groups. What is your conclusion

4 Tidying up

► Make sure your script file has sensible comments.

► Save the script file (source file) again: Strg + S or *File > Save as*.

► Leave RStudio by typing the command:

```
> q()
```

When R asks you *Save workspace image ...?*, click on **Don't save**!

► Feierabend!