INST0065 Practical – Week 3

Plotting data

This week, we will start by looking at the plot() command, which is part of the base R installation. There are a variety of alternative plotting functions (and associated libraries) and we shall look later on at one of these – ggplot().

Later on we will look at data import, but to start with, we shall focus on *anscombe* - one of the tables in the datasets library. This contains four pairs of x,y co-ordinates labelled x1,x2,x3,x4 and y1,y2,y3,y4.

Task

1. Check the anscombe data exists:

```
> anscombe
```

   You should see a listing of the data frame that starts

```
   x1 x2 x3 x4    y1   y2    y3    y4
1  10 10 10  8  8.04 9.14  7.46  6.58
2   8  8  8  8  6.95 8.14  6.77  5.76
```

   If you get an error stating that anscombe is not found (I don't expect this to be the case), please try:

```
> install.packages("datasets")
> library(datasets)
```

2. The functions mean(), var() and sd() respectively report the arithmetic mean, variance and sample standard deviation, e.g.

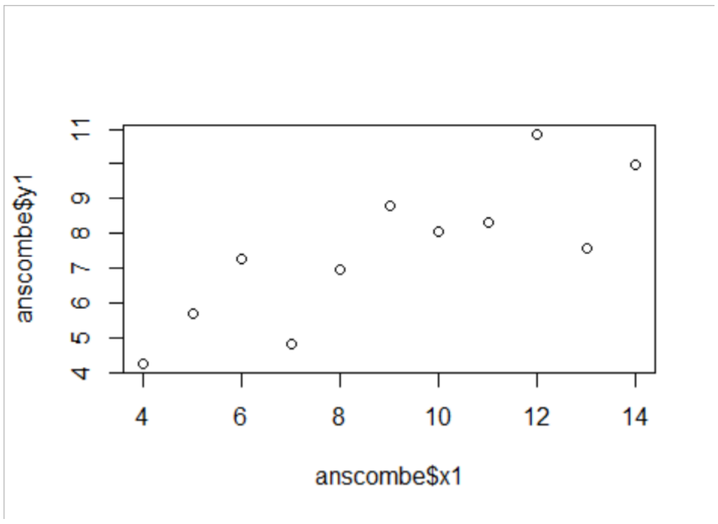
```
> mean(anscombe$x1)
[1] 9
```

3. Use these commands to test all of the components of the Anscombe data. How similar are the x variables? How similar are the y variables?

We can plot these data using plot(); the simplest way of doing this is simply to nominate x and y variables. We can do this as plot(x1,y1) for example (that is, R expects the first argument to be the x variable, and the second argument to be the y variable), but we can present this more formally as:

```
> plot(x=anscombe$x1,anscombe$y1)
```

This will generate a basic graph looking like this:



This is a representation of the data, but is not very attractive.

Information about the plot() function is here:
https://www.rdocumentation.org/packages/graphics/versions/3.6.2/topics/plot

One issue is that the axes only cover the extent of our data. We might want to show a fuller image. The xlim and ylim options allow us to define the axis limits for the x and y axes directly. Both require a vector of two numbers giving the start and end of the required range.

4. Try adding an xlim:

```
> plot(x=anscombe$x1,y=anscombe$y1,xlim=c(1,10))
```

   Is this the range you wanted? Try different ranges. Do you want a range that would accommodate all four Anscombe sets?

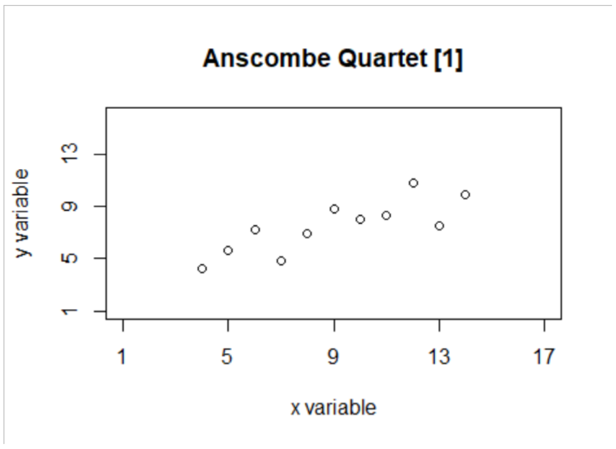
   Now, add a ylim parameter as well.

5. Now, you might note that the axis labels aren't very friendly. We can add parameters to the plot for axis labels, a main label and a subtitle. All of these take strings. Use main, xlab and ylab to set labels, e.g. plot(…. main="Anscombe plot")

6. You might also say that the axis tick marks aren't ideal either. We can adjust these using xaxp and yaxp; both require a vector giving the extreme values (i.e. first and last) and the number of gaps between tick labels.

   Update your plot command to use xaxp and yaxp. What are the end points you will use? Do the gaps nest nicely?

You should now have a plot that looks something like this:



We might still want to improve on this, and will look at three further aspects: the marker appearance, size, and colour.

Marker appearance can be set with the parameter ‘pch’, which sets the plot character. A list of options for points is shown at:
http://www.sthda.com/english/wiki/r-plot-pch-symbols-the-different-point-shapes-available-in-r

Option 16 is widely used – it is a solid circle.

The marker size can set with cex=(value).

7. Update your plot adding in, to start with, pch=16 and cex=1. What effect do these have? Try altering cex: what are the range of values you think it might take? Try different settings for pch; do you have a preferred one.

   Both pch and cex can take a (multi-value) vector as their value, in which case they will cycle through each option. Try doing this with both the Anscombe data and – when you get to it – the sine wave example below.

   Can you think of a case where alternating symbols may be useful?

Finally, we can turn to colour. The colour of markers can be set with ‘col’. For some markers both (line) colour and fill colour can be set; the latter is set with ‘bg’. Many colours are named in R, and can be set directly, e.g. col="red". Colours can also be set using a variety of schemes, including hexadecimal red-green-blue values, for example col="#00AAEE". A variety of colour palette tools will give you these hex values.

For a set of recognised colours see: https://www.r-graph-gallery.com/42-colors-names.html

8. Choose a colour, and set your markers to that colour.

Type of graph

The plot function can produce different types of plot. So far we have only looked at basic scatter plots. To illustrate this, we can use a simple graph of a trigonometric function.

9. Produce a basic sequence:

```
> x <- -10:10
```

10. Now, plot x and sin(x) (or a similar trig function of your choice)

```
> plot(x=x,y=sin(x))
```

11. Now, add in a graph *type*:

```
> plot(x=x,y=sin(x),type='l')
```

Type 'l' is a line graph, rather than one using points. A set of types is given in the documentation linked above

12. Try the different types. What effect do they have? The original graph as shown may seem a bit coarsely drawn. Use the seq() function (look this up, or refer to last week's notes) to generate a sequence that is finer grain than -10:10. What effect does this have on the graph?

Plots in R can be thought of us as being like they were drawn on paper. We can add things to the plot, but once things are committed to 'ink', we can't remove them (without starting afresh).

As an example of adding an element to a plot, we will add a regression line to the Anscombe data. We will use the linear model function lm(), but will not worry too much about how it works. If you have done statistics previously you will probably have carried out a linear regression. You will probably have calculated equations of lines in maths at school.

13. Repeat the Anscombe graph you produced earlier. You should find that the arrow keys on your keyboard will allow you to step back to a previous command, rather than have to type it.

14. Run the following command (if you have used one of the other Anscombe data sets, then substitute the appropriate column names).

```
> lm(anscombe$y1 ~ anscombe$x1)
```

This will carry out a regression, and show you the parameters (intercept and slope) of a line that predicts y for a given value of x. We can use the parameters to add a line to our plot. The function abline() allows us to add a line to plot (the plot we have just drawn). We can simply use the output of lm() to do this.

15. Add a line to the plot using:

```
> abline(lm(anscombe$y1~anscombe$x1))
```

   a. Try adding a col= statement to the abline command.
   b. Look at the documentation for graphical parameters: https://www.rdocumentation.org/packages/graphics/versions/3.6.2/topics/par

Find the possible values for linetype (lty) and experiment with different line types. You may need to redraw the original plot.

Data import

So far we have used built in datasets, and example vectors etc generated from sequences. For any data processing system, what we really want to do is load and use our own data.

When loading data, we typically need to clean the data (remove incorrect entries etc), and often to recode the data.

R contains a number of functions for reading and writing data; RStudio also supplies helper facilities that make it easier to load data. We shall start by looking at the R functions.

Loading data using R

In order to read in a data file, we need first to make sure that R's *working directory* is correct; that is, it is the same directory in which the file that we want to read (or the location to which we want to write output).

The command getwd() returns the curret working directory, for example:

```
> getwd()
[1] "C:/Users/oliverdw/Documents"
```

The command dir() returns a directory listing of files in the working directory:

```
> dir()
 [1] "inst0065-2020-21-3a.mp4"
 [2] "inst0065-2020-21-3a.vtt"
 [3] "inst0065-2020-21-3b.mp4"
 [4] "inst0065-2020-21-3b.vtt"
```

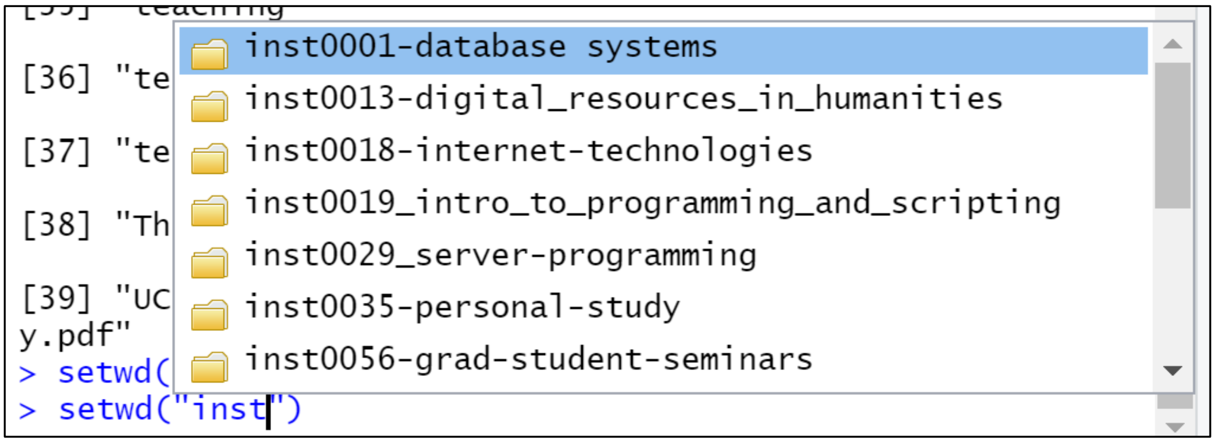
The command setwd() allows us to change the current working directory, and thus to make sure we are in the correct directory.

```
> setwd("N:/work/teaching/")
```

Note that if setwd() is successful, it will not show any feedback, you will simply be returned to the command prompt. If there is a problem, you will be shown an error message. You do not need to type the while directory path in one go – the path is (unless preceded by a root or drive letter) relative to the current directory – so one can change directory in a series of steps to get to the intended location.

Important: if you are using Windows, you will need to use forward slashes (/), as above, rather backslashes (\) as normally used in Windows filepaths. This is because '\' is interpreted as an escape character[1] rather than interpreted literally. If you want to use backslashes they must be escaped – i.e. the path shown above would be: N:\\work\\teaching.

If you are using RStudio, pressing the tab key whilst typing a directory (or file) name will offer a set of suggested completions, as shown below; the required option can be selected with the arrow keys and then the return key.



```
[36] "te
[37] "te
[38] "Th
[39] "UC
y.pdf"
> setwd(
> setwd("inst")
```

```
inst0001-database systems
inst0013-digital_resources_in_humanities
inst0018-internet-technologies
inst0019_intro_to_programming_and_scripting
inst0029_server-programming
inst0035-personal-study
inst0056-grad-student-seminars
```

Once you are in the right directory, a file can be read with the command read.csv(). As its name suggests, this can be used for reading comma separated values.

The simplest approach is simply to give the filename:

```
mydf <- read.csv('mydata.csv')
```

Note that read.csv() on its own will simply read a data file, it will not do anything further with the data. In order to use it, we need to assign the output of read.csv() to an object, as shown in the example above (we use '<-' to assign it to the object 'mydf', a data frame.

By default, R will assume that the first row is a header and contains variable names. We can leave this as default, but it can be easier to remember that this is occuring if we use the header option:

```
mydf <- read.csv('mydata.csv',header=TRUE)
```

This has the same result as before, but can be useful when we come to read data that does *not* have a header – regularly including the setting will remind us that this can be adjusted.

More significant is the as.is option; by default R will turn character strings into a modified data form known as *factors*. However, this is often not what we want to do, and most guides suggest that we should try to suppress this behaviour, using the as.is option. This will cause strings to be left as strings.

See, for example: https://www.dummies.com/programming/r/how-to-use-read-csv-to-import-data-in-r/

So, we can read the data more reliably as:

```
mydf <- read.csv('mydata.csv',header=TRUE, as.is=TRUE, sep=',')
```

[1] The next character has some special meaning – for example \n usually refers to a newline.

Note that here we have included the ‘sep’ option. As with header, the option we are using (a comma) is the default behaviour – but this reminds us that we could use a different separator if wanted to. As described above, we have also included header=TRUE (a default setting) to say that we’ll use the first row as labels, and as.is=TRUE to ensure that strings are simply read as strings.

If you are using RStudio, you’ll find that there is also an ‘Import dataset’ tool, under the ‘File’ menu. This ultimately uses the same R commands to read data, but make it much easier to find the file that you want, and update appropriate import settings. You will also notice that it is possible to import from Excel and other sources. Again, these options trigger relevant R commands.

For the practical, we are going to import a pre-prepared data set, taken from the web site ‘Box Office Mojo’. This provides ranked lists of film revenue, using various criteria. We are going to use a table listing all time gross, adjusted to take inflation into account.

https://www.boxofficemojo.com/chart/top_lifetime_gross_adjusted/?adjust_gross_to=2020

A dataset has been prepared [moodle link] that shows the top 200 films adjusted to 2020 prices. Note that the web page above contains a note explaining how calculations have been made. There is inevitably some difficulty in making like for like comparisons, but it is clear that if we are using unadjusted monetary values, then any comparison between contemporary and older films would be meaningless. Of course, in recent years, ‘box offie’ takes and viewing figures are affected by streaming technologies. Companies such as Netflix do not routinely publish viewership data, making it increasingly hard to compare films in this way.

The data as presented on screen are well structured and easy to copy and paste into an application such as Excel, with limited further cleaning needed. Additional rows were deleted, and then the file was saved in CSV format.

Tasks

16. Download the boxoffice file from moodle.
17. Use the R commands getwd(),setwd() and dir() to navigate to the directory where you have saved this file.
18. Read the data in using:

```
> boxoffice <- read.csv('boxoffice-data.csv',header=TRUE,as.is=TRUE,sep=',')
```

19. You should now do some explorations of the data, using commands described in the lecture and in supporting documents. Note that the films are ranked from 1-100. The table will not be re-ordered, unless you proactively take action to do so[2].

    a. Show a list of the top 10 films
    b. Show a list of film rank,title and adjusted lifetime gross, in reverse order, from 20 to 11

[2] Note that this is the same as handling data in a spreadsheet – we can edit data, but that will not affect row order. In contrast, if were using an SQL database (and we might see this as analogous, as R may be used to process and ‘manage’ large data files), changing any value might affect the order in which rows are listed: SQL makes no guarantees about row order, unless an explicit ‘ORDER BY’ clause is used.

c. Create a logical index (see last week's practical notes if needed) based on the boxoffice table, which is TRUE when the value of 'Year' is prior to 1960. Remember, you can refer to columns within a data frame using the dollar notation: boxoffice$Year
d. Use that logical index to list the details of the films in the all time list that were released prior to 1960
e. Use the notes from week 2 on pattern selection using grep to select films for which the title:
    i. Matches the pattern "Star Wars" (anywhere in the title)
    ii. Has a number at the end of the title (requires knowledge of regular expressions!]
f. Use the sum() function to calculate the total Rank value of all films you found in (d)
g. Follow the same processes as used in steps (c) and (d) to find the total Rank of films in the top 200 released in the 1970s.
h. Try calculating the sum of adjusted gross for the top 5 films. What happens?

When we try to calculate a value in (h), we find that we get an error message. This is because the values shown in the lifetime and adjusted gross columns are strings. We cannot treat these directly as numbers, even though it is clear to us (as humans) that that is what the strings contain.

First, we need to covert these strings into numbers. In order to do so, we need to remove both the $ currency symbol, and the commas. We can do this using the gsub function.

The function gsub() allows us to replace substrings that match a given pattern (using regular expression notation). Some description of gsub() can be found here:

https://bookdown.org/rdpeng/rprogdatascience/regular-expressions.html#sub-and-gsub

Some background information on regular expressions may be found here:

[to be added]

If we want to remove both the $ symbol and the , symbols, we can do so with the following pattern.

```
> gsub("[$,]","",boxoffice$Adj..Lifetime.Gross)
```

Here, we supply three arguments: the pattern we are looking for, the replacement text, and the data on which we're operating. To make this more explicit, we could name our arguments as in other examples:

```
> gsub(pattern="[$,]",replacement="",x=boxoffice$Adj..Lifetime.Gross)
```

The pattern we are looking for is [$,]. This will replace any character in the group [$,] (i.e. a dollar or a comma) with the nominated replacement text, which in this case is an empty string. R offers two similar functions, sub() and gsub(); the former will replace only the first match it finds, the latter will replace all matches – it is therefore equivalent to adding the 'g' (global) suffix flag after a standard regular expression. The parameter x identifies the data we will use, in this case the column "Adj..Lifetime.Gross" of the data frame boxoffice.

Running the command as above will return results that start:

```
  [1] "1895421694" "1668979715" "1335086324" "1329174791" "1270101626" "1227470000"
  [7] "1200856389" "1163149635" "1036314504" "1021330000" "1013038487" "936225101"
```

Note that these values are still strings, even though they only contain numeric characters.

We can convert the results to a numeric data type using as.numeric():

```
> as.numeric(gsub(pattern="[$,]", replacement="",x=boxoffice$Adj..Lifetime.Gross))
[1] 1895421694 1668979715 1335086324 1329174791 1270101626 1227470000
```

Note that the above commands are just writing their results to the console. We need to assign them to a new or existing column in or data frame in order to make progress.

```
> boxoffice$adj_usd <
as.numeric(gsub(pattern="[$,]",replacement="",x=boxoffice$Adj..Lifet
ime.Gross))
```

Note that in this command we are adding a new column to our data frame, called adj_usd; we have referred to this using the dollar notation. The '_usd' suffix in the name refers to US dollars, but note that this is just an artitrary name, it does not identify to R in any way that this is a currency amount.

Tasks

20. Using gsub() create a column in boxoffice with a numeric representation of the adjusted gross values
21. Create another column with numeric representation of the (unadjusted) lifetime gross
22. Can you create a column called 'decade', which gives, as text, the decade in which the film was released (e.g. "1970s")?

    Hint: for a typical solution you will need to extract a numeric value from boxoffice$Year, such that 1970, 1971, 1972 … 1979, all give you the result '1970'. How will you do this?

    You will then need to add on the character 's' at the end.

    The operators and functions needed to do this were described in (separate sections of) last week's practical

Plotting the film gross data

Having prepared our data, we are finally ready to plot some of this data. We wil look mostly at ggplot() in this section, but let us start off with plot()

23. Let us think about the distribution of values for adjusted gross. Do you expect to drop off quickly, or more steadily?

    How can you arrange a plot to show the rate at which gross declines from highest to lowest (within the top 200)? What variables are needed on the x and y axes?

    Use plot() to create a suitable graph

24. Are the labels for gross as you would expect? A simple approach would be show the grosses in millions of dollars rather than in dollars.

    How can you do this? Remember to set the axis label appropriately.

We will now try plotting the same data using ggplot().

25. As described in the lecture, we must set this up.
    a. If you have not yet installed tidyverse then do so now (comment on Moodle if you have any problems installing this):
       ```
       > install.packages("tidyverse")
       ```
    b. Load the library:
       ```
       > library(tidyverse)
       ```

The ggplot() function is described in R for Data Science; see: https://r4ds.had.co.nz/data-visualisation.html (NB in the printed edition, ggplot is introduced in chapter 1; in the online edition this material is in chapter 3).

26. We will start by returning to the Anscombe data for a moment. In order to use ggplot we need to identify a data source, and then add one or more layers. Different sorts of layers exist, we are going to use a point layer. For each layer we need to create a *mapping*, defined by an *aesthetic*.

    ```
    > ggplot(anscombe) + geom_point(mapping=aes(x=x1,y=y1))
    ```

    This produces a basic plot as before, but a little more attractive. Again we can adjust the axis limits if we want to. The ggplot function uses xlim and ylim – similar parameter names to those used before, but it should be remembered that these are parts of a different command. These versions take two values, the start and end points. They are added as separate layers:

    ```
    > ggplot(anscombe) +
    geom_point(mapping=aes(x=x1,y=y1))+xlim(c(1,16))
    ```

    Similarly, we can alter the axis labels with xlab() and ylab(); in their simplest form we just need to supply a string:

    ```
    > ggplot(anscombe) +
    geom_point(mapping=aes(x=x1,y=y1))+xlim(c(1,16))+xlab("x")
    ```

We will now plot some of the film grosses data using ggplot

27. Use ggplot to produce a plot with lifetime gross as the y variable, and year released as the x variable. As decscribed above, you will need to use a numeric version of lifetime gross.

28. Use ggplot to produce a plot of adjusted gross against rank

29. Use ggplot to produce a plot of adjusted gross against year.

In each case, you should use appropriate functons to set suitable labels.

The last technqiue we shall look at today is the use markers to show further data. The ggplot function allows us to adjust characteristics such as marker size or colour on the basis of a variable.

30. Add the parameter 'colour' to your mapping, for example:

```
> ggplot(boxoffice)
+geom_point(mapping=aes(x=Year,y=adj_usd/1000000,color=decade))
```

This will adjust colour on the basis of the column 'decade' (set in one of the exercises above). Given that one of the axes is already showing year, is colouring by decade helpful?

Compare the results with these two plots:

```
> ggplot(boxoffice)
+geom_point(mapping=aes(x=Rank,y=adj_usd/1000000,color=decade))

> ggplot(boxoffice)
+geom_point(mapping=aes(x=Rank,y=adj_usd/1000000,color=Year))
```

How does this change the appearance? Can you explain the differences in the presentation of the legend?

Finishing up

When you exit RStudio, you will be asked if you want to save your workspace. You should choose 'save' if you want to keep the data frames etc that you have created above. If not, they will cease to exist when you quit and would need to be created again in order to do further work.

We have only touched the surface of plot() anf ggplot() this week; please feel free to continue to look at ways that these plots can be adjusted and fine-tuned.