**M2**

**R Scripts, Environments & Workspace**

**Setting a working directory**

Q: I cannot get the setwd("C:/Users/Rcourse") command to work. Below are the commands and error messages:

> setwd("C:/Users/Rcourse")

Error in setwd("C:/Users/Rcourse") : cannot change working directory

A: Sometimes R gives error messages that are less informative than one might wish. The error "cannot change working directory" would be more useful if it read instead "the folder does not exist", because that is the reason R cannot change the working directory – you are trying to change it to a folder that doesn't exist. In order for it to work you can either:

(1) create the folder “C:/Users/Rcourse” and then run the setwd() command again; or

(2) pick an folder that already exists and use that in the setwd() command

Be careful because capitalization is important in R, so make sure you are spelling and capitalizing the folder names correctly.

**M2**

**R Scripts, Environments & Workspace**

**Setting a working directory**

Q: I ran the setwd("C:/Users/Rcourse") command but nothing happened, the folder is empty.

A: It only looks like nothing happened, but R has now changed the working directory to "C:/Users/Rcourse". That doesn't mean anything should show up in this folder though. What it means is simply that if you try for example to save something, R will automatically save into this folder. Also if you try to open a file, R will first look in this folder. So the code worked fine, the only problem is that it didn't do anything obvious. You can check this by using the companion function getwd(). Run this command (with empty parenthesis) and you will get the current working directory. Start a new R session (go to Session>Restart R) and then try the following:

getwd()

setwd("C:/Users/Rcourse")

getwd()

This way you can see how the working directory has changed after the second line.

**M4**

**Bar plots**

**Creating bar plots**

Q: I am creating bar plots on my laptop. They correspond with the graphs in the course but appear far smaller and less clear. How can I make them look clearer and more professional?

A: The underlying issue is that when you instruct R to produce a plot, it doesn't really know where this plot will be rendered (this is called the graphics device). In most cases, it is being rendered in the RStudio plot window. The size of this varies with your screen size, which can be an issue if you are working on a small laptop. The size will also vary with the size of the panes within RStudio, which you can stretch out or make smaller.

So first of all, you need to decide where you want your final plot to end up. The RStudio plotting window is great for quick and unrefined plots, to give you an impression of the data etc., but when you want presentation standard charts, you will want to output some other way. So, before you make all the detailed changes to the position of the legend and the font size etc., you need to decide that. You have plenty of options, which you can look up in the help if you look up "device". There you will see listed options such as pdf, png, jpg etc.

Once you pick one depending on your requirements, you also have to decide the size you want, which will affect the resolution as well (unless you pick one of the vector formats, such as pdf or postscript). And only once you have both of these set, does it make sense to do precise formatting.

Here’s an example of how this works:

1. first you open a graphical device by giving it a filename (it will be saved in your working directory unless you specify otherwise) and the size of the device (this will affect the resolution in a format like png):

png("lending-type-barplot.png", width = 560, height = 320)

2. manipulating the margins is very useful, especially if you want to position the legend outside the plotting area

par(mar=c(4, 2.6, 3, 1))

3. now you draw your plot

barplot(lending\_income, main="Stacked Bar Plot",

xlab="Type of Lending", ylab="Frequency",

col = c("blue", "green", "yellow", "red", "purple", "grey"), beside=FALSE, cex.names = 0.8,

legend = row.names(lending\_income),

args.legend = listxjust=4.3, yjust=--0.9, cex=0.7, bty='n'))

4. finally, don't forget to close the device once you are done plotting to it.

dev.off()

5. Now you can open the image file and check whether you are happy with it, otherwise go through the steps above again.

The RStudio functionality that lets you save the chart from the plotting window also has some of this functionality: you can change the size there, and click update preview to see what it looks like. But this is not very reproducible, so it is recommended that you script the whole thing as shown in the example, by explicitly calling png() yourself, which will also save you time in the long run.

**M4**

**Examining the internal structure of an R object**

**Accessing the data set**

Q: I am not able to install one of the packages:

> library(devtools)

> devtools::install\_github("tdhock/animint")

# Enter one or more numbers separated by spaces, or an empty line to cancel

1: 1

Rcpp (0.12.19 -> 1.0.0) [CRAN] I

nstalling 1 packages: Rcpp Installing package into ‘C:/Users/…/…/…/…/…’ (as ‘lib’ is unspecified) trying URL 'https://cran.rstudio.com/bin/windows/contrib/3.5/Rcpp\_1.0.0.zip'

Content type 'application/zip' length 4479288 bytes (4.3 MB) downloaded 4.3 MB

…

package ‘Rcpp’ successfully unpacked and MD5 sums checked

Error: (converted from warning) cannot remove prior installation of package ‘Rcpp’

A: The reason for this error is probably that the package that is causing the problem (Rcpp) is currently loaded, that’s why it cannot be updated. Try this:

1. Restart your R session - on the menu this is under Session/Restart R.

2. Then run library(devtools).

3. Now run devtools::install\_github("tdhock/animint").

4. If this doesn’t work, repeat steps 1.-3. BUT this time, when you are asked which of the packages you would like to replace, select the last option, the one which says "None"

Whenever you now re-run this script you can comment out (or delete) the line devtools::install\_github(), since you already have the package installed and only need to load it with library().

**M4**

**Scatter plot**

**Scatter plot using car**

Q: What is shown by this scatter plot; what can we learn from it?

A: Perhaps most obviously this chart can be contrasted with the chart produced without conditioning on the lending variable. You can do this by replacing the formula "fertility.rate~life.expectancy|lending" with "fertility.rate~life.expectancy" which treats all the data points together. If you look at the help documentation for the car::scatterplot() function you can also figure out what the lines mean: regLine=TRUE controls the drawing of a fitted OLS regression line - these are the full straight lines that are drawn on both plots.

If you use a grouping variable, a separate regression is estimated for each group. In your example this allows you to see that the slopes vary quite significantly between the four groups except for the countries with "blend" and "IBRD" lending types which overlap almost perfectly. Compared to these, in the countries with "IDA" lending type fertility rates decrease at a much slower rate as life expectancy increases. On the other hand the countries where the lending type is down as "Not classified" (presumably countries that have not been receiving World Bank loans?) the fertility rates decrease at a much faster rate as life expectancy increases (the slope of the regression line is much steeper).

The argument smooth=TRUE controls the drawing of a LOESS or LOWESS line = locally estimated scatterplot smoothing. In the un-grouped data a smooth line is drawn both for the mean and the variance, while in the grouped data only the mean lines are drawn. This is a non-parametric strategy i.e. you are not assuming the data fits any particular distribution, just finding the best line of fit. Here you can e.g. see that "blend" and "IBRD" lending types are quite distinct in their shapes with "blend" having a significant bulge in the middle range of life expectancies.

I hope this is helpful and gives you some points of departure for researching this further. As this is an introduction to R course and not an introduction to statistics course I have tried to point you to how the R documentation can be used to find information about what you are doing, which you can then explore further concepts such as OLS regression or LOESS lines in your preferred statistical textbook or online.

**M5**

**Loading the data set**

**Loading the data set**

Q: I cannot load the GSS dataset and I receive this error message: "labels must be unique". I have downloaded the packages as instructed and loaded the haven library.

A: Try to import the data using the read.dta() command from the foreign library. In order to do so:

# first make sure you have the foreign package installed

install.packages("foreign")

# then load the package

library(foreign)

# now import the file (make sure the path is correct if it is not in the working directory)

gss14 <- read.dta("GSS2014.dta")

# because foreign::read.dta()is slightly different than haven::read\_dta(), let's make sure the output format is the same:

gss14 <- as\_tibble(gss14)

# now have a quick look at it: head(gss14, n = 6)

So here we are using another package to import the .dta file. Because it is a different function it treats the data slightly differently, so you additionally reformat the table into a tibble. That way, you should get a similar result to the one in the course materials. I cannot really say why this issue has cropped up, but it stems at least partly from the fact that we are using a non-commercial software (R) to import data created by proprietary software (such as SPSS, STATA and similar). So we are lucky to have several developers of packages writing functions to import the data, but it can sometimes mean we have to try several attempts based on a combination of operating system and versions of software.

**M6**

**Visualizing interactions**

**Publication quality tables**

Q: I ran the below code as instructed. R ran the code with no errors, but I cannot see the table. I had my working directory set, but the table is not there either.

htmlreg(list("Model 1" = reg1, "Model 2"=reg3), inner.rule = "\_",

outer.rule = "\_", column.spacing =1,

caption = "OLS Models Predicting Attitudes to Work", captcaption.above = TRUE, fontsize = "small",

center=TRUE, star.symbol = "\\\*", doctype = FALSE)

A: You are right, the code you have run does not explicitly produce any tables. But what it does do is produce the html code for a table. In the case of the lines you have run, there was no instruction to save the html code to a file, therefore R has simply printed out the code in the console (it starts with <table cellspacing="0" align="center" style="border: none;">.....). In the next command in the course materials you expand on this code, by customizing the names of the coefficients but, more importantly, there you also add the argument file = "table1.htm. So, after you run that command, you should see the file saved in your working directory. You can then open it in a browser to see the table (if you open int in a text editor you will of course see only the html code, same thing that was printed out in your console earlier.

**General**

**Arrays and data frames**

Q: What are the differences between arrays and data frames?

A: For the most part you will really only be working with data frames. Data frames are collections of vectors of the same length, so that each vector can be seen as a variable. Because each vector can only have one data type, this means each column (vector) in the data frame has to have its own type. But in a data frame you can have some columns that are, say, numeric and others that are factors or characters, which is useful, since that is the sort of mix of data we often deal with.

An array is basically just a single vector that has multiple dimensions. For example, a 2-dimensional array (which is really a matrix) is a vector that has a number or rows and a number of columns. You can also add more dimensions e.g. a 3-dimensional array is like a data cube, with layers of tables. But, don't forget, in essence it is really just a vector. This means you cannot mix data types, so an array can only be all numeric, or all character etc.

Most of the time arrays are really not something you will be working with, data frames are much more useful in statistics, unless you are multiplying matrices and doing more mathematical work. Below is a quick example.

# create a vector

vec <- c(1:16)

vec

# coerce the vector into a 2-d array

arr1 <- array(vec, dim = c(8,2))

arr1

# you can also change the array into a data frame

as.data.frame(arr1)

# coerce the vector into a 3-d array with 4 rows, 4 columns, and 2 layers

arr2 <- array(vec, dim = c(4,4,2))

arr2

# if you change the array into a dataframe, it will flatten the layers into a 2d table

as.data.frame(arr2)

**General**

**Capitalization**

Q: Is R case-sensitive i.e. does capitalization matter?

A: Yes, capitalization matters. In R e.g."Year\_started" and "year\_started" refer to two different variables. This is also important when it comes to file names and folder names, so make sure you check the capitalization carefully if R cannot find a file or folder.

**General**

**Commenting on code**

Q: How do I comment on code?

A: Start a line with the # symbol - R doesn't run the commands in the line after this symbol, but treats them as comments. You should use comments liberally, that way whenever you come back to your script you know what the purpose was of the code you wrote. Have a look at the course provided scripts for an idea of what is good practice. You will thank yourself later!

**General**

**Correcting errors in code**

Q: What should I do if I need to correct an error in my code?

A: You should fix the code in your script file and run it again. Your code should be treated as a sequence that has to be executed in the same order from the beginning of the script. In most cases, it is fine to just go one step back and, say, redefine an object because R just overwrites the one that was there before, and then continue with your script. But you should really make sure your code is always written in the sequence it needs to be executed from beginning to end. Never set yourself up to have to jump back and run a line and then skip forward again. The main aim is, when you are finished, to have a single file with all the steps of your analysis: loading the packages, importing the data, running the analysis and producing any plots or outputs.

**General**

**Error messages**

Q: What should I do if I see an error message?

A: R can sometimes be a bit opaque when it comes to errors and warnings, and it's completely natural when you're starting out to just want to shirk away from the red error message that looks super complicated. But actually, more often than not, there are useful clues in the wording, even if you cannot understand all of it, so do try to decipher them. And if that doesn't work, you can always try Googling them, that can lead you to an explanation quite quickly as well. If you ever need to ask for help, it’s really important to send your question along with the code you ran and the output you got in the console, as it makes it easier for the other person to see what you've done and what you see.

**General**

**Installing and loading packages**

Q: What is the difference between installing a package and loading it?

A: You install a package with the install.packages() command, and you only need to do this once on your computer. Then you can delete the line from your script, or add a # sign in front of it, so you can quickly un-comment it if you need it at some future date, e.g. when you are working on another computer.

You load a package with the library() command to be able to use it. You have to load the package anew every time you start a new session in R.

**General**

**Installing and loading packages**

Q: Where can I see which packages I have installed and loaded?

A: Once you have installed and loaded a package, you will see it listed in the menu below "Global Environment" in the “Environment” tab. You can also explore which packages you have installed and which ones are loaded via the "Packages" tab in the pane with the "Files" and "Plots" tabs. There you will see listed all the installed packages, and a tick box ticked in front of the ones loaded in this session. Only for the loaded ones can you access their functions. Alternatively, the command sessionInfo() will print out all the currently loaded packages.

**General**

**Loading packages**

Q: When you load a package, should you put it in quotation marks or not?

A: It does not matter whether you use quotation marks or not. The code works the same way either way. But it is good practice to be consistent in whatever you choose.

**General**

**Opening old scripts**

Q: Can I open up old R scripts and continue working in them?

A: Yes, the idea is that you work on a single script and add to it incrementally and iteratively until it contains your complete analysis and is self-standing. In the end you should be able to run the whole script from beginning to end. Of course while you are writing it you will probably just run sections of it: you can highlight whole sections of the script - e.g. as far as you got in your last session - and then hit Ctrl+Enter or hit the "Run" button on the top right of your script tab. And then continue working on the script from that point on. This should become second nature, you should feel comfortable starting a session from scratch as often as needed, especially if you think you might have made a mistake along the way. The final product is the script that can be run in a clean session. And a clean session means there are no packages loaded, there are no data or other objects loaded, everything is clean and new and nothing is remembered from the previous session. So everything you want R to do has to be in the script, and you should make sure you are not loading data from previous sessions (see the next question).

**General**

**Loading data from previous sessions**

Q: Can I use the data from a previous session?

A: You can, but you really shouldn’t. The default setup for RStudio is to always save your workspace (all the objects in your environment) into a file called .Rdata in your working directory, and every time you open a new session, to look if there is an .Rdata file in your working folder and load it. That way your previous workspace is always restored. This might seem like a good idea, but in practice can lead to a lot of problems, since there is no traceability of what is in the workspace, and how you have manipulated and changed those objects. Instead it is best practice to always start a session with a clean environment, and load the data you need from the raw data file.

In order to stop RStudio from automatically saving and restoring your old data, do the following:

Go to the menu Tools>Global Options and under General/Basic you should see the heading "Workspace". There make sure you **untick** the checkbox in front of "Restore .Rdata into workspace at startup" and underneath, make sure "Save workspace to .Rdata on exit" is set to "Never".