Spatial Analysis and Data Exploration in History and Archaeology, Spring 2021
LDA-H313

WEEK 1 EXERCISE: Introduction to plotting and mapping with R

Troubleshooting: R can be sensitive to formatting. Be careful if you copy-paste example script directly from this document. For example, look out for line breaks in the middle of a function, or quotation marks (“”, ‘’) may become wrongly formatted. In such a case, if you find that you are following the instructions exactly but nevertheless receive an error message it is worth double check this, and perhaps typing out the script yourself directly into the console. Also: R differentiates between upper and lower case letters.

When you are manipulating objects and you start getting persistent error messages, sometimes it is best to reload the object and start the workflow from the beginning. It may be that you have changed or edited the object wrong at some point, and it is quickest to reset.

Tip: You can use the up and down arrows to scroll through previous commands.

You can install R from https://cran.r-project.org. Select the correct OS and follow the instructions. Note: Mac users may have additionally to install XQuartz (see instructions under Mac OS X).

It is assumed that you have completed the Datacamp tutorials **Hello R** and **Getting Hooked on R** to gain some familiarity in using R. Note that the Datacamp tutorial uses the **RStudio** interface. This displays the R *script editor* and the R *console* as separate windows. Basic R software shows only one windows which combines the two, otherwise it is similar.

1. STARTING WITH R

Launch R after having installed it on your computer. You can see that it works just like the Datacamp tutorial. Next to the **>** symbol, write the following line and press enter:

```
1+2+3
```

You will get the result:

```
[1] 6
```

Now try creating an object with characters. Remember that you can create an object in the R workspace with the "left-pointing arrow", or **<–**. So:

```
hi <- "Hello world!"
```

Type **hi** and hit enter to see what is inside the object. You can also use functions such as **class** and **summary** to examine an object, such as with:

```
class(hi)
```

2. IMPORTING DATA IN R

2.1 You should have the Week 1 data package that came with these instructions. Take a look at the files it contains.

axes.xlxs is has a number of axe heads dated to the Iron Age and historical period that have been found by members of the public in Finland, mostly by metal-detecting.

If you open this excel sheet, you will find that it contains several columns. The *km_number* (or collections number / päänumero) column gives the archival collection ID within which the find is classified. *find_type* gives the type of object (here kirveet, or axe), length, width, thickness and weight give the measurement values of the individuals objects, and x and y columns give the coordinates of the findpost.

In order to carry out this excerise, you must save this data in a CSV (or Comma Separated Value) format. Go to File -> Save As and select **.csv** as the file format, saving the new file as **axes.csv** in the week1 folder.

monuments_fha.csv contains varous sites from the Finnish Heritage Agency's Muinaisjäännörekisteri (register of ancient monuments) that are dated either to the Stone Age or the Iron Age. This is already in a CSV file format, but if you Excel or another similar spreadsheet program installed you can open it and easily examine the contents.

Inside the sub-folder **finland** you will find **finland.shp** and other members of its shapefile package. Shapefiles are geospatial files developed by ESRI (the makers of the popular commercial GIS program ArcGIS) that store data in a vector format. We will be using them a lot.

2.2 Now you need to import some data to R.

Create a folder on your computer called **sade2021** place the week1 folder in it. The command immediately below, and in future weeks, assumes that you place it by you C:

drive (if on a Windows machine) or in your Documents folder (if on a Mac, note that slash is single and reversed), but you can create a different path to the folder.

Your work in R will be done in a working directory, and it is good idea to set this up right away. Point R to your new folder with the following command:

Windows:

```
setwd("C:\\sade2021\\week1")
```

Mac:

```
setwd("~/Documents/sade2021/week1")
```

Now you can start importing the data to your R workspace as objects. First, we will import the swords.csv file and save it as an object.

```
axes <- read.csv(file="axes.csv", header=TRUE, sep=",")
```

You can now examine the object and its contents. Type:

```
class(axes)
```

and you will see that the function **read.csv** saves tables as a data frame. Now type the name of the object and you will see the R print the contents of the object data frame. If you have a large data frame, you can use the following command to investigate the column names and the first few rows (by default the first 6):

```
head(axes)
```

2.3 Now we can do some statistical breakdowns of the data. Type:

```
summary(axes)
```

and you will get summary values of the on the columns, such as the minimum and maximum values, the mean and the median. The column find_type contains **nominal** values (the object term) and so for that row the function returns only the total count (there are 16 rows of “kirveet”). You can look into an individual column by specifying its name:

```
summary(axes$weight)
```

You can obtain the standard deviation:

```
sd(axes$weight)
```

But if you try this with the column thickness, you will get a NA result. This is because not all rows have a value entered in this column. In order to carry out this command for this rows that have values, you must removed the NAs. Try:

```
sd(axes$thickness, na.rm=T)
```

This **re**m**o**ves the **NA**s by specifying an additional argument in the function that sets the na.rm parameter to **T**RUE. You can find this by typing **?sd** and pressing enter, which calls up an R Documentation help window for that function.

2.4 Calling up plots.

R has a number of simply but powerful statistical and graphical functions built into the basic package. It is very easy, for instance, to create a scatter plot that compares an aspect of the size (here length) and the weight of these objects with each other using the **plot** function. Enter:

```
plot(x=axes$length, y=axes$weight, col="black", pch=16,
main="Axe heads recovered by members of the public")
```

This calls up a new Quartz windows displaying the scatter plot. Very unsurprisingly, there appears to be a correlation between the size and the weight of the objects. See what happens when you enter different colours to **col** and numbers to **pch**. Note: if you can't see the full plot or the x/y axis headings, try maximising the new window and then returning it back to the initial size. Alternatively, you can set the size of the margins yourself by first entering this:

```
par(mai=c(2,1.6,2,1.6))
```

Try this with different sets of four different small values separated by commas.

Another useful function is boxplot. Try:

```
boxplot(axes$length)
```

This is a graphical representation of the information you get from the **summary** function. Enter this for length, and compare the results with the boxplot. Now create a boxplot for width - note that the dot at the upper edge of the graph represents a statistical *outlier*.

You can add several boxplots into the same graphic, and lets a **lab**el to the **y**-axis. For example:

```
boxplot(axes$length, axes$width, ylab="millimeters")
```

But of course they must have same units of measurement. If you added weight (grams) to a graph that also displays length (millimeters), the comparison would be meaningless!

You can create a simple bar chart:

```
barplot(axes$weight, ylab="grams")
```

This is not terribly easy to read, though. Using the **sort** function, order the axes heads by weight value:

```
barplot(sort(axes$weight), ylab="grams")
```

And finally you can create a histogram.

```
hist(axes$weight)
```

The histogram automatically sets the bin size. You can toggle this with the break argument:

```
hist(axes$weight, breaks=seq(0, 1200, 100))
```

Here we place the **seq** function inside the **hist** function. This creates a vector of sequential numbers set as follows: the first value (here 0) sets the smallest value, the second sets the largest, and the third sets the interval.

A histogram may look a lot like a bar chart, but important to realise they are different. In short, bar charts are used to compare variables (such as weight values), and bar chart are used to show their distributions as ranges. If you call up a histogram and a barplot of axe head weights side by side and can study what this means.

```
dev.new(device=pdf, height=6, width=12)
par(mfrow=c(1,2), mai=c(1,1,1,1))
barplot(sort(axes$weight), ylab="grams", names.arg =
c(axes$km_number), las=3)
hist(axes$weight)
```

3 PLOTTING MAPS AND SPATIAL DATA

Finally, in preparation for next week's session, lets plot a few basic maps. First, basic R package does not have this functionality, and we must install additional libraries or "packages". Type the following, and select a mirror that is close by:

```
install.packages()
```

Then from the list select the packages "rgdal", and then repeat same process for "raster". You have now installed these packages onto your computer, which needs to be done only once, but for each new R session you must also load them. Do it by:

```
library(rgdal)
library(raster)
```

Now load the shapefile for Finland.

```
fin <- readOGR(dsn="finland",
layer="finland_valtakunta_simpler")
```

You can examine the object characteristics and plot it right away:

```
summary(fin)
plot(fin, col="lightgrey")
```

This is a shapefile polygon. The notation that R uses to describe a CRS (coordinate reference system) is called *project4string*; you can see that the UTM zone reference this polygon has is zone 35, which is indeed where Finland is located.

Now lets being in the archaeological monuments data. Load the CSV file and then examine it using the **summary** function.

```
monu <- read.csv(file="monuments_fha.csv", header=TRUE,
sep=",")
```

We indicate coordinate values from the relevant columns in the data frame:

```
coordinates(monu) <- ~X+Y
crs(monu) <- crs(proj4string(fin))
```

You can now plot the sites as points. Use **`?points`** and try toggling the values of the arguments to get different visualisations. Note: the **`points`** function does not call up a

graphic on its own, it only adds to a pre-exiting plot. Notice how at this large spatial scale we can see how the topography clearly influences the distribution of sites.

```
points(monu, pch=19, cex=0.1)
```

Lets try differentiating between sites from the Iron Age and the Stone Age. We can restrict the display to sites from just one period thus:

```
points(monu[monu$period == "stone age", ], pch=19,
cex=0.1, col="red")
```

Now do the same for the Iron Age sites, marking them with another colour. The distributions are strikingly different.

You can similarly examine the data by monument type categories and subcategories. You can take a quick peek at the terms (nominal values) in various columns using the **summary** function. Note the sheet is a bit messy with lots of loose commas before and after many terms, an artefact of it having been exported from a database. Try, for instance, “kuppikivi” (cup stone) in the subcategory column. Are the distributions truly reflective of past human activity… or perhaps more likely of the modern archaeological and institutional processes by which the data has been collected and recorded regionally?

Can you extract a specific selection of the data into a wholly new object? Look into the first page for a hint.

4 SAVING THE DATA

You can get a list everything in your current environment (here the objects you have loaded or created) by typing **`ls()`**. If you want to remove something from environment, you can do so with **`rm()`**.

There are various ways by which you can save the objects you have been working with. For instance, you can save a data frame as a CSV file:

```
write.csv(axes, file="axes_new.csv")
```

You can save a specific object as an RData file:

```
save(monu, file="week1_monuments.RData")
```

Or save everything in the workspace

```
save.image("week1_sade.RData")
```

In the future, if you want to load everything up again, you can do so by:

```
load("week1_sade.RData")
```

You have now finished with the Week 1 exercise.