STAT1003

Introduction to Data Science

Solution: Lab 3

Contents

Gapminder data from Lab 2

In the first part of this Lab, we'll continue analyzing the Gapminder data from Lab 2 to produce some plots. Please use the help file for the different functions we're going to be using.

The questions are similar to, but not exactly the same as, the last few questions of Lab 2.

```
# Load the gapmoinder data
library(gapminder)
```

1. Construct a plot of the average life expectancy **over the last four years** in each country against its average GDP per capita over that same time period, and label it appropriately.

As usual, there are lots of ways of doing this, some more efficient or elegant than others. In the first code chunk, we first create matrices consisting of life expectancy and GDP, and then extract the last four rows. Then, we can use the function `colMeans` to calculate the means of the last four years.

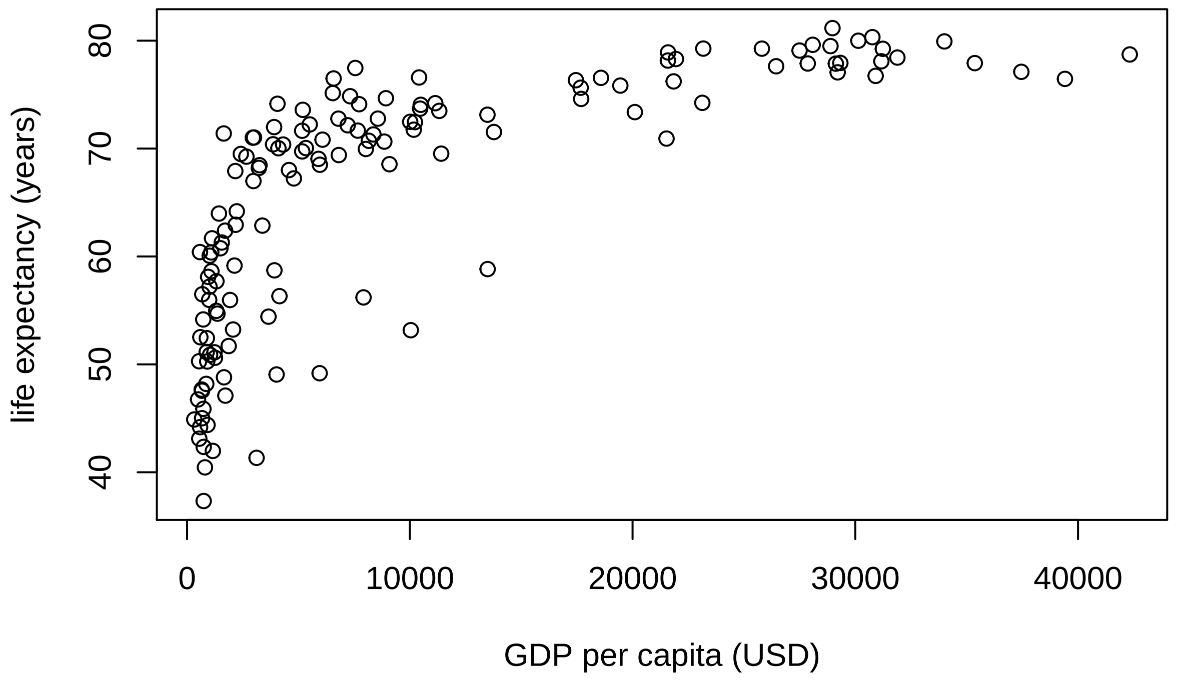
```
# In painstaking detail:
le <- gapminder$lifeExp  # extract life expectancy over all years
le <- matrix(le, nrow = 12, byrow = FALSE)  # make a matrix with years as rows
# and countries as columns
le <- le[9:12, ]  # extract the last four years for each country
avgle_4 <- colMeans(le)

# Do the same for GDP
gdp <- gapminder$gdpPercap
gdp <- matrix(gdp, nrow = 12, byrow = FALSE)
gdp <- gdp[9:12, ]
avggdp_4 <- colMeans(gdp)
```

A shorter way of doing this - that we haven't discussed yet - is to use tapply, but to provide a 'custom' function to calculate the mean of the last four years.

```
avgLE_4 <- tapply(gapminder$lifeExp, gapminder$country, function(x) {
    mean(x[9:12])
})
avgGDP_4 <- tapply(gapminder$gdpPercap, gapminder$country, function(x) {
```

```r
    mean(x[9:12])
})
```

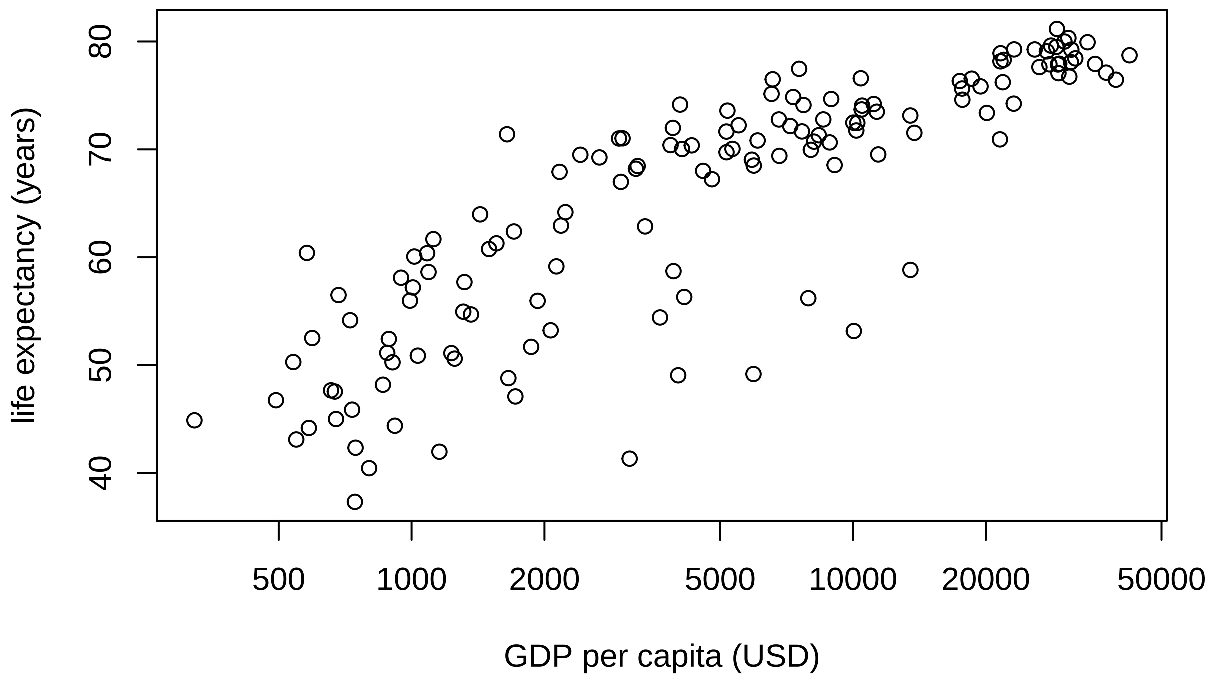
```r
plot(avgGDP_4, avgLE_4, xlab = "GDP per capita (USD)", ylab = "life expectancy (years)")
```



2. What do you notice about the scale of GDP? Transform it appropriately and re-plot.

Clearly the plot above is nonlinear, and furthermore, there is a very wide range of GDP. So, something to try (and indeed Hans Rosling does that too) is to plot the GPD on a log scale. You don't have to explicitly create a new log-transformed variable; have a look at the plot statement below to see how easy it is. After we carry out the transformation, the plot looks somewhat more linear, but there are several countries that do not follow the general linear pattern.

```r
plot(avgGDP_4, avgLE_4, xlab = "GDP per capita (USD)", ylab = "life expectancy (years)",
    log = "x")
```



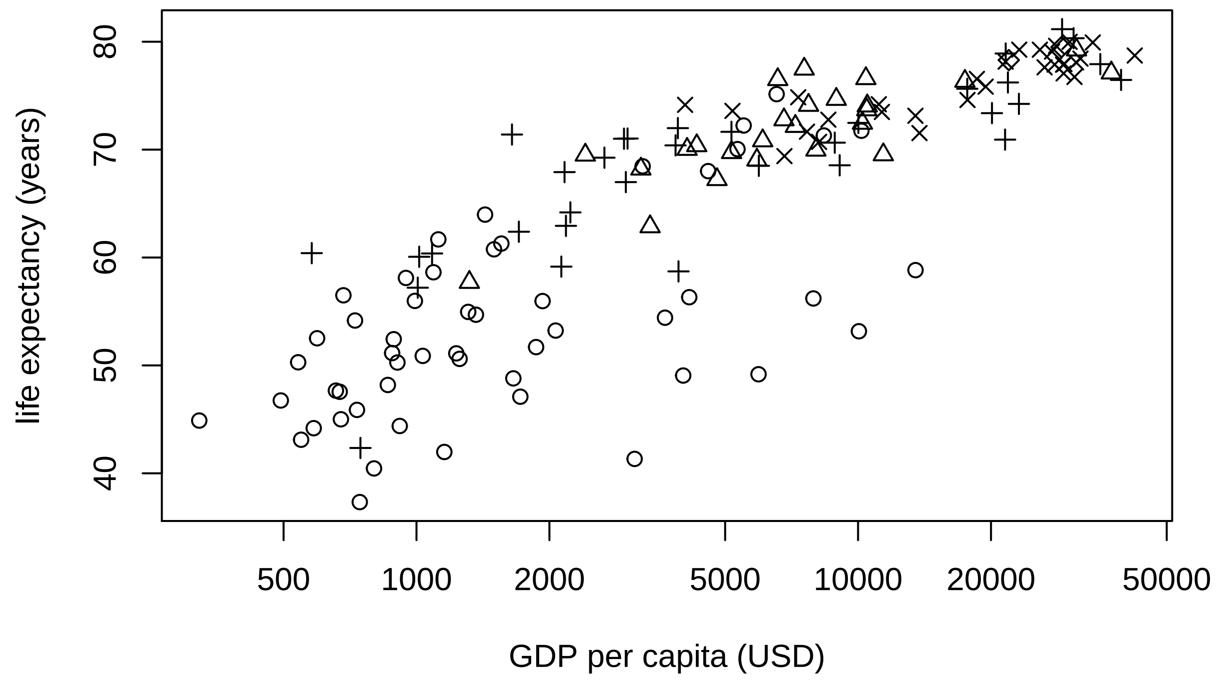
3. Use different symbols for countries in different continents. Hint: you'll need the argument `pch`, so see the help file for `plot`. First create a vector that gives you the continent that each country is in.

In the plot above, we have plotted the averages for the last four years for each country, and so there are 142 (the number of countries) pairs of points. So, we need to extract the continent that each of these countries is in. The brute-force way is to identify the index of the first row where a country first appears. See also a more elegant way just below that; to understand what it does, work it through from the inside out.

```
Continents <- gapminder$continent[1 + (0:141) * 12]
# alternatively, you could do this: Continents <-
# gapminder$continent[!duplicated(gapminder$country)]
```

Now the vector `Continents` is a character vector, and we're going to use it to define the plotting symbol; first, however, we need to convert it to a number, because the type of plotting characters, e.g., circles, squares, triangles, ..., are defined by assigning a numeric argument to `pch`.

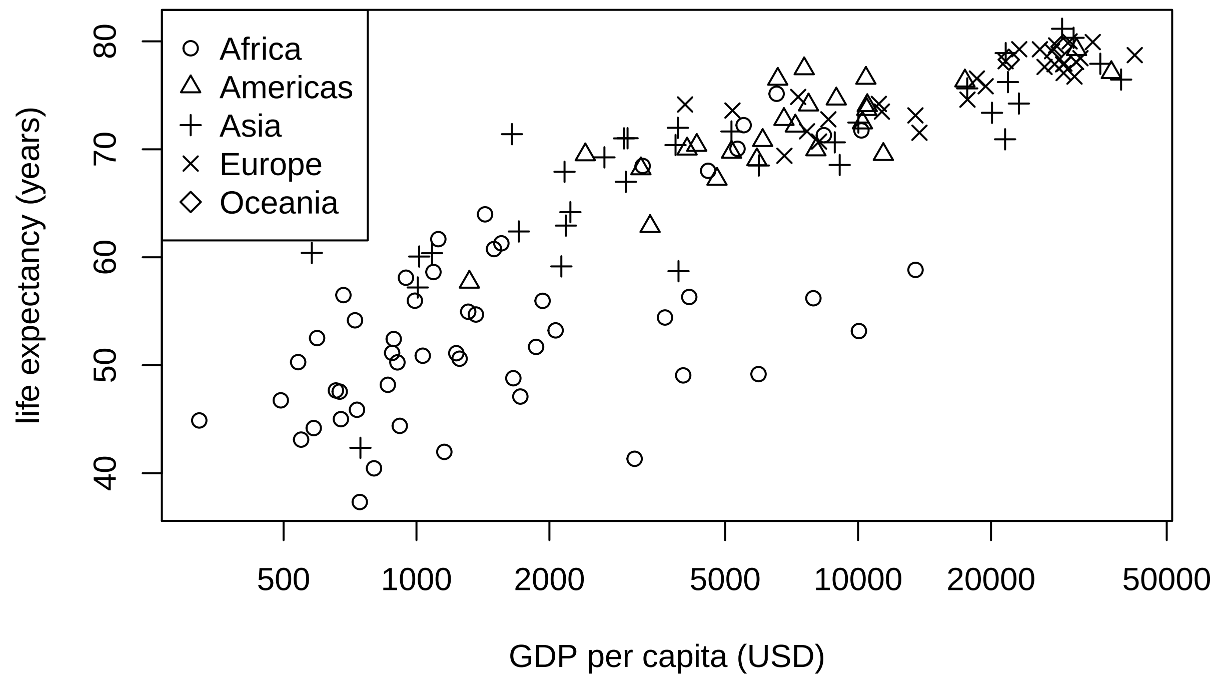
```
plot(avgGDP_4, avgLE_4, xlab = "GDP per capita (USD)", ylab = "life expectancy (years)",
    log = "x", pch = as.numeric(Continents))
```



4. Add a legend so that we know which symbol corresponds to which continent.

See the help for `legend` and the example below to see how to plot a legend.

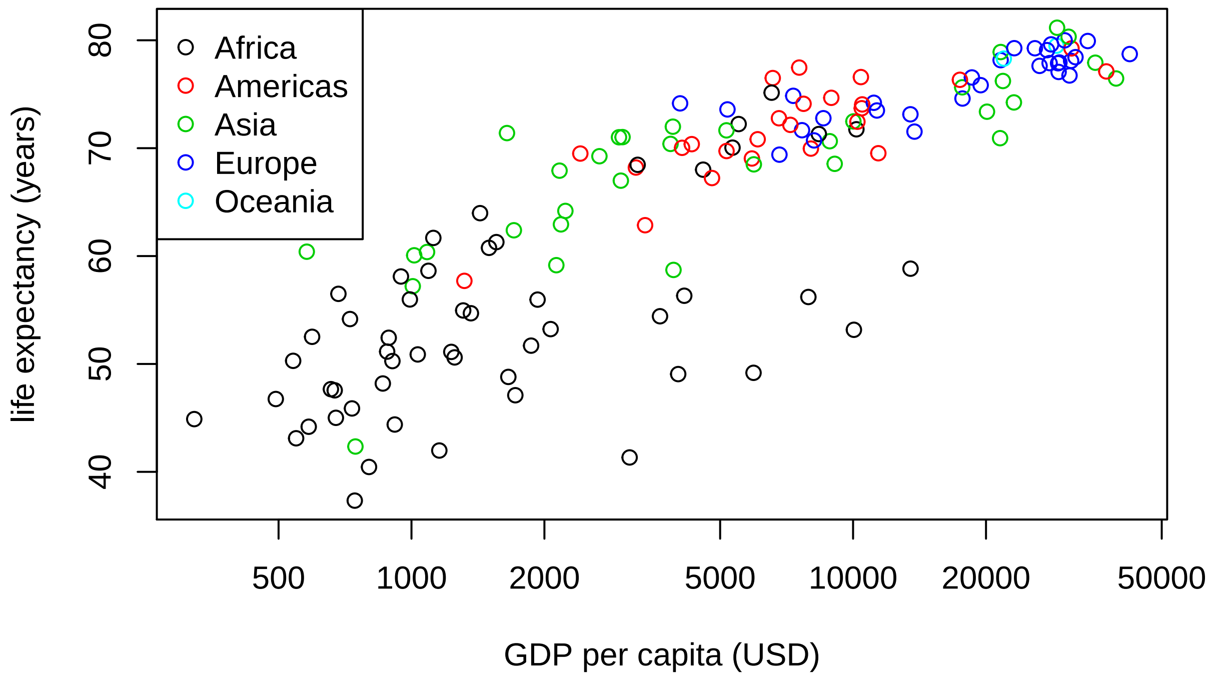
```r
plot(avgGDP_4, avgLE_4, xlab = "GDP per capita (USD)", ylab = "life expectancy (years)",
    log = "x", pch = as.numeric(Continents))
legend("topleft", legend = sort(unique(Continents)), pch = 1:5)
```



5. Construct a plot similar to Q4, but this time use the same symbol but different colours for each continent.

Finally, to produce a plot in which the continents have different colours, see below. The `col` argument can either be a character vector giving the colour (e.g., “black”, or it can be a number - 1 is black, 2 is red, 3 is blue, etc.)

```r
plot(avgGDP_4, avgLE_4, xlab = "GDP per capita (USD)", ylab = "life expectancy (years)",
    log = "x", col = as.numeric(Continents))
legend("topleft", legend = sort(unique(Continents)), col = 1:5, pch = 1)
```



Where do students live?

For a particular set of first-year units, `S1_2018_STAT1002_Lab_3.RData` contains some information on the number of students who live at different postcodes in WA. There is also information on the latitude and longitudes of the centroids of each postcode.

1. Load the data file. What objects are loaded, and what’s in them? For the time being, ignore the object `WA`. Which suburb has the highest number of students?

```r
print(load("S1_2020_STAT1003_Workshop_3.RData"))
```

```
[1] "SuburbTable" "WAPostcodes" "WA"
```

Make sure you understand what each of the data frames contains. `WAPostcodes` contains geographical information: postcodes, place names, and the latitude and longitude (presumably the centroid) of each postcode region. `SuburbTable` contains the information about students: postcodes in which they live, and how many of them live there.

The function `str` is useful if you want to get an overview of the structure of the variables in a data frame, and finding the suburb with the most students is straightforward.

```r
str(SuburbTable)
```

```
'data.frame':	243 obs. of  2 variables:
 $ Suburb   : Factor w/ 243 levels "ALEXANDER HEIGHTS",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ Frequency: int  3 2 1 7 2 4 4 2 1 3 ...
```

```r
str(WAPostcodes)
```

```
'data.frame':	1779 obs. of  7 variables:
 $ postcode  : int  6000 6000 6000 6003 6003 6004 6005 6005 6006 6007 ...
 $ place_name: chr  "PERTH" "PERTH GPO" "CITY DELIVERY CENTRE" "NORTHBRIDGE" ...
 $ state_name: Factor w/ 8 levels "Australian Capital Territory",..: 8 8 8 8 8 8 8 8 8 8 ...
 $ state_code: Factor w/ 8 levels "ACT","NSW","NT",..: 8 8 8 8 8 8 8 8 8 8 ...
 $ latitude  : num  -32 -32 -32 -31.9 -31.9 ...
 $ longitude : num  116 116 116 116 116 ...
 $ accuracy  : int  4 3 3 4 4 4 4 3 4 4 ...
```

```r
which.max(SuburbTable$Frequency)  # gives the row of PC_Table that contains the largest frequency
```

```
[1] 234
```

```r
SuburbTable[which.max(SuburbTable$Frequency), ]  # displays that row
```

```
       Suburb Frequency
234 WILLETTON        24
```

How would you find the suburb with the second-largest number of students?

2. **Merge** the two datasets together to create a new data frame that contains only information on the students for whom we have information. Call this merged data frame `GeoTable`.

The `merge` command is very useful to combine two data frames based on information in a common column, in this case, the variable `postcode`. Make sure you understand what's in the resulting data frame.

```r
GeoTable <- merge(SuburbTable, WAPostcodes, by.x = "Suburb", by.y = "place_name",
    all.x = TRUE)
```

3. Install the libraries `maps` and `mapdata` onto your computer, and then load them in to your *R* session.

To install the packages, you would use the `Tools -> Install Packages` menu item in RStudio and then put `maps` and `mapdata` in the dialogue box. Then, to load the libraries (or packages) into your R session, you would simply invoke the following command.

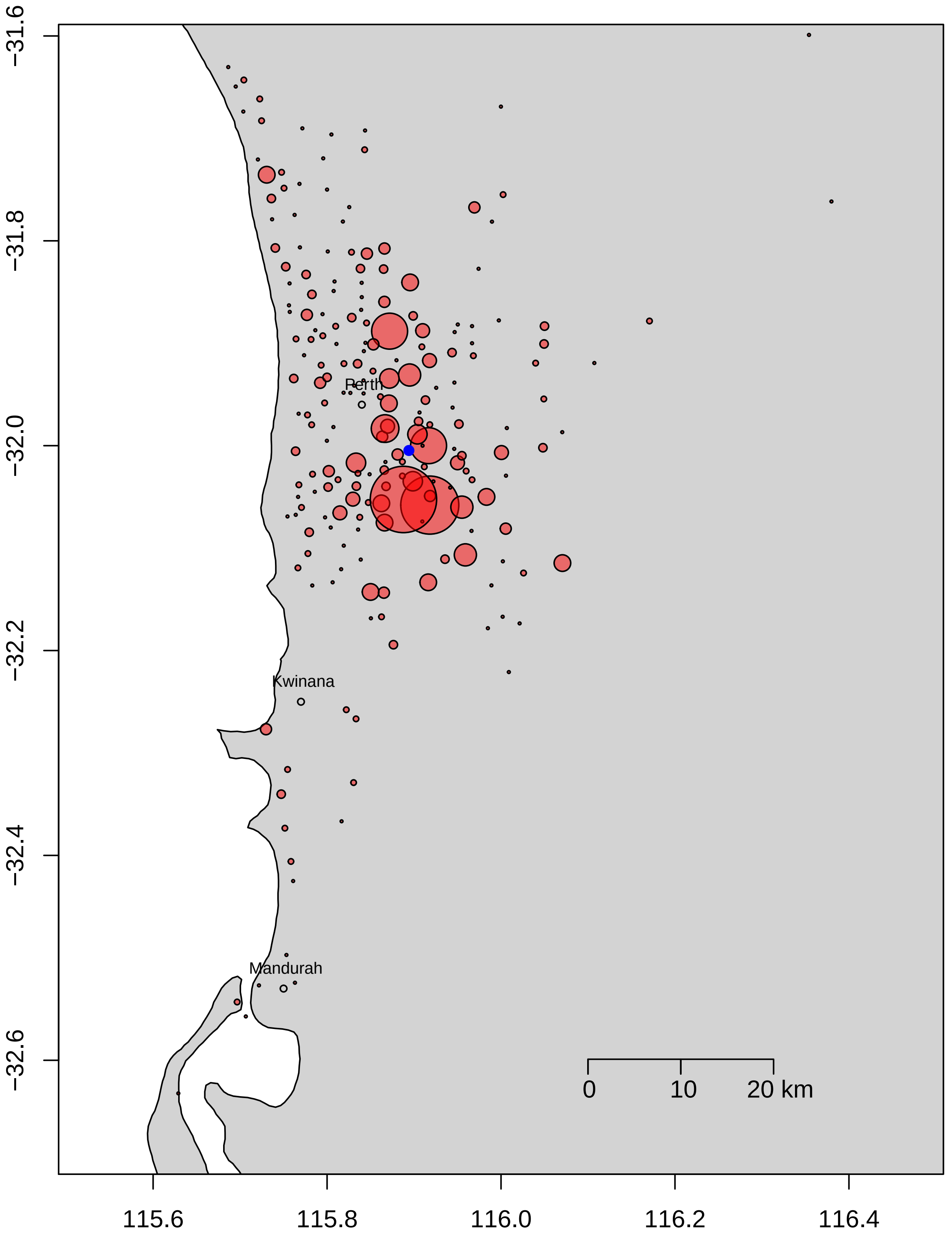
```r
library(maps)
library(mapdata)
```

4. Using the following code, look at the appropriate help files to plot on a map of the Perth area the frequency of students from each suburb. Use a larger symbol for larger frequencies. Fill in the appropriate values of `XX`, `YY`, `ZZ`, etc. Change the `FALSE` to `TRUE` in the code chunk header to evaluate the code.

In the `map` command, your task was to work out the latitude and longitude limits for the Perth region, and then assign them to the appropriate arguments. In the `points` command, you were meant to put the longitude and latitudes of each of the postcode areas in which the students live. Furthermore, we want bigger circles where there are more students, so we can simply assign to the **c**haracter **ex**pansion argument `cex` some scaled value of the student frequencies in each postcode.

```r
map("worldHires", ylim = c(-32.7, -31.6), xlim = c(115.5, 116.5), fill = TRUE, col = "lightgrey",
    mar = c(4.5, 4, 1, 1))
map.axes()
map.scale(x = 116.1, ratio = FALSE)
points(GeoTable$long, GeoTable$lat, cex = GeoTable$Frequency/4, pch = 21, col = "black",
    bg = rgb(1, 0, 0, 0.5))
map.cities()
```

```r
# Add a blue circle for the co-ordinates of Curtin University.
points(115.89405, -32.00469, pch = 16, col = "blue")
```



5. As appealing as the figure above might be, we can produce an even more useful map known as a *choropleth*. A choropleth is simply a map in which geographic regions are shaded according to a variable of interest. In this case, the geographic regions are suburbs, and of course, the variable of interest is the number of students from that suburb taking these units.

   The object `WA` is a *shapefile*, which contains, among other things, co-ordinates of polygons representing the boundaries of suburbs in WA, their names, areas, and other information.

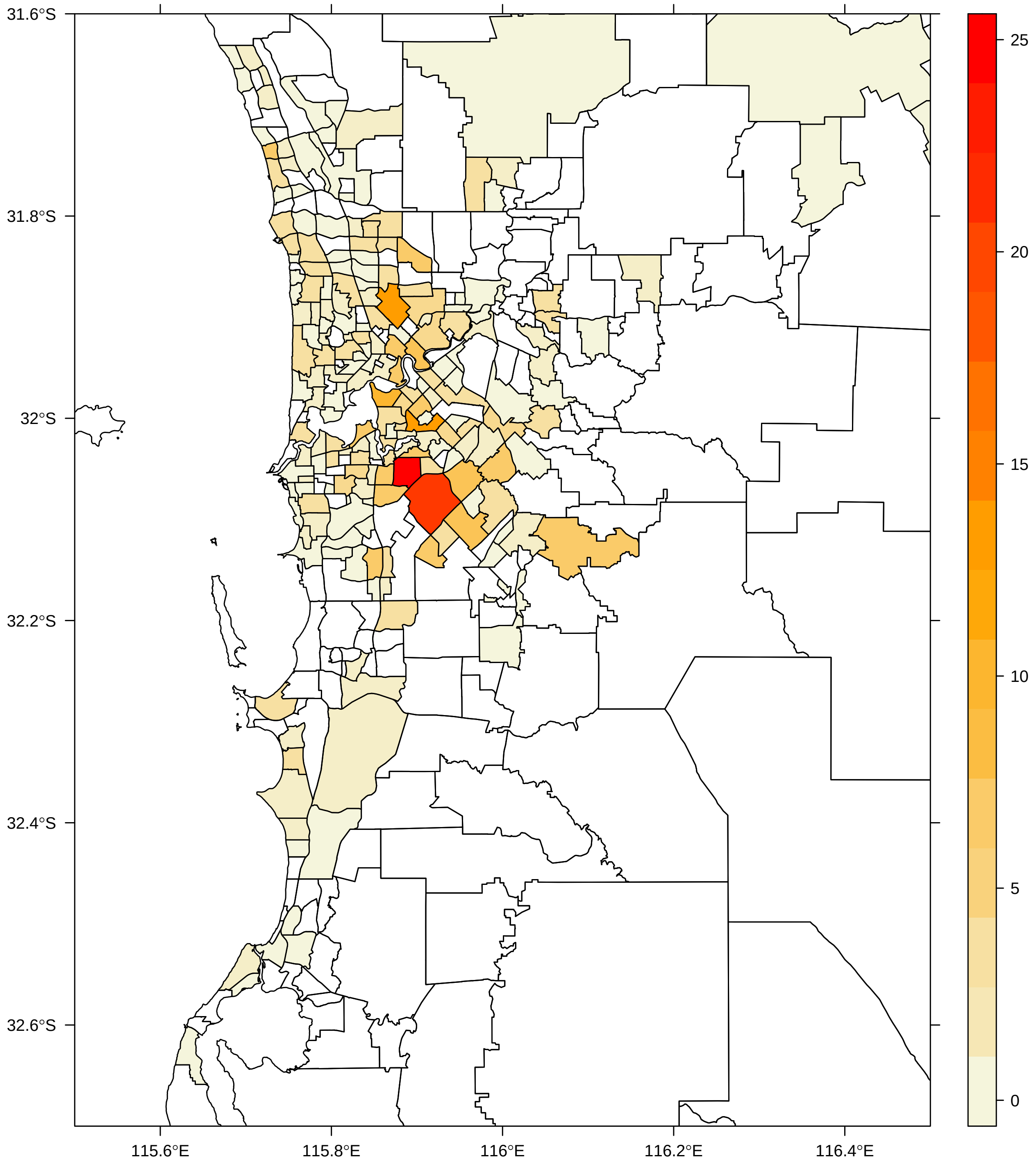
   As above, **merge** the object `GeoTable` above with the shapefile `WA`, and call the resulting object `Freq_by_Table`. Then run the code below, after loading installing the library `sp`.

   Which plot is more useful?

```
# This must be loaded before the merge() command below
library(sp)
```

```
# Merge with GeoTable
Freq_by_Suburb <- merge(WA, GeoTable, by.x = "SSC_NAME", by.y = "Suburb")
```

```
spplot(Freq_by_Suburb, z = "Frequency", xlim = c(115.5, 116.5), ylim = c(-32.7, -31.6),
    col.regions = colorRampPalette(c("beige", "orange", "red"))(24), scales = list(draw = TRUE))
```



31.6°S
31.8°S
32°S
32.2°S
32.4°S
32.6°S
115.6°E
115.8°E
116°E
116.2°E
116.4°E
25
20
15
10
5
0