Introduction to spatial analysis and GIS in R

Modified from a tutorial by Robin Lovelace (R.Lovelace@leeds.ac.uk), James Cheshire, Rachel Oldroyd and others

Stephanie Melles

October, 2017

Contents

Preface

This tutorial is an introduction to visualising and analysing spatial data in R based on the **sp** class system. For a guide to the more recent **sf** package check out tutorials by Edzer Pebesma or try Chapter 2 of the in-development book Geocomputation with R, the source code of which can be found at github.com/Robinlovelace/geocompr.

Although **sf** supersedes **sp** in many ways, the content in this tutorial teaches principles that will be useful regardless of software. Specifically this tutorial focusses on map-making with R's 'base' graphics and various dedicated map-making packages for R including **tmap** and **leaflet**. It aims to teach the basics of using R as a fast, user-friendly and extremely powerful command-line Geographic Information System (GIS).

By the end of the tutorial you should have the confidence and skills needed to convert a diverse range of geographical and non-geographical datasets into meaningful analyses and visualisations. All of the results are reproducible using data and code provided in this repository, culminating in publication-quality maps.

The tutorial is practical in nature: you will load-in, visualise and manipulate spatial data. We assume no prior knowledge of spatial data analysis but some experience with R will help. If you have not used R before, it may be worth following an introductory tutorial, such as *Efficient R Programming* (Gillespie and Lovelace, 2016), *R for Data Science* (Grolemund and Wickham, 2016) or tutorials suggested on rstudio.com and cran.r-project.org.

To distinguish between prose and code, please be aware of the following typographic conventions used in this document: R code (e.g. `plot(x, y)`) is written in a `monospace` font and package names (e.g. **rgdal**) are written in **bold**. A double hash (`##`) at the start of a line of code indicates that this is output from R.

Lengthy outputs have been omitted from the document to save space, so do not be alarmed if R produces additional messages: you can always look up them up on-line.

As with any programming language, there are often many ways to produce the same output in R. The code presented in this document is not the only way to do things. We encourage you to play with the code to gain a deeper understanding of R. Do not worry, you cannot 'break' anything using R and all the input data can be re-loaded if things do go wrong. As with learning to ride a bike, you learn by falling and getting back up and on your bike! Getting an `Error:` message in R is much less painful than falling onto concrete! Error's mean you are trying new things.

Part I: Introduction: a guide to R's syntax and preparing for the tutorial

Prerequisites

For this tutorial you need a copy of R. The latest version can be downloaded from http://cran.r-project.org/.

We also suggest that you use an R editor, such as RStudio, as this will improve the user-experience and help with the learning process. This can be downloaded from http://www.rstudio.com. The R Studio interface is comprised of a number of windows, the most important being the console window and the script window. Anything you type directly into the console window will not be saved, so use the script window to create scripts which you can save for later use. There is also a Data Environment window which lists the dataframes and objects being used. Familiarise yourself with the R Studio interface before getting started on the tutorial.

When writing code in any language, it is good practice to use consistent and clear conventions, and R is no exception. Adding comments to your code is also useful; make these meaningful so you remember what the code is doing when you revisit it at a later date. You can add a comment by using the `#` symbol before or after a line of code, as illustrated in the block of code below. This code should create Figure 1 if typed correctly into the Console window:

```
# Generate data
x <- 1:400
y <- sin(x / 10) * exp(x * -0.01)

plot(x, y) # plot the result
```

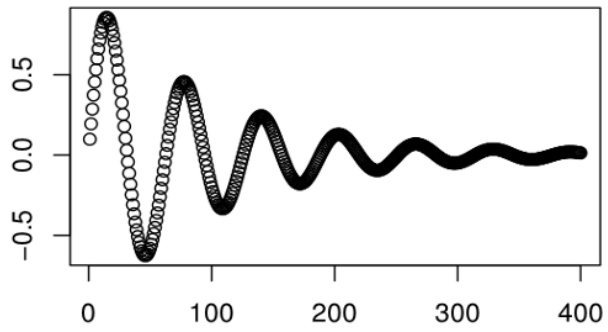


Figure 1: Basic plot of x and y (right) and code used to generate the plot (right).

This first line in this block of code creates a new *object* called `x` and assigns it to a range of integers between 1 and 400. The second line creates another object called `y` which is assigned to a mathematical formula, and

the third line plots the two together to create the plot shown.

Note `<-`, the directional "arrow" is an assignment symbol, which creates a new object and assigns it to the value you have given.[1]

If you require help on any function, use the `help` command, e.g. `help(plot)`. Because R users love being concise, this can also be written as `?plot`. Feel free to use it at any point you would like more detail on a specific function (although R's help files are famously cryptic for the un-initiated). Help on more general terms can be found using the `??` symbol. To test this, try typing `??regression`. For the most part, *learning by doing* is a good motto, so let's continue on and download some packages and data.

R Packages

R has a huge and growing number of spatial data packages. We recommend taking a quick browse on R's main website to see the spatial packages available: http://cran.r-project.org/web/views/Spatial.html.

In this tutorial we will use the packages from the '**sp**verse', that use the **sp** package:

- **ggmap**: extends the plotting package **ggplot2** for maps
- **rgdal**: R's interface to the popular C/C++ spatial data processing library gdal
- **rgeos**: R's interface to the powerful vector processing library geos
- **maptools**: provides various mapping functions
- **dplyr** and **tidyr**: fast and concise data manipulation packages
- **tmap**: a new packages for rapidly creating beautiful maps

For a tutorial based on the recent **sf** package you will have to look at Edzer Pebesma's tutorials, or elsewhere - like the website geocompr, which is the online home of the forthcoming book *Geocomputation with R*.

Some packages may already be installed on your computer. To test if a package is installed, try to load it using the `library` function; for example, to test if **ggplot2** is installed, type `library(ggplot2)` into the console window. If there is no output from R, this is good news: it means that the library has already been installed on your computer.

If you get an error message, you will need to install the package using `install.packages("ggplot2")`. The package will download from the Comprehensive R Archive Network (CRAN); if you are prompted to select a 'mirror', select one that is close to current location. If you have not done so already, install these packages on your computer now. A quick way to do this in one go is to enter the following lines of code:

```
x <- c("ggmap", "rgdal", "rgeos", "maptools", "dplyr", "tidyr", "tmap")
 # install.packages(x) # warning: uncommenting this may take a number of minutes
lapply(x, library, character.only = TRUE) # load the required packages
```

[1] Tip: typing `Alt -` on the keyboard will create the arrow in RStudio. The equals sign `=` also works.

Part II: Spatial data in R - describes basic spatial functions in R

Starting the tutorial and downloading the data

Now that we have looked at R's basic syntax and installed the necessary packages, let's load some real spatial data. The next part of the tutorial will focus on plotting and exploring spatial objects. The first file we are going to load into R Studio is the City of Toronto's neighbourhood map, "NEIGHBOURHOODS_WGS84", which can be downloaded at *opendata.toronto.ca*. This *shapefile* will be downloaded to the 'data' folder of your project. It is worth looking at this input dataset in your file browser before opening it in R. You will notice that there are several files named "NEIGHBOURHOODS_WGS84", all with different file extensions. This is because a shapefile is actually made up of a number of different files, such as .prj, .dbf and .shp.

```
download.file("http://opendata.toronto.ca/gcc/neighbourhoods_planning_areas_wgs84.zip",
              "data/neighbourhoods_planning_areas_wgs84.zip")
setwd("C:/Steph2016_2017/GITHUB/Intro_spatial_GIS_R/data")
unzip("neighbourhoods_planning_areas_wgs84.zip", files = NULL, list = FALSE,
      overwrite = TRUE, junkpaths = FALSE, exdir = ".", unzip = "internal",
      setTimes = FALSE)
```

You could also try opening the file "NEIGHBOURHOODS_WGS84.shp" file in a conventional GIS such as QGIS or ArcGIS to see what a shapefile contains. You should also open "NEIGHBOURHOODS_WGS84.dbf" in a spreadsheet program such as Excel to see what this file contains. Once you think you understand the input data, it's time to open it in R. There are a number of ways to do this, and the most commonly used and versatile manner is to use `readOGR`. This function, from the **rgdal** package, automatically extracts the information regarding the data. **rgdal** is Rs interface to the "Geospatial Abstraction Library (GDAL)", which is used by other open source GIS packages such as QGIS and enables R to handle a broader range of spatial data formats. If you've not already *installed* and loaded the **rgdal** package (see the 'prerequisites and packages' section) do so now.

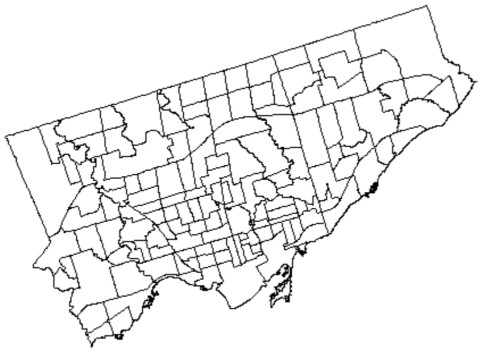


Figure 2: Plot of City of Toronto shapefile using basic plotting function without georeferencing.

```
plot(TorontoMap)
```

The second line of code above the `readOGR` function is used to load a shapefile and assign it to a new spatial object called `TorontoMap`. This shapefile has an associated data table that contains information about the name and ID of each neighbourhood.

```
head(TorontoMap@data, n = 2)
```

```
##   AREA_S_CD                    AREA_NAME
## 0       097          Yonge-St.Clair (97)
## 1       027 York University Heights (27)
```

`readOGR` is a *function* of the **rgdal** package, the first *argument* of which is `dsn`: "data source name", the file or directory where the geographic data are stored. Important note about objects inside a function's brackets: arguments are separated by a comma, and the order in which they are specified is important. You do not have to explicitly type `dsn =` or `layer =` as functions in R are programmed to expect arguments in a particular order. `readOGR("data", "NEIGHBORHOODS_WGS84")` would work just as well. For clarity, it is good practice to include argument names when learning new functions so we will continue to do so.

`TorontoMap` is now an object representing Toronto neighbourhood boundaries. For information about how to load different types of spatial data, see the help documentation for `readOGR`. This can be accessed by typing `?readOGR`.

Attribute joins

Attribute joins are used to link additional pieces of information to our polygons. In the `TorontoMap` object, for example, we have 2 attribute variables — that can be found by typing `names(TorontoMap)`. But what happens when we want to add more variables from an external source? We will use the example of City of Toronto Health and Wellbeing data to demonstrate this. These data were downloaded from: *opendata.toronto.ca*. The spreadsheet contains non-spatial data on health and wellbeing statistics per neighbourhood from 2011. The spreadsheet was opened in Excel and saved as a 'City_Tor_WB-Health_2011.csv' file from the sheet called, '*RawData-Ref Period 2011*'.

```
TorontoHealth <- read.csv("data/City_Tor_WB-Health_2011.csv")
# load related data on health and wellbeing in the City
```

```
nrow(TorontoMap) # return the number of rows in the data attribute table of our shapefile.
```

```
## [1] 140
```

```
# This is equivalent to the number of polygons (neighbourhoods) in the shapefile.
nrow(TorontoHealth)
```

```
## [1] 140
```

```
names(TorontoHealth)
```

```
## [1] "Neighbourhood"    "NID"              "BCS"
## [4] "CCS"              "DineSafe"         "FemFertil"
## [7] "HealthProviders"  "PremMort"         "StudentNutrition"
```

The number of rows in the neighbourhood health data table must be equivalent to the number of rows in the shapefile for us to create an attribute join. Now the challenge is to join the health and wellbeing data to the `TorontoMap` object. We will base our join on neighbourhood ID's, `NID`, from the `TorontoHealth` dataset object. It is not always straight-forward to join objects based on names as the names do not always match. Let's see which names in the `TorontoMap` shapefile object match the health and wellbeing data.

```
head(TorontoMap@data) # returns first five lines of datatable
```

```
##   AREA_S_CD                    AREA_NAME
## 0       097          Yonge-St.Clair (97)
## 1       027 York University Heights (27)
## 2       038        Lansing-Westgate (38)
## 3       031      Yorkdale-Glen Park (31)
```

```
## 4       016     Stonegate-Queensway (16)
## 5       118 Tam O'Shanter-Sullivan (118)
```

```
# AREA_S_CD represents the neighbourhood ID,
# but this variable was read in as character string
# because of the way it was coded in Excel.
# The variable was then converted to a FACTOR variable when imported in R.
class(TorontoMap$AREA_S_CD) # factor
```

```
## [1] "factor"
```

We can *coerce* the variable into the correct, numeric, format with the command:

```
TorontoMap$AREA_S_CD<-as.numeric(TorontoMap$AREA_S_CD)
# change variable class to numeric before we can do the join
```

We are now ready to join the spatial and non-spatial datasets. It is recommended to use the `left_join` function from the **dplyr** package but the `merge` function could equally be used. Note that when we ask for help for a function that is not loaded, nothing happens, indicating we need to load it:

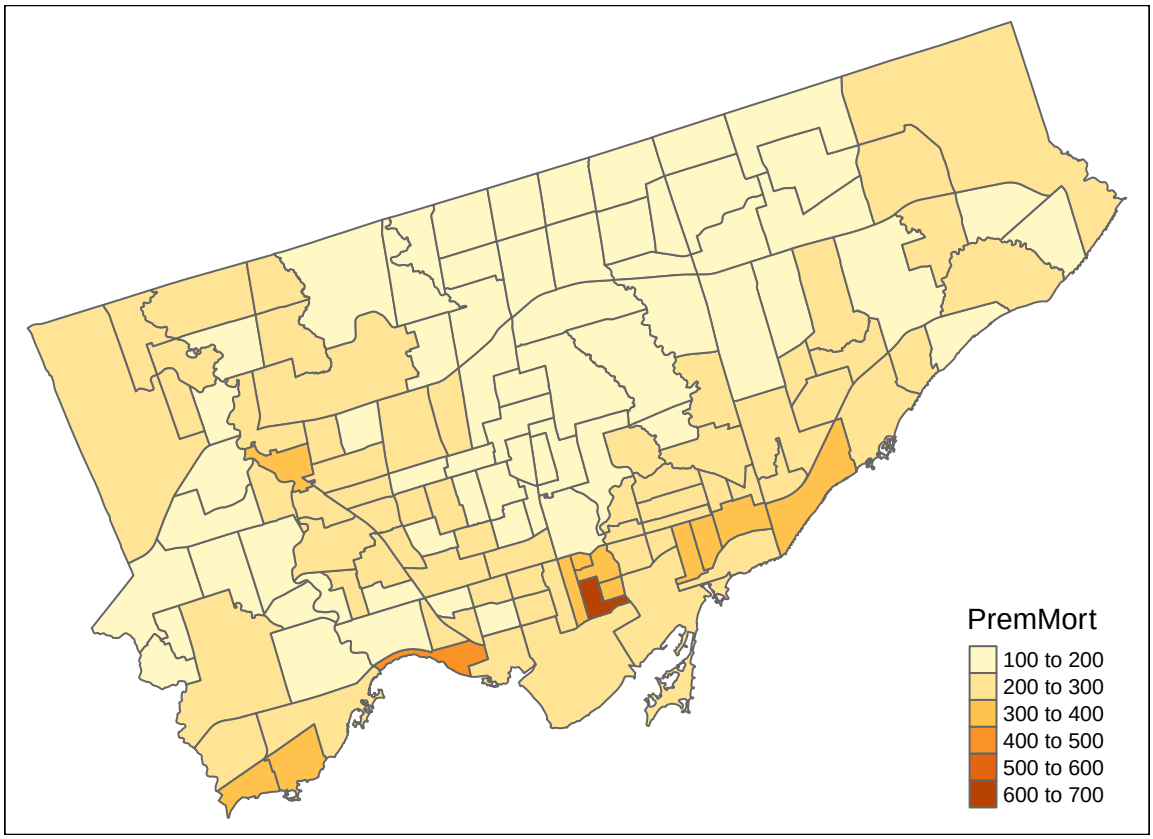
We use `left_join` because we want the length of the data frame to remain unchanged, with variables from new data appended in new columns (see `?left_join`). The `*join` commands (including `inner_join` and `anti_join`) assume, by default, that matching variables have the same name. Here we will specify the association between variables in the two data sets:

```
head(TorontoMap$AREA_S_CD) # dataset to add to (results not shown)
head(TorontoHealth$NID) # the variables to join

# head(left_join(TorontoMap@data, TorontoHealth)) # test it works
TorontoMap@data <- left_join(TorontoMap@data, TorontoHealth, by = c('AREA_S_CD' = 'NID'))
```

Take a look at the new `TorontoMap@data` object. You should see new variables added, meaning the attribute join was successful. Congratulations! You can now plot the rate of premature mortality in Toronto by Neighbourhood (see below).

```
library(tmap) # load tmap package (see Section III)
qtm(TorontoMap, "PremMort") # plot the basic map
```



With the attribute joining skills you have learned in this section, you should now be able to take datasets from many sources, e.g. *opendata.toronto.ca*, and join them to your geographical data.

The structure of spatial data in R

Spatial objects like the `TorontoMap` object are made up of a number of different *slots*, the key *slots* being `@data` (non geographic *attribute data*) and `@polygons` (or `@lines` for line data). The data *slot* can be thought of as an attribute table and the geometry *slot* holds the polygons that make up the physcial boundaries. Specific *slots* are accessed using the `@` symbol. Let's now analyse the Toronto health and wellbeing attribute object with some basic commands:

```
head(TorontoMap@data, n = 2)
```

```
##   AREA_S_CD                    AREA_NAME           Neighbourhood   BCS
## 1        97          Yonge-St.Clair (97)          Yonge-St.Clair 70.10
## 2        27 York University Heights (27) York University Heights 59.93
##    CCS DineSafe FemFertil HealthProviders PremMort StudentNutrition
## 1 71.8       14  29.69886              56    142.0               0
## 2 58.9       46  52.58659              67    186.5            4065
```

```
mean(TorontoMap@data$PremMort) # rate of deaths per 100,000)
```

```
## [1] 217.0057
```

Take a look at the output created (note the table format of the data and the column names). There are two important symbols at work in the above block of code: the `@` symbol in the first line of code is used to refer to the data *slot* of the `TorontoMap` object. The `$` symbol refers to the `PremMort` column (a variable within

the table) in the `data` *slot*, which was identified from the result of running the first line of code.

We've seen the `head` function already above and it simply means "show the first few lines of data" (try entering `head(TorontoMap@data)`, see `?head` for more details).

The second line calculates the mean rate of Premature Deaths (under the age of 75) per 100,000 people for neighbourhoods in Toronto. The results works because we are dealing with numeric data. To check the classes of all the variables in a spatial dataset, you can use the following command:

```
sapply(TorontoMap@data, class)
```

```
##        AREA_S_CD        AREA_NAME    Neighbourhood              BCS
##        "numeric"         "factor"         "factor"        "numeric"
##              CCS         DineSafe        FemFertil  HealthProviders
##        "numeric"        "integer"        "numeric"        "integer"
##         PremMort StudentNutrition
##        "numeric"        "integer"
```

Basic plotting

Now we have seen something of the structure of spatial objects in R, let us look at plotting them. Note, that plots use the *geometry* data, contained primarily in the `@polygons` slot.

```
plot(TorontoMap) # shown above - try it on your computer
```

`plot` is one of the most useful functions in R, as it changes its behaviour depending on the input data (this is called *polymorphism* by computer scientists). Inputting another object such as `plot(TorontoMap@data)` will generate an entirely different type of plot. Thus R is intelligent at guessing what you want to do with the data you provide it with.

R has powerful subsetting capabilities that can be accessed very concisely using square brackets, as shown in the following example:

```
# select rows of TorontoMap@data where PremMort is less than 120
TorontoMap@data[TorontoMap$PremMort < 120, 1:3]
```

```
##     AREA_S_CD                              AREA_NAME
## 114        52                   Bayview Village (52)
## 115        39               Bedford Park-Nortown (39)
## 117        41 Bridle Path-Sunnybrook-York Mills (41)
##                         Neighbourhood
## 114                   Bayview Village
## 115               Bedford Park-Nortown
## 117 Bridle Path-Sunnybrook-York Mills
```

The above line of code asked R to select only the rows from the `TorontoMap` object, where the Premature Death rate is lower than 120, in this case rows 114, 115 and 117, which are Bayview Village, Bedford Park-Nortown and Bridle Path-Sunnybrook-York Mills respectively. The square brackets work as follows: anything before the comma refers to the rows that will be selected, anything after the comma refers to the number of columns that should be returned. For example if the data frame had 1000 columns and you were only interested in the first two columns you could specify `1:2` after the comma. The ":" symbol simply means "to", i.e. columns 1 to 2. Try experimenting with the square brackets notation (e.g. guess the result of `TorontoMap@data[1:2, 1:3]` and test it).

So far we have been examining only the attribute data *slot* (`@data`) of the `TorontoMap` object, but the square brackets can also be used to subset spatial objects, i.e. the geometry *slot*. Using the same logic as before try to plot a subset of neighbourhoods with high Premature Mortalities.

```
# Select zones where Premature Mortality is between 200 and 300
sel <- TorontoMap$PremMort > 200 & TorontoMap$PremMort < 300
plot(TorontoMap[sel, ]) # output not shown here
```

This plot is quite useful, but it only displays the areas which meet the criteria. To see the sporty areas in context with the other areas of the map simply use the `add = TRUE` argument after the initial plot. (`add = T` would also work, but we like to spell things out in this tutorial for clarity). What do you think the `col` argument refers to in the below block?

If you wish to experiment with multiple criteria queries, use `&`.

```
plot(TorontoMap, col = "lightgrey") # plot the neighbourhood shapefile object
sel <- TorontoMap$PremMort > 250
plot(TorontoMap[ sel, ], col = "turquoise", add = TRUE) # add selected zones to map
```

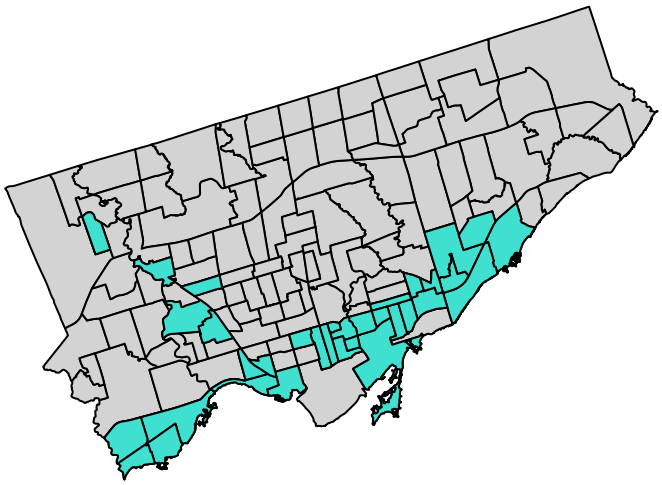


Figure 3: Simple plot of Toronto with areas of high mortality highlighted in blue

Congratulations! You have just examined and visualised a spatial object: where are the neighbourhoods with high levels of Premature Mortality in Toronto? The map tells us a story. Do not worry for now about the intricacies of how this was achieved: you have learned vital basics of how R works as a language; you will learn more details in subsequent sections.

You should not expect to understand everything in this tutorial upon first try: simply typing the commands and thinking briefly about the outputs is all that is needed at this stage.

Part III: Making maps with tmap, ggplot2 and leaflet

Map making with **tmap**, **ggplot2** and **leaflet**: this section demonstrates map making with more advanced visualisation tools.

tmap

tmap was created to overcome some of the limitations of base graphics and **ggmap**. A concise introduction to **tmap** can be accessed (after the package is installed) by using the vignette function:

```
# install.packages("tmap") # install the CRAN version
library(tmap)
vignette("tmap-nutshell")
```

A couple of basic plots show the package's intuitive syntax and attractive default parameters.

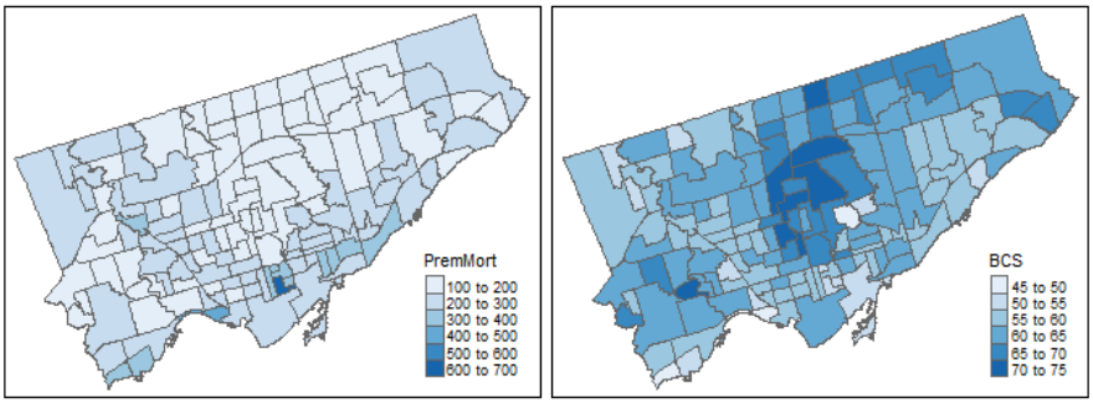


Figure 4: Side-by-side maps of Premature Mortality and Breast Cancer Screening rates.

```
qtm(shp = TorontoMap, fill = c("PremMort", "BCS"), fill.palette = "Blues", ncol=2)
```

The plot above shows the ease with which tmap can create maps next to each other for different variables, but the placement of the legend is problematic.

```
?tmap # get more info on tmap
```

ggmap

ggmap is based on the **ggplot2** package, an implementation of the Grammar of Graphics (Wilkinson 2005). **ggplot2** can replace the base graphics in R (the functions you have been plotting with so far). It contains default options that match good visualisation practice and is well-documented: http://docs.ggplot2.org/current/ .

As a first attempt with **ggplot2** we can create a scatter plot with the attribute data in the `TorontoMap` object created previously:

```
library(ggplot2)
p <- ggplot(TorontoMap@data, aes(BCS, CCS))
```

What you have just done is set up a ggplot object where you say where you want the input data to come from. `TorontoMap@data` is actually a data frame contained within the wider spatial object `TorontoMap` (the `@` enables you to access the attribute table of the shapefile). The characters inside the `aes` argument refer to the parts of that data frame you wish to use (the variables `BCS` and `CCS`). This has to happen within the brackets of `aes()`, which means, roughly speaking 'aesthetics that vary'.

If you just type p and hit enter you get an empty plot. This is because you have not told `ggplot` what you want to do with the data. We do this by adding so-called "geoms", in this case `geom_point()`.

```
p + geom_point()
```

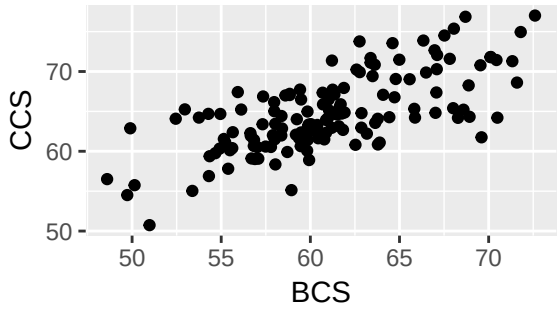


Figure 5: A simple graphic produced with **ggplot2**

Within the brackets you can alter the nature of the points. Try something like 'p + geom_point(aes(colour = "red") and experiment.

```
p + geom_point(aes(colour = "red"))
```

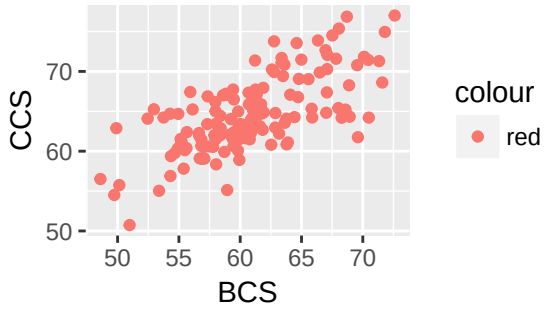


Figure 6: A simple graphic produced with **ggplot2**

This idea of layers (or geoms) is quite different from the standard plot functions in R, but you may find that each of the functions does a lot of clever stuff to make plotting much easier (see the documentation for a full list).

Creating interactive maps with leaflet

Leaflet is the world's premier web mapping system, serving hundreds of thousands of maps worldwide each day. The JavaScript library actively developed at github.com/Leaflet/Leaflet, has a strong user community. It is fast, powerful and easy to learn.

The **leaflet** package creates interactive web maps in few lines of code. One of the exciting things about the package is its tight integration with the R package for interactive on-line visualisation, **shiny**. Used together, these allow R to act as a complete map-serving platform, to compete with the likes of GeoServer! For more information on **rstudio/leaflet**, see rstudio.github.io/leaflet/ and the following on-line tutorial: robinlovelace.net/r/2015/02/01/leaflet-r-package.html.

```
library(leaflet)
summary(TorontoMap)
leaflet() %>%
  addTiles() %>%
  addPolygons(data = TorontoMap)
```

Figure 7: Snapshot of what your interactive map will look like.

```
m = leaflet() %>% addTiles()  # a map with the default OSM tile layer
m
```

```
# follow 'Leaflet' link on map
m %>% fitBounds(-79, 43, -80, 44) # check out fitBounds details using ?fitBounds
```

```
# move the center to Ryerson
m = m %>% setView(-79.3788, 43.6577, zoom = 17)
m
```

popups

```
m %>% addPopups(-79.3788, 43.6577, '<b>Ryerson University Campus</b>')
```

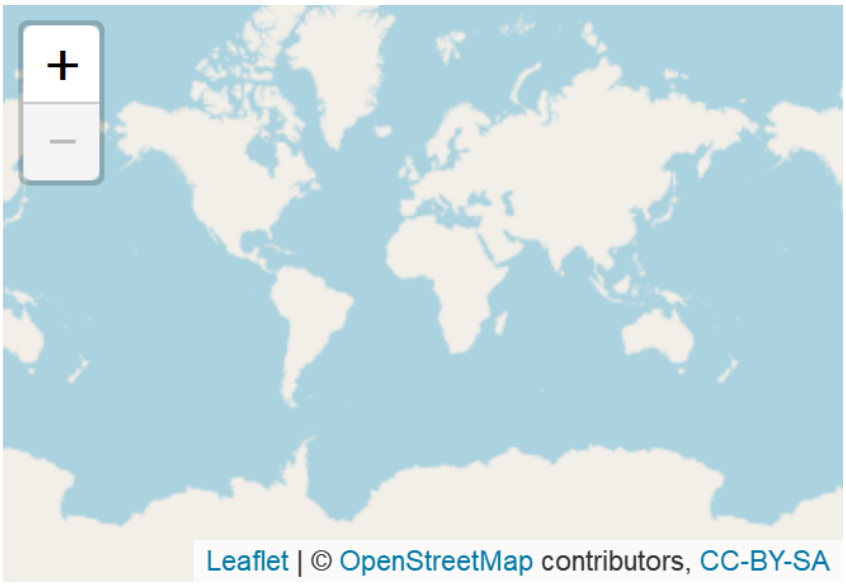


Figure 8: Explore the world of leaflet.

Figure 9: Centre your world in the Great Lakes Region.

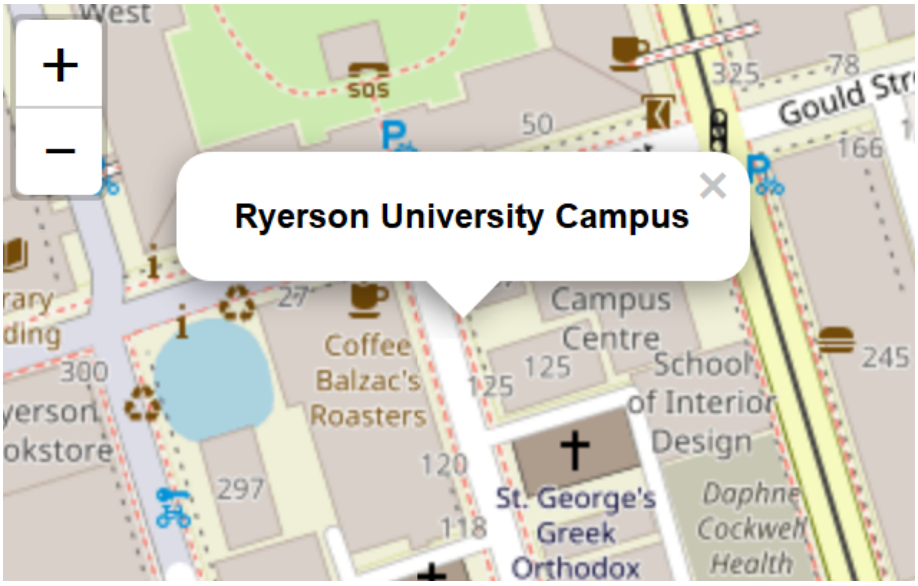


Figure 10: Check out Ryerson.