IHME

Instituto Nacional de Salud Pública

Spatial mapping in R

Shade Wilson

Jamal Yearwood

16/8/19

UNIVERSITY of WASHINGTON

Institute for Health Metrics and Evaluation

Who we are

Shade Wilson
Post-Bachelor Fellow
Skin Diseases, Causes of Death

Jamal Yearwood
Post-Bachelor Fellow
Heath Access and Equality Index (HAQI), Health Systems

Introductions

This is a workshop, not a lecture! We want to know more about you

- Name
- Your work at INSP

- Experience using R
- Experience in Stata, other programming languages

About this training

- Aim to talk ~15 mins per section
- Majority of time on exercises

- Goals:
  - R basics
  - Spatial mapping

- Feel free to stop and ask questions

Outline

- Intro to R
- R basics
- Vectors
- Data frames
- Introduction to spatial data in R
- Manipulating spatial data

If time permits:

- R vs Stata (?)
- Spatial data in R using ggplot2 and leaflet

Intro to R

What is R?

- R is a language for statistical computing and graphics
- Originally developed in 1992 by Robert Gentleman and Ross Ihaka based on the programming language S
- The core of the R language is maintained by the R Core Team
- A (very) large number of packages which add additional functionality are maintained by other contributors

Why use R?

- R can do many useful things
  - Flexible data management
  - Powerful statistical capabilities, particularly for modeling
  - Extensive graphics capabilities

- R is free software - You don't have to pay for it (and you can share it with anyone) - You can use and modify it as you see fit

- R has a large (and enthusiastic) user base - This makes finding help relatively straightforward - New methods are often implemented in R very quickly

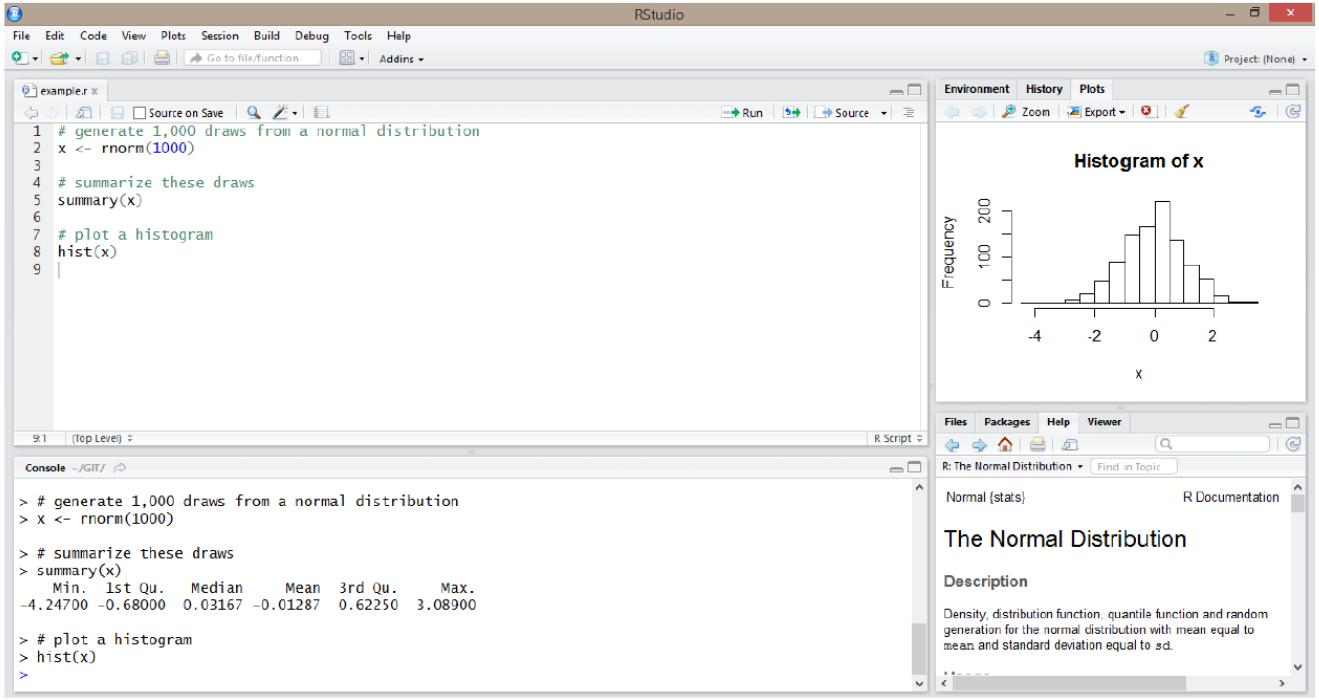
What is RStudio?

- "Integrated development environment"

- Convenient interface for R which incorporates a number of useful features for developing code - syntax highlighting - code completion - code navigation - debugging tools - etc.

- Also provides integration with other useful tools - Shiny (for developing web apps) - R Markdown (for authoring documents and slides) - Git/Subversion (for version control)

RStudio interface



R Basics

Packages

Most basic R functionality is part of `base` and is loaded automatically when you start R. Additional functionality can be added through packages.

The first time you use a package, it needs to be installed:

```
> install.packages("ggplot2")
```

After that, you just need to load the package using the `library()` command whenever you start a new instance of R:

```
> library(ggplot2)
```

R as calculator: Quick exercise

1. How many seconds are there in September?

```
> 30 * 24 * 60 * 60
[1] 2592000
```

2. What is 80 degrees Fahrenheit in degrees Celsius?

```
> (80 - 32) * (5/9)
[1] 26.66667
```

3. How much longer is 1 mile than 1600 meters (in feet)?

```
> 5280 - 1600 * 3.28084
[1] 30.656
```

Functions

R functions are used to transform input into output in some way.

For example...

```
> log(10)
[1] 2.302585
```

```
> exp(3)
[1] 20.08554
```

```
> sqrt(80)
[1] 8.944272
```

Functions: anatomy

```
> log(x = 300, base = 10)
[1] 2.477121
```

1. Function name: `log()`
2. Argument name(s): `x`, `base`
3. Argument value(s): `300`, `10`
4. Output: `2.4771213`

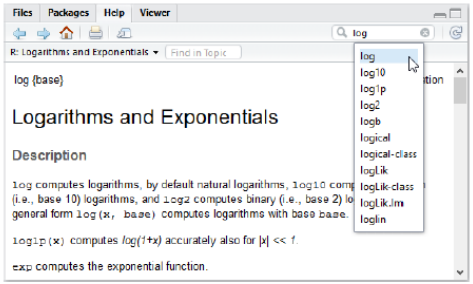
Help files

Every function has a help file.

You can access a help file from the console:

```
> help(log)
```

or from the help tab in RStudio:



Vectors

Objects

Objects are how we store information (e.g., data, functions) in R.

Objects have three components:

1. Name
2. Value
3. Properties (class, dimension, etc.)

Objects

We create objects using an assignment operator (either <- or =) to assign a value to a name:

```
> my_office <- 424
> my_office
[1] 424
```

```
> my_office = 424
> my_office
[1] 424
```

Basic classes

Characters

Country iso3 codes: CAN, USA, MEX

Numerics

Population: 35.16, 318.9, 122.3

Integers

Number of administrative units: 13, 51, 31

Logicals

Primary languge is Spanish: FALSE, FALSE, TRUE

Vectors

A vector is an *ordered* collection of values that are all of the *same basic class*.

Vectors can be created manually using the combine function `c()`:

```
> iso3 <- c("CAN", "USA", "MEX")
> iso3
[1] "CAN" "USA" "MEX"
```

```
> pop <- c(35.16, 318.9, 122.3)
> pop
[1]  35.16 318.90 122.30
```

```
> admin1 <- c(13L, 51L, 31L)
> admin1
[1] 13 51 31
```

```
> spanish <- c(FALSE, FALSE, TRUE)
> spanish
[1] FALSE FALSE  TRUE
```

Creating vectors

There are lots of other functions which create vectors. Some useful ones:

`seq()` or `:` for numeric sequences:

```
> seq(from = 0, to = 27, by = 3)
 [1]  0  3  6  9 12 15 18 21 24 27
> 1990:2000
 [1] 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000
```

`rep()` for repeating elements of any class of vector:

```
> rep(c(1, 2, 3), each = 2)
[1] 1 1 2 2 3 3
> rep(c("red", "blue", "green"), times = 2)
[1] "red"   "blue"  "green" "red"   "blue"  "green"
```

Indexing by number

You can select one or more values from within a vector using the position number:

```
> hi_temp[1]
[1] 77
> hi_temp[c(1, 3, 10)]
[1] 77 75 76
> hi_temp[1:5]
[1] 77 73 75 80 79
> hi_temp[seq(1, 31, 7)]
[1] 77 73 84 72 81
> hi_temp[c(1, 1, 1)]
[1] 77 77 77
```

Indexing by logical

You can also select values from within a vector using logicals.

```
> iso3[c(FALSE, FALSE, TRUE)]
[1] "MEX"
> iso3[spanish == TRUE]
[1] "MEX"
> iso3[spanish]
[1] "MEX"
```

This is generally how you select values that meet a certain criteria:

```
> pop > 300
[1] FALSE  TRUE FALSE
> pop[pop > 300]
[1] 318.9
> iso3[pop > 300]
[1] "USA"
```

Indexing by logical

This also one way to find missing (or non missing) values:

```
> rain[is.na(rain)]
[1] NA NA NA
> rain[!is.na(rain)]
[1] FALSE FALSE FALSE FALSE
```

More complicated relational/logical statements can also be used:

```
> iso3[pop > 100 & admin1 < 50]
[1] "MEX"
```

A quick note about object names...

Object names MUST start with an alphabetic character (a-z, A-Z). After the first character, objects names can contain alphabetic characters (a-z, A-Z), digits (0-9), underscores (_), and periods (.).

There are MANY conventions to choose from. Some common ones:

- `all_lower_with_underscores`
- `all.lower.with.periods`
- `CamelCase`

The key is consistency. And finding a balance between names that are short enough to type (repeatedly) quickly, but informative enough to be useful. (Some helpful advice on writing readable R code can be found here: http://adv-r.had.co.nz/Style.html)

Data frames

Data frames

For spreadsheet-like data, R uses data frames (`data.frame`).

Data frames are essentially collections of vectors of the same length where each vector forms a column of a table; these vectors *do not need to be the same class*.

Creating a data frame from vectors

A data frame can be constructed by combining several related vectors:

```
> iso3 <- c("CAN", "USA", "MEX")
> pop <- c(35.16, 318.9, 122.3)
> admin1 <- c(13L, 51L, 31L)
> spanish <- c(FALSE, FALSE, TRUE)
```

```
> df <- data.frame(iso3, pop, admin1, spanish)
> df
  iso3    pop admin1 spanish
1  CAN  35.16     13   FALSE
2  USA 318.90     51   FALSE
3  MEX 122.30     31    TRUE
> rm(iso3, pop, admin1, spanish)  # clean up the work space
```

Viewing a data frame

Like other data structures, you can view a data frame by printing it in the console:

```
> df
  iso3    pop admin1 spanish
1  CAN  35.16     13   FALSE
2  USA 318.90     51   FALSE
3  MEX 122.30     31    TRUE
```

Viewing a data frame

For larger data frames this is often cumbersome. . .

```
> data(airquality)  # load the airquality data
> airquality
    Ozone Solar.R Wind Temp Month Day
1      41     190  7.4   67     5   1
2      36     118  8.0   72     5   2
3      12     149 12.6   74     5   3
4      18     313 11.5   62     5   4
5      NA      NA 14.3   56     5   5
6      28      NA 14.9   66     5   6
7      23     299  8.6   65     5   7
8      19      99 13.8   59     5   8
9       8      19 20.1   61     5   9
10     NA     194  8.6   69     5  10
11      7      NA  6.9   74     5  11
12     16     256  9.7   69     5  12
13     11     290  9.2   66     5  13
14     14     274 10.9   68     5  14
15     18      65 13.2   58     5  15
16     14     334 11.5   64     5  16
17     34     307 12.0   66     5  17
```

Viewing a data frame

In this case, the `head()` and `tail()` functions are useful:

```
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

```
> tail(airquality, 3)
    Ozone Solar.R Wind Temp Month Day
151    14     191 14.3   75     9  28
152    18     131  8.0   76     9  29
153    20     223 11.5   68     9  30
```

Viewing a data frame

`nrow()`, `ncol()`, and `dim()` are useful for figuring out a data frame's size:

```
> nrow(airquality)
[1] 153
> ncol(airquality)
[1] 6
> dim(airquality)
[1] 153   6
```

and you can get a list of columns using `names()`:

```
> names(airquality)
[1] "Ozone"   "Solar.R" "Wind"    "Temp"    "Month"
[6] "Day"
```

Selecting columns

A single column can be selected *by number* or *by name*:

```
> df[, 2]
[1]  35.16 318.90 122.30
> df[, "pop"]
[1]  35.16 318.90 122.30
```

When using a name, R provides a convenient shorthand using $:

```
> df$pop
[1]  35.16 318.90 122.30
```

Selecting columns

A data frame column is a vector, so all of the operations that are valid for vectors are also valid for data frame columns:

```
> mean(df$pop)
[1] 158.7867
> log(df$pop)
[1] 3.559909 5.764878 4.806477
> df$pop/df$admin1
[1] 2.704615 6.252941 3.945161
> df$iso3 == "USA"
[1] FALSE  TRUE FALSE
> df$iso3[df$pop > 200]
[1] USA
Levels: CAN MEX USA
```

Creating columns

You can create a new column in a data frame using one of the assignment operators, similar to how you create a vector:

```
> df$gdp <- c(1827, 16770, 1261)
```

Similarly, you can create new columns that are functions of existing columns:

```
> df$log_gdp <- log(df$gdp)
> df$gdp_pc <- (1e+09 * df$gdp)/(1e+06 * df$pop)
```

```
> df
  iso3    pop admin1 spanish   gdp  log_gdp   gdp_pc
1  CAN  35.16     13   FALSE  1827 7.510431 51962.46
2  USA 318.90     51   FALSE 16770 9.727347 52587.02
3  MEX 122.30     31    TRUE  1261 7.139660 10310.71
```

Removing columns

You can remove a single column from a data frame by assigning it to NULL:

```
> df$log_gdp <- NULL
> df[, "gdp_pc"] <- NULL
```

```
> head(df)
  iso3       pop admin1 spanish       gdp
1  CAN  35160000     13   FALSE 1.827e+12
2  USA 318900000     51   FALSE 1.677e+13
3  MEX 122300000     31    TRUE 1.261e+12
```

Removing columns

Alternatively, you can subset to just the columns you want to keep and then reassign the object:

```
> airquality <- airquality[, c("Ozone", "Wind", "Month", "Day")]
```

```
> head(airquality)
  Ozone Wind Month Day
1    41  7.4     5   1
2    36  8.0     5   2
3    12 12.6     5   3
4    18 11.5     5   4
5    NA 14.3     5   5
6    28 14.9     5   6
```

Selecting rows

A subset of rows of a data frame can be selected *by index* (position):

```
> df[1, ]
  iso3      pop       gdp
1  CAN 35160000 1.827e+12
```

```
> df[2:3, ]
  iso3       pop       gdp
2  USA 318900000 1.677e+13
3  MEX 122300000 1.261e+12
```

```
> df[c(1, 3), ]
  iso3       pop       gdp
1  CAN  35160000 1.827e+12
3  MEX 122300000 1.261e+12
```

Selecting rows

. . . or by *logical statements*:

```
> df[c(T, F, F), ]
  iso3      pop       gdp
1  CAN 35160000 1.827e+12
```

```
> df[df$iso3 != "CAN", ]
  iso3       pop       gdp
2  USA 318900000 1.677e+13
3  MEX 122300000 1.261e+12
```

```
> df[df$gdp < 1e+13, ]
  iso3       pop       gdp
1  CAN  35160000 1.827e+12
3  MEX 122300000 1.261e+12
```

Removing rows

You can remove rows entirely by selecting just the ones you want to keep and then assigning or reassigning to an object:

```
> df <- df[df$gdp < 1e+13, ]
> df
  iso3       pop       gdp
1  CAN  35160000 1.827e+12
3  MEX 122300000 1.261e+12
```

```
> airquality_nomiss <- airquality[!is.na(airquality$Ozone), ]
> nrow(airquality)
[1] 153
> nrow(airquality_nomiss)
[1] 116
```

Things to keep in mind...

Both rows and columns can be selected using a `df[,]` notation.

In this framework, everything *before* the comma tells R what row(s) you want...

```
> df[df$pop > 1e+08, ]
  iso3       pop       gdp
3  MEX 122300000 1.261e+12
```

and everything *after* the comma tells R what column(s) you want:

```
> df[, "iso3"]
[1] CAN MEX
Levels: CAN MEX USA
```

Things to keep in mind...

If you want to store the output of these selections you must assign them to an object:

```
> big_pop <- df$iso3[df$pop > 1e+08]
```

```
> airquality_may <- airquality[airquality$Month == 5, ]
```

Note that the object you store your selection in might be either a vector or a data.frame, depending on the nature of your selection.

```
> class(big_pop)
[1] "factor"
> class(airquality_may)
[1] "data.frame"
```

Things to keep in mind...

(Re)assignment can also be used to store the output of your selections using the original object name:

```
> nrow(airquality)
[1] 153
> range(airquality$Month)
[1] 5 9
```

```
> airquality <- airquality[airquality$Month == 5, ]
```

```
> nrow(airquality)
[1] 31
> range(airquality$Month)
[1] 5 5
```

R vs Stata

Tools for data analysis, a comparison

Features	SPSS	SAS	Stata	JMP (SAS)	R	Python (Pandas)
Learning curve	Gradual	Pretty steep	Gradual	Gradual	Pretty steep	Steep
User interface	Point-and-click	Programming	Programming/ point-and-click	Point-and-click	Programming	Programming
Data manipulation	Strong	Very strong	Strong	Strong	Very strong	Strong
Data analysis	Very strong	Very strong	Very strong	Strong	Very strong	Strong
Graphics	Good	Good	Very good	Very good	Excellent	Good
Cost	Expensive (perpetual, cost only with new version). Student disc.	Expensive (yearly renewal) Free student version, 2014	Affordable (perpetual, cost only with new version). Student disc.	Expensive (yearly renewal) Student disc.	Open source (free)	Open source (free)
Released	1968	1972	1985	1989	1995	2008

NOTE: The R content presented in this document is mostly based on an early version of Fox, J. and Weisberg, S. (2011) *An R Companion to Applied Regression*, Second Edition, Sage; and from class notes from the ICPSR's workshop *Introduction to the R Statistical Computing Environment* taught by John Fox during the summer of 2010.

R	Stata
Working directory	

`getwd()  # Shows the working directory (wd)`
`setwd("C:/myfolder/data")  # Changes the wd`
`setwd("H:\\myfolder\\data") # Changes the wd`
	`pwd  /*Shows the working directory*/`
	`cd c:\myfolder\data  /*Changes the wd*/`
	`cd "c:\myfolder\stata data" /*Notice the spaces*/`

Installing packages/user-written programs	

`install.packages("ABC")  # This will install the package --ABC--. A window will pop-up, select a mirror site to download from (the closest to where you are) and click ok.`
 	
`library(ABC)  # Load the package --ABC-- to your workspace in R`
	`ssc install abc  /*Will install the user-defined program 'abc'. It will be ready to run.`
	
	`findit abc  /*Will do an online search for program 'abc' or programs that include 'abc'. It also searcher your computer.`

Getting help	

`?plot # Get help for an object, in this case for the --plot- function. You can also type: help(plot)`
 	
`??regression # Search the help pages for anything that has the word "regression". You can also type: help.search("regression")`
 	
`apropos("age")  # Search the word "age" in the objects available in the current R session.`
 	
`help(package=car)  # View documentation in package 'car'. You can also type: library(help="car")`
 	
`help(DataABC)  # Access codebook for a dataset called 'DataABC' in the package ABC`
	`help tab  /* Get help on the command 'tab'*/`
	
	`search regression  /* Search the keywords for the word 'regression'*/`
	
	`hsearch regression  /* Search the help files for the work 'regression'. It provides more options than 'search'*/`

6

R	Stata
Data from *.csv	

`# Reading the data directly`
 	
`mydata <- read.csv("c:\mydata\mydatafile.csv", header=TRUE)`
 	
`# The will open a window to search for the *.csv file.`
 	
`mydata <- read.csv(file.choose(), header = TRUE)`
	`/* In the command line type */`
	
	`insheet using "c:\mydata\mydatafile.csv"`
	
	`/* Using the menu */`
	
	`Go to File->Import->"ASCII data created by spreadsheet". Click on 'Browse' to find the file and then OK.`

Data from/to Stata	

`library(foreign) # Load package --foreign--`
 	
`mydata <- read.dta("http://dss.princeton.edu/training/students.dta")`
 	
`mydata.dta <- read.dta("http://dss.princeton.edu/training/mydata.dta", convert.factors=TRUE, convert.dates=TRUE, convert.underscore=TRUE, warn.missing.labels=TRUE)`
	`/* To open a Stata file go to File -> Open, or type: */`
	
	`use "c:\myfolder\mydata.dta"`
	
	`Or`
	
	`use "http://dss.princeton.edu/training/mydata.dta"`
	
	`/* If you need to load a subset of a Stata data file type */`
	
	`use var1 var2 using "c:\myfolder\mydata.dta"`

Exploring data

R

```
str(mydata)       # Provides the structure of the
                   dataset
summary(mydata)   # Provides basic descriptive
                   statistics and frequencies
names(mydata)     # Lists variables in the dataset
head(mydata)      # First 6 rows of dataset
head(mydata, n=10)# First 10 rows of dataset
head(mydata, n= -10)  # All rows but the last 10
tail(mydata)      # Last 6 rows
tail(mydata, n=10)    # Last 10 rows
tail(mydata, n= -10)  # All rows but the first 10
mydata[1:10, ]    # First 10 rows of the
mydata[1:10,1:3] # First 10 rows of data of the
                   first 3 variables
edit(mydata)      # Open data editor
```

Stata

```
describe      /* Provides the structure of the
                 dataset*/
summarize     /* Provides basic descriptive
                 statistics for numeric data*/
ds            /* Lists variables in the dataset */
list in 1/6   /* First 6 rows */
edit          /* Open data editor (double-click to
                 edit*/
browse        /* Browse data */
```

Missing data

R

```
sum(is.na(mydata))# Number of missing in dataset
rowSums(is.na(data))# Number of missing per
variable
rowMeans(is.na(data))*length(data)# No. of missing
                  per row
mydata[mydata$age=="&   ","age"] <- NA    # NOTE:
     Notice hidden spaces.
mydata[mydata$age==999,"age"] <- NA
The function complete.cases() returns a logical vector
indicating which cases are complete.
# list rows of data that have missing values
mydata[!complete.cases(mydata),]
The function na.omit() returns the object with listwise
deletion of missing values.
# create new dataset without missing data
newdata <- na.omit(mydata)
```

Stata

```
tabmiss      /* # of missing. Need to install, type
                scc install tabmiss. Also try findit
                tabmiss and follow instructions */

/* For missing values per observation see the
                function 'rowmiss' and the 'egen'
                command*/
```

R	Stata
Descriptive Statistics	

```
mean(mydata) # Mean of all numeric variables, same
using --sapply--('s' for simplify)
mean(mydata$SAT)
with(mydata, mean(SAT))
median(mydata$SAT)

table(mydata$Country) # Mode by frequencies ->
max(table(mydata$Country)) / names(sort(-
table(mydata$Country)))[1]

var(mydata$SAT)   # Variance
sd(mydata$SAT)   # Standard deviation
max(mydata$SAT)   # Max value
min(mydata$SAT)   # Min value
range(mydata$SAT)   # Range
quantile(mydata$SAT)
quantile(mydata$SAT, c(.3,.6,.9))
fivenum(mydata$SAT)   # Boxplot elements. From
   help: "Returns Tukey's five number summary
   (minimum, lower-hinge, median, upper-hinge,
   maximum) for the input data ~ boxplot"
length(mydata$SAT)    # Num of observations when a
                      variable is specify
length(mydata)        # Number of variables when a
                  dataset is specify
which.max(mydata$SAT) # From help: "Determines the
   location, i.e., index of the (first) minimum or
   maximum of a numeric vector"
which.min(mydata$SAT) # From help: "Determines the
   location, i.e., index of the (first) minimum or
   maximum of a numeric vector"
stderr <- function(x) sqrt(var(x)/length(x))
incster <- tapply(incomes, statef, stderr)
```

```
summarize   /*N, mean, sd, min, max*/
summarize, detail /*N, mean, sd, min, max,
            variance, skewness, kurtosis,
            percentiles*/
summarize age, detail
summarize sat, detail
tabstat  age sat score heightin read  /*Gives the
            mean only*/
tabstat  age sat score heightin read,
            statistics(n, mean, median, sd, var,
            min, max)
/*Type help tabstat for a list of all statistics*/

tabstat  age sat score heightin read, by(gender)
tabstat  age sat score heightin read,
            statistics(mean, median) by(gender)

table gender, contents(freq mean age mean score)

tab gender major, sum(sat)   /*Categorical and
            continuous*/

bysort studentstatus: tab gender major, sum(sat)
```

Introducing spatial data in R

Spatial data packages

Packages we'll use today:

- **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
- **tmap**: a new package for rapidly creating beautiful maps
- **leaflet:** simple, fast way to host interactive maps

Others:

- **ggmap**, **raster**, **maptools**, and many more!

Spatial data in R: shapefiles

- **Shapefile**: a file format for storing information on geographic features like cities, countries, and more. Can be represented by points, lines, or polygons

- Made up of a number of different files such as *.prj*, *.dbf*, *.shp*

Many can be downloaded online for free

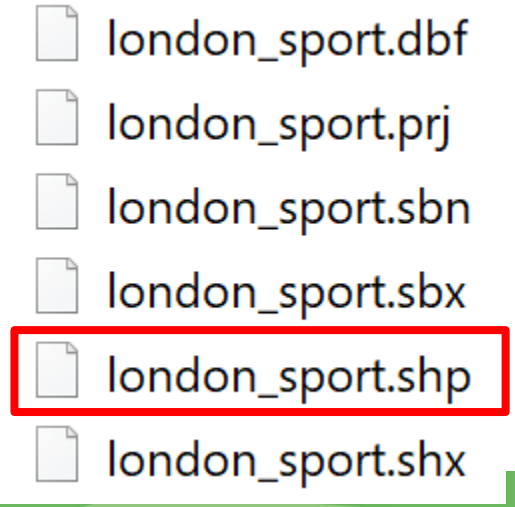
Greater London area

Spatial data in R: shapefiles

- **Shapefile**: a file format for storing information on geographic features like cities, countries, and more. Can be represented by points, lines, or polygons

- Made up of a number of different files such as *.prj*, *.dbf*, *.shp*

Many can be downloaded online for free



Main file, stores feature geometry

Reading shapefiles into R

readOGR()

- *dsn* – data source name; the folder the file is saved in
- *layer* – the file name (no need to specify the *.shp* file extension)

Information on the population of London Boroughs in 2001 and the percentage of the population participating in sporting activities

```
library(rgdal)

lnd <- readOGR(dsn = "data", layer = "london_sport")
```

The structure of spatial data

Inputting the object name (*lnd*) into the console prints a summary

CRS (Coordinate Reference System) says where spatial objects are in the world. Spatial data should always have a CRS.

More on this later

Greater London area

```
> lnd
class       : SpatialPolygonsDataFrame
features    : 33
extent      : 503571.2, 561941.1, 155850.8, 200932.5  (xmin, xmax, ymin, ymax)
crs         : +proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000 +y_0=-100000 +ellps=airy +units=m +no_defs
variables   : 4
names       : ons_label,                 name, Partic_Per, Pop_2001
min values  :      00AA, Barking and Dagenham,        9.1,        1
max values  :      00BK,          Westminster,       28.4,       33
```

The structure of spatial data

Spatial objects like *lnd* are made up of a number of different **slots:**

- **@data** – non-geographic attribute data aka the information we want to graph
- **@polygons**, **@lines** – make up the physical boundaries
- Slots are accessed using the @ symbol

```
> head(lnd@data)
  ons_label                    name Partic_Per Pop_2001
0      00AF                 Bromley       21.7   295535
1      00BD Richmond upon Thames       26.6   172330
2      00AS              Hillingdon       21.5   243006
3      00AR                Havering       17.9   224262
4      00AX Kingston upon Thames       24.4   147271
5      00BF                  Sutton       19.3   179767
> mean(lnd$Partic_Per) # short for mean(lnd@data$Partic_Per)
[1] 20.05455
```

View first few rows

The structure of spatial data

- We can explore the @**data** slot as if it were any other *data.frame*

```
> sapply(lnd@data, class)
 ons_label       name Partic_Per   Pop_2001
  "factor"   "factor"  "numeric"  "numeric"
> nrow(lnd)
[1] 33
> ncol(lnd)
[1] 4
```

→ What are the data types of the columns?

Basic plotting

```
# basic plotting
plot(lnd)


# Select zones where sports participation
# is between 20 and 25%
subset <- lnd$Partic_Per > 20 & lnd$Partic_Per < 25
plot(lnd[subset, ])


# layered plot
plot(lnd, col = "lightgrey")
plot(lnd[subset, ], col = "turquoise", add = TRUE)
```

Basic plotting

```
# basic plotting
plot(lnd)


# Select zones where sports participation
# is between 20 and 25%
subset <- lnd$Partic_Per > 20 & lnd$Partic_Per < 25
plot(lnd[subset, ])


# layered plot
plot(lnd, col = "lightgrey")
plot(lnd[subset, ], col = "turquoise", add = TRUE)
```

Basic plotting

```
# basic plotting
plot(lnd)


# Select zones where sports participation
# is between 20 and 25%
subset <- lnd$Partic_Per > 20 & lnd$Partic_Per < 25
plot(lnd[subset, ])


# layered plot
plot(lnd, col = "lightgrey")
plot(lnd[subset, ], col = "turquoise", add = TRUE)
```

Basic plotting

```
# basic plotting
plot(lnd)


# Select zones where sports participation
# is between 20 and 25%
subset <- lnd$Partic_Per > 20 & lnd$Partic_Per < 25
plot(lnd[subset, ])


# layered plot
plot(lnd, col = "lightgrey")
plot(lnd[subset, ], col = "turquoise", add = TRUE)
```

Spatial data: dealing with coordinates

- **coordinates()** – set or retrieve spatial coordinates for a Spatial object. Returns a *matrix* (similar to a *data.frame*)

```
> head(coordinates(lnd))
      [,1]     [,2]
0 542916.5 165647.4
1 517372.0 172874.7
2 507892.4 183654.6
3 554011.5 187283.0
4 519267.5 166816.1
5 526941.7 164153.1
> head(coordinates(lnd)[, 1])
       0        1        2        3        4        5
542916.5 517372.0 507892.4 554011.5 519267.5 526941.7
```

latitude, longitude

Select just latitude

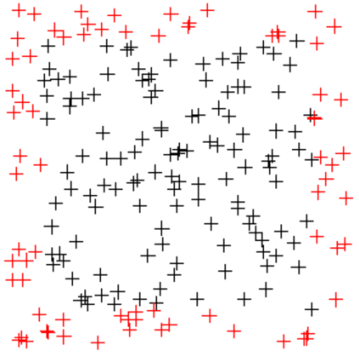
Spatial data: dealing with coordinates

```r
# Find the centre of the london area
lat <- coordinates(gCentroid(lnd))[[1]]
lng <- coordinates(gCentroid(lnd))[[2]]

# arguments to test whether or not a coordinate is east or north of the centre
east <- sapply(coordinates(lnd)[,1], function(x) x > lat)
north <- sapply(coordinates(lnd)[,2], function(x) x > lng)

# test if the coordinate is east and north of the centre
lnd@data$quadrant <- ""
lnd@data$quadrant[east & north] <- "northeast"


plot(lnd, col = "lightgrey") # plot the london_sport object
subset <- lnd$quadrant == "northeast"
plot(lnd[subset, ], col = "turquoise", add = TRUE) # add selected zones to map
```

Creating and manipulating spatial data

- **SpatialPoints()** – creates a spatial points object from a *data.frame* or a *matrix*
- **SpatialPointsDataFrame()** – creates a spatial points object with an associated *data.frame*

```
df <- data.frame(x = runif(200, 0, 1), y = runif(200, 0, 1))
sp1 <- SpatialPoints(coords = df)
spdf <- SpatialPointsDataFrame(sp1, data = df)

plot(spdf)
subset <- spdf$x > .9 | spdf$x < .1 | spdf$y > .9 | spdf$y < .1
plot(spdf[subset, ], col = "red", add = TRUE)
```



Creating and manipulating spatial data

- **SpatialPoints()** – creates a spatial points object from a *data.frame* or a *matrix*
- **SpatialPointsDataFrame()** – creates a spatial points object with an associated *data.frame*

```
> spdf
class       : SpatialPointsDataFrame
features    : 200
extent      : 0.005603223, 0.9964228, 0.001488753, 0.9858649  (xmin, xmax, ymin, ymax)
crs         : NA
variables   : 2
names       :                   x,                   y
min values  : 0.00560322310775518, 0.00148875289596617
max values  :    0.99642278579995,   0.985864923568442
> head(spdf@data)
          x          y
1 0.88288692 0.02915883
2 0.30995625 0.11639213
3 0.15914881 0.24334859
4 0.75013789 0.87596064
5 0.07227254 0.97365393
6 0.49872466 0.56085314
```

Manipulating spatial data

Coordinate Reference System (CRS)

“A location of (140, 12) is not meaningful if you do know where the origin is and if the x-coordinate is 140 meters, kilometers, or perhaps degrees away”

The most common system is the World Geodesic System 1984 (**WGS84**), which is essentially a model of the shape of the Earth

The most common CRSs are assigned a **EPSG** code (European Petroleum Survey Group).

- Unique ID that is a simple way to identify a CRS

Coordinate Reference System (CRS) example

EPSG	CRS
EPSG:27561	+proj=lcc +lat_1=49.5 +lat_0=49.5 +lon_0=0 +k_0=0.999877341 +x_0=6 +y_0=2 +a=6378249.2 +b=6356515 +towgs84=-168,-60,320,0,0,0,0 +pm=paris +units=m +no_defs

Setting and transforming CRS in R

- **proj4string()** – Set or retrieve the CRS of a spatial object

```
> proj4string(lnd) <- NA_character_ # remove CRS information from lnd
> proj4string(lnd) <- CRS("+init=epsg:27700") # assign a new CRS
> lnd
class       : SpatialPolygonsDataFrame
features    : 33
extent      : 503571.2, 561941.1, 155850.8, 200932.5  (xmin, xmax, ymin, ymax)
crs         : +init=epsg:27700 +proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x
s84=446.448,-125.157,542.060,0.1502,0.2470,0.8421,-20.4894
variables   : 5
names       : ons_label,                  name, Partic_Per, Pop_2001,  quadrant
min values  :      00AA, Barking and Dagenham,        9.1,        1,
max values  :      00BK,          Westminster,       28.4,       33, northeast
```

Setting and transforming CRS in R

- **proj4string()** – Set or retrieve the CRS of a spatial object

27700 represents the British National Grid

```
> proj4string(lnd) <- NA_character_ # remove CRS information from lnd
> proj4string(lnd) <- CRS("+init=epsg:27700") # assign a new CRS
> lnd
class       : SpatialPolygonsDataFrame
features    : 33
extent      : 503571.2, 561941.1, 155850.8, 200932.5  (xmin, xmax, ymin, ymax)
crs         : +init=epsg:27700 +proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +
s84=446.448,-125.157,542.060,0.1502,0.2470,0.8421,-20.4894
variables   : 5
names       : ons_label,                 name, Partic_Per, Pop_2001,  quadrant
min values  :      00AA, Barking and Dagenham,        9.1,        1,
max values  :      00BK,          Westminster,       28.4,       33, northeast
```

Setting and transforming CRS in R

Converting *lnd*'s CRS to the more widely used **WGS84** using **spTransform()**

```
EPSG <- make_EPSG() # create data frame of available EPSG codes
EPSG[grepl("WGS 84$", EPSG$note), ] # search for WGS 84 code

lnd84 <- spTransform(lnd, CRS("+init=epsg:4326")) # reproject

# Save lnd84 object
saveRDS(object = lnd84, file = "data/lnd84.Rds")
```

Console:

```
> EPSG[grepl("WGS 84$", EPSG$note), ] # search for WGS 84 code
     code     note                                                  prj4
249  4326 # WGS 84                    +proj=longlat +datum=WGS84 +no_defs
5311 4978 # WGS 84 +proj=geocent +datum=WGS84 +units=m +no_defs
```

Joining data sets to spatial data

Read in London crime data, filter out all but “Theft and Handling” crimes, and aggregate the total number per borough

```r
# Create and look at new crime_data object
crime_data <- read.csv("data/mps-recordedcrime-borough.csv",
                       stringsAsFactors = FALSE)

# Extract "Theft & Handling" crimes and save
crime_theft <- crime_data[crime_data$CrimeType == "Theft & Handling", ]

# Calculate the sum of the crime count for each district, save result
crime_ag <- aggregate(CrimeCount ~ Borough, FUN = sum, data = crime_theft)
# Show the first two rows of the aggregated crime data
head(crime_ag)
```

```
> head(crime_ag)
                Borough CrimeCount
1 Barking and Dagenham      12222
2                Barnet      19821
3                Bexley       8155
4                 Brent      16823
5               Bromley      15172
6                Camden      36493
```

Joining data sets to spatial data

Now that our data set is ready, we can join it to our spatial object *lnd* using **left_join()**

- *by* specifies what column(s) the two data sets should be joined using. Since the column we want to join on is named differently in the two *data.frame*s, we use this syntax

```
library(dplyr)

lnd@data <- left_join(lnd@data, crime_ag, by = c('name' = 'Borough'))
head(lnd@data)
```

```
> head(lnd@data)
  ons_label                 name Partic_Per Pop_2001 CrimeCount
1      00AF              Bromley       21.7   295535      15172
2      00BD Richmond upon Thames       26.6   172330       9715
3      00AS           Hillingdon       21.5   243006      15302
4      00AR             Havering       17.9   224262      12611
5      00AX Kingston upon Thames       24.4   147271       9023
6      00BF               Sutton       19.3   179767       8810
```

Joining data sets to spatial data

Now that our data set is ready, we can join it to our spatial object *lnd* using **left_join()**

- *by* specifies what column(s) the two data sets should be joined using. Since the column we want to join on is named differently in the two *data.frame*s, we use this syntax

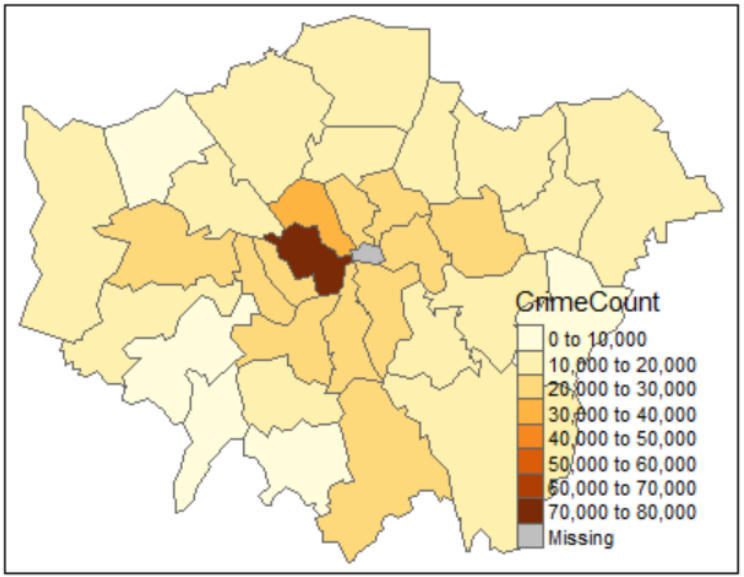
```
library(dplyr)

lnd@data <- left_join(lnd@data, crime_ag, by = c('name' = 'Borough'))
head(lnd@data)
```

```
> head(lnd@data)
  ons_label                 name Partic_Per Pop_2001 CrimeCount
1      00AF              Bromley       21.7   295535      15172
2      00BD Richmond upon Thames       26.6   172330       9715
3      00AS           Hillingdon       21.5   243006      15302
4      00AR             Havering       17.9   224262      12611
5      00AX Kingston upon Thames       24.4   147271       9023
6      00BF               Sutton       19.3   179767       8810
```

Joining data sets to spatial data: plotting

- **qtm()** – create a **q**uick **t**hematic **m**ap
  - The *fill* argument allows you to color the map based on a column's values

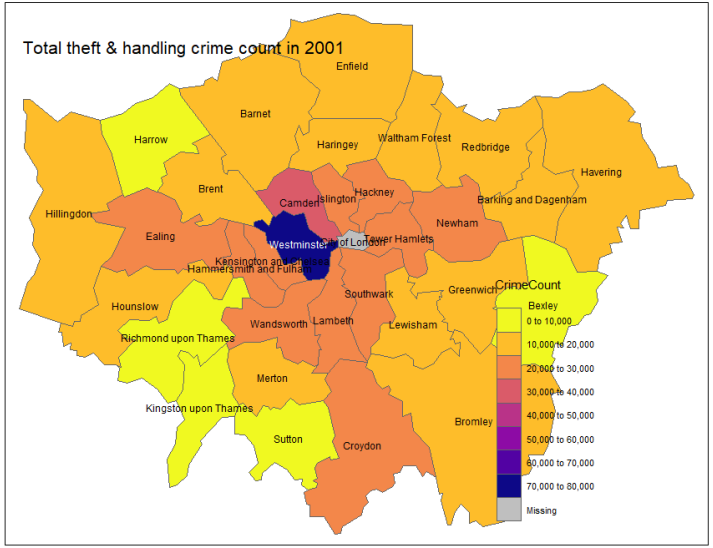
```
library(tmap)
qtm(lnd, fill = "CrimeCount")
```



Joining data sets to spatial data: plotting

- **qtm()** – create a **q**uick **t**hematic **m**ap
  - The *fill* argument allows you to color the map based on a column's values
  - Many other parameters!

```
library(tmap)
qtm(lnd, fill = "CrimeCount", text = "name", text.size = 0.75,
    title = "Total theft & handling crime count in 2001",
    fill.palette = "-plasma")
```



Spatial data in R using *ggplot2* and *leaflet*

Spatial data in *ggplot2*

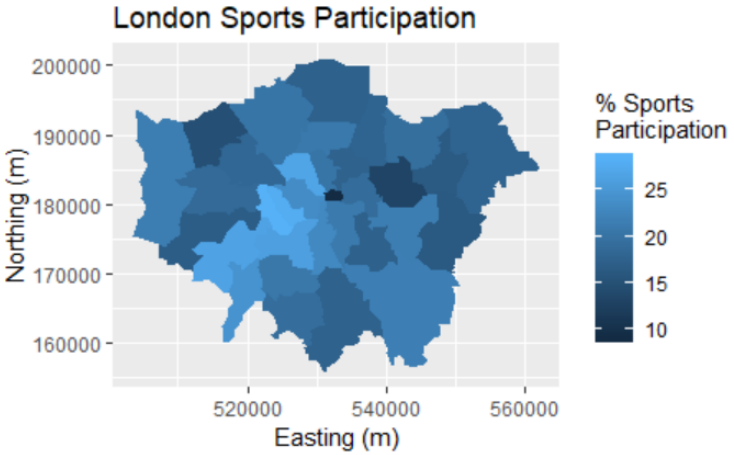
plot() can handle the Spatial objects we've been working with; *ggplot2* cannot so we have a few extra steps

- **fortify()** – extracts the spatial data into a *data.frame*
- We then need to rejoin our data using **left_join()** after we create a variable to merge on

```
library(rgeos)
lnd_f <- fortify(lnd)
lnd$id <- row.names(lnd) # allocate an id variable to the sp data
lnd_f <- left_join(lnd_f, lnd@data, by = "id") # join the data
```

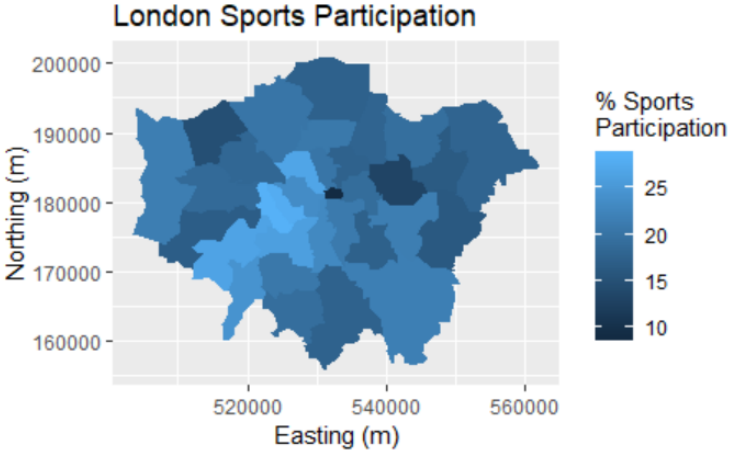
Spatial data in *ggplot2*

```
ggplot(lnd_f, aes(long, lat, group = group, fill = Partic_Per)) +
  geom_polygon() + coord_equal() +
  labs(x = "Easting (m)", y = "Northing (m)",
       fill = "% Sports\nParticipation") +
  ggtitle("London Sports Participation")
```

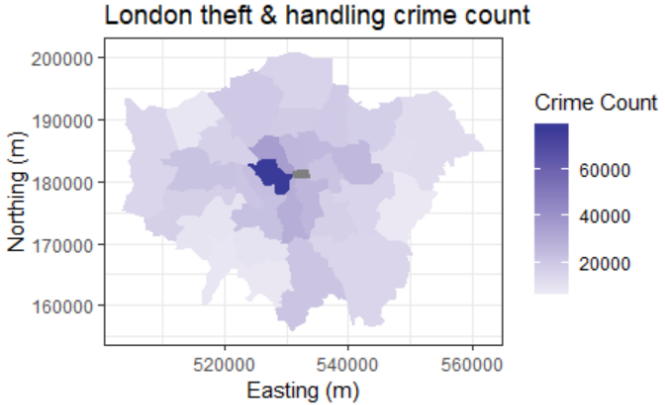


Spatial data in *ggplot2*

```
ggplot(lnd_f, aes(long, lat, group = group, fill = Partic_Per)) +
  geom_polygon() + coord_equal() +
  labs(x = "Easting (m)", y = "Northing (m)",
       fill = "% Sports\nParticipation") +
  ggtitle("London Sports Participation")
```




```
ggplot(lnd_f, aes(long, lat, group = group, fill = CrimeCount)) +
  geom_polygon() + coord_equal() +
  scale_fill_gradient2() +
  labs(x = "Easting (m)", y = "Northing (m)",
       fill = "Crime Count") +
  ggtitle("London theft & handling crime count") +
  theme_bw()
```

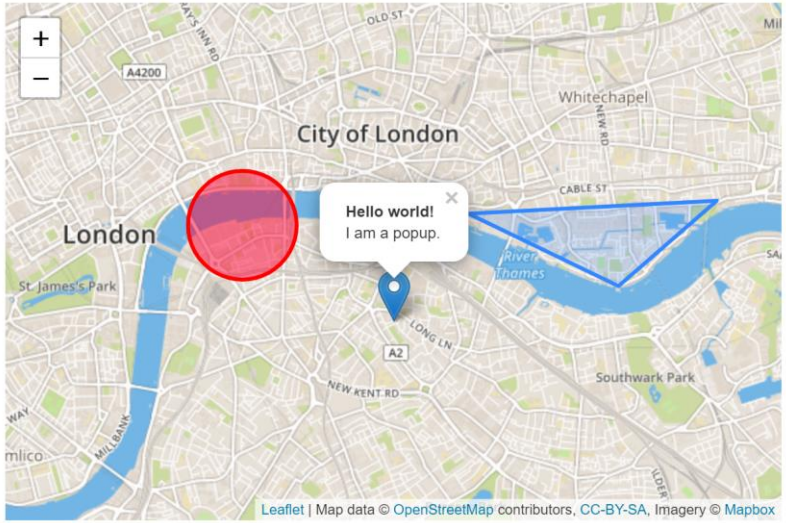


Spatial data in *leaflet*

- *leaflet* is a powerful web mapping system built on a JavaScript library, making it easy to create interactive maps with only a few lines of code
  - Pairs nicely with *shiny*, an R package for creating interactive online visualizations

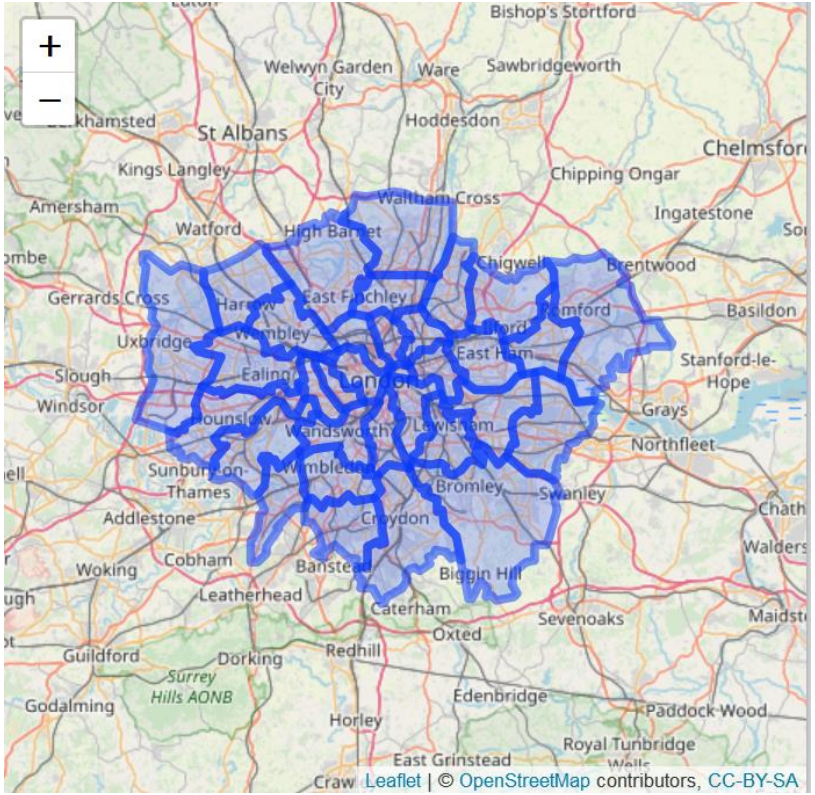
See:

- github.com/Leaflet/Leaflet
- rstudio.github.io/leaflet/

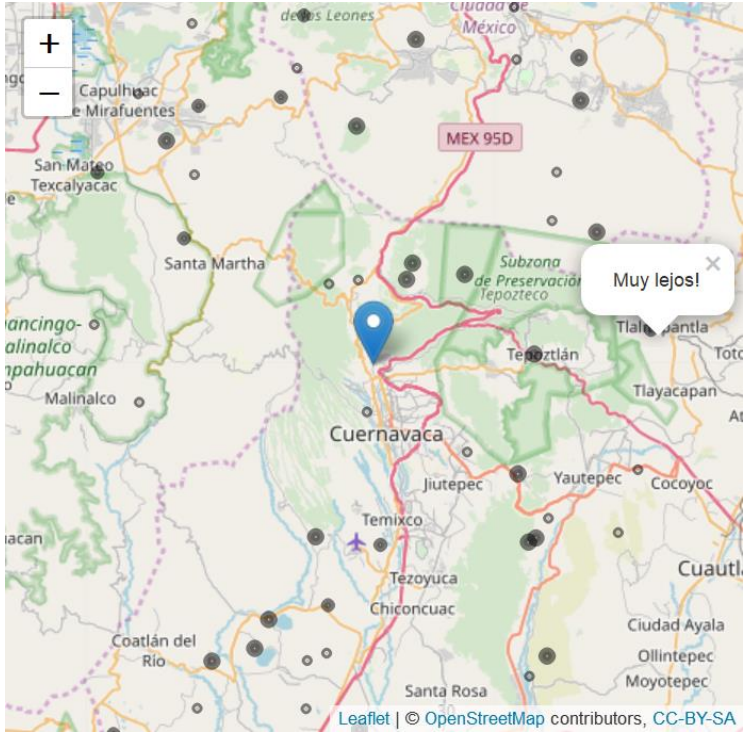


leaflet: getting started

```
library(leaflet)

lnd84 <- readRDS('data/lnd84.Rds')

leaflet() %>%
  addTiles() %>%
  addPolygons(data = lnd84)
```



leaftlet




```
m <- leaflet() %>% addTiles()
(m2 <- m %>%
    setView(lng = -99.246, lat = 18.977, zoom = 10) %>% # map location
    addMarkers(-99.246, 18.977) %>% # add a marker
    addPopups(-99, 19, popup = "Muy lejos!") %>% # popup
    # add som circles:
    addCircles(color = "black", runif(90, -99.5, -99), runif(90, 18.5, 19.5), runif(90, 10, 500)))
```