Mapping & Population Analyis with R

Dr. Jeffrey Strickland

8/28/2018

Introduction

> *"I knew not what wild beast we were about to hunt down in the dark jungle of criminal London, but I was well assured from the bearing of this master huntsman that the adventure was a most grave one, while the sardonic smile which occasionally broken through his ascetic gloom boded little good for the object of our quest."*
>
> —Dr. John Watson, The Adventure of the Speckled Band

Many problems faced by businesses, healthcare, education, and government invlove populations and their geographic dispositions. Today's nonhomogeneous society make data-based studies more complex than they were in less recent years. Examples include studies of epidemics, crime analysis, politcal polling, community service, and so on.

Preliminaries

R offers a variety of functionality to perform mapping and population studies. Some of these packages are used in this module and are described below.

Package Descriptions

- 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
- install.packages(x)
- install.packages(c("rgdal", "maptools", "dplyr", "tidyr", "tmap", "tmaptools", "rgeos"))

Create a New Project

- Starts a new project entitled “Creating-maps-in-R” project using File -> New Project... on the top menu
- Start a new R Script using File -> New File... -> R Script
- Type all your commands in the R Script
- Load the libraries:

```
library(maptools)

## Loading required package: sp

## Checking rgeos availability: TRUE

library(dplyr)

##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union

library(tidyr)
library(tmap)
library(rgdal)

## rgdal: version: 1.3-4, (SVN revision 766)
##  Geospatial Data Abstraction Library extensions to R successfully loaded
##  Loaded GDAL runtime: GDAL 2.2.3, released 2017/11/20
##  Path to GDAL shared files: C:/Users/jeff/Documents/R/win-
library/3.5/rgdal/gdal
##  GDAL binary built with GEOS: TRUE
##  Loaded PROJ.4 runtime: Rel. 4.9.3, 15 August 2016, [PJ_VERSION: 493]
##  Path to PROJ.4 shared files: C:/Users/jeff/Documents/R/win-
library/3.5/rgdal/proj
##  Linking to sp version: 1.3-1

library(rgeos)

## rgeos version: 0.3-28, (SVN revision 572)
##  GEOS runtime version: 3.6.1-CAPI-1.10.1 r0
##  Linking to sp version: 1.3-1
##  Polygon checking: TRUE
```

Load the London Sport Data Using readOGR

The first file we are going to load into R Studio is the “london_sport” shapefile located in the “data” folder of the project. The readOGR function is used to load a shapefile and assign it to a new spatial object called “lnd”; short for London. readOGR is a function which accepts two arguments: - dsn which stands for “data source name” and specifies the directory in which the file is stored - layer which specifies the file name (there is no need to include the file extention .shp) The lnd object contains the population of London Boroughs in 2001 and the percentage of the population participating in sporting activities.

```
library(rgdal)
mydir<-"C:/Users/jeff/Documents/Crime Analysis/Creating-maps-in-R-master/data"
lnd <- readOGR(dsn = mydir, layer = "london_sport")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "C:\Users\jeff\Documents\Crime Analysis\Creating-maps-in-R-master\data", layer: "london_sport"
## with 33 features
## It has 4 fields
## Integer64 fields read as strings:  Pop_2001
```

```
lnd_b <- readOGR(dsn = mydir, layer = "LondonBoroughs")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "C:\Users\jeff\Documents\Crime Analysis\Creating-maps-in-R-master\data", layer: "LondonBoroughs"
## with 33 features
## It has 8 fields
## Integer64 fields read as strings:  Pop_2001 PopDensity PopDen
```

```
lnd_s <- readOGR(dsn = mydir, layer = "ukbord")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "C:\Users\jeff\Documents\Crime Analysis\Creating-maps-in-R-master\data", layer: "ukbord"
## with 1 features
## It has 1 fields
```

Explore the Output

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 lnd object. - The $ symbol refers to a column (a variable within the table) in the data slot The head function in the first line of the code above simply means “show the first few lines of data.” The second line calculates finds the mean sports participation per 100 people for zones in London.

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

```
##   ons_label                  name Partic_Per Pop_2001
## 0      00AF              Bromley       21.7   295535
## 1      00BD Richmond upon Thames       26.6   172330
```

```r
mean(lnd$Partic_Per) # short for mean(lnd@data$Partic_Per)
```

```
## [1] 20.05455
```

Show the Data in a Table

```r
library(DT)
datatable(head(lnd@data), options = list(scrollX='400px'))
```

Show 10 entries Search:

	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

Showing 1 to 6 of 6 entries Previous 1 Next

Check the Classes in the Data Slot

To check the classes of all the variables in a spatial dataset, you can use the following command:

```r
sapply(lnd@data, class)
```

```
##  ons_label       name Partic_Per   Pop_2001
##   "factor"   "factor"  "numeric"   "factor"
```

This shows that, unexpectedly, Pop_2001 is a factor. We can coerce the variable into the correct, numeric, format with the following command:

```r
lnd$Pop_2001 <- as.numeric(as.character(lnd$Pop_2001))
```

Further Exploration

To explore lnd object further, try typing

```
nrow(lnd)

## [1] 33

ncol(lnd)

## [1] 4

lnd@proj4string

## CRS arguments:
##  +proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000
## +y_0=-100000 +ellps=airy +units=m +no_defs
```

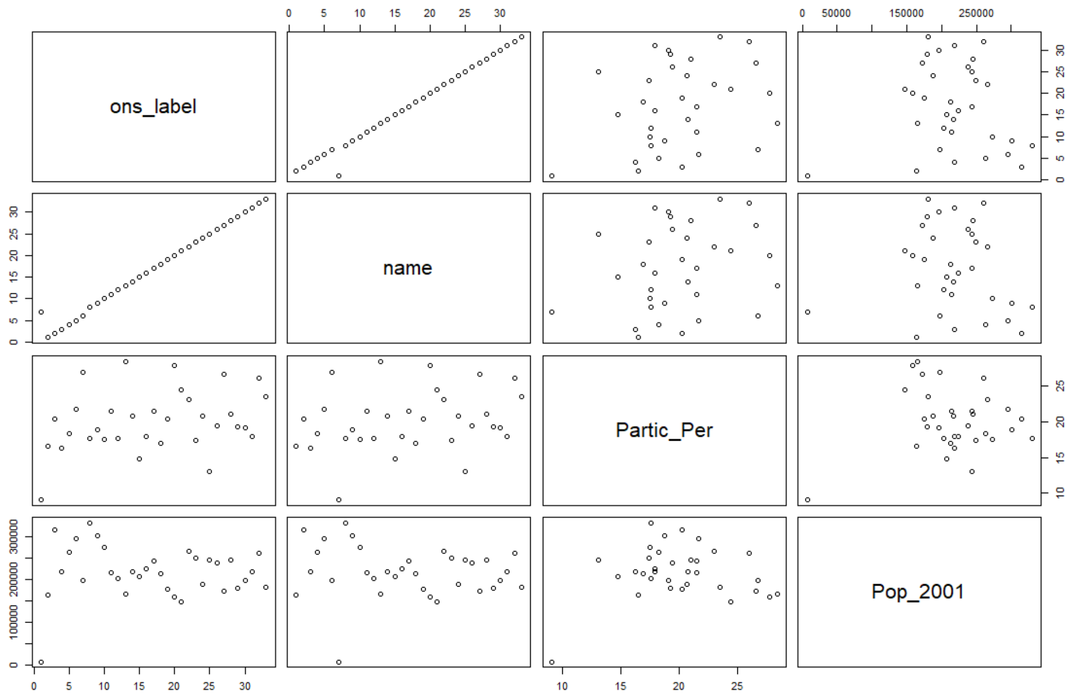
Visualize the Data

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(lnd)
```

Inputting another object such as plot(lnd@data) will generate an entirely difierent type of plot.

```
plot(lnd@data)
```



Square Brackets

R has powerful subsetting capabilities that can be accessed very concisely using square brackets. Select rows of lnd@data where sports participation is less than 15

```
lnd@data[lnd$Partic_Per < 15, ]

##    ons_label           name Partic_Per Pop_2001
## 17      00AQ         Harrow       14.8   206822
## 21      00BB         Newham       13.1   243884
## 32      00AA City of London        9.1     7181
```

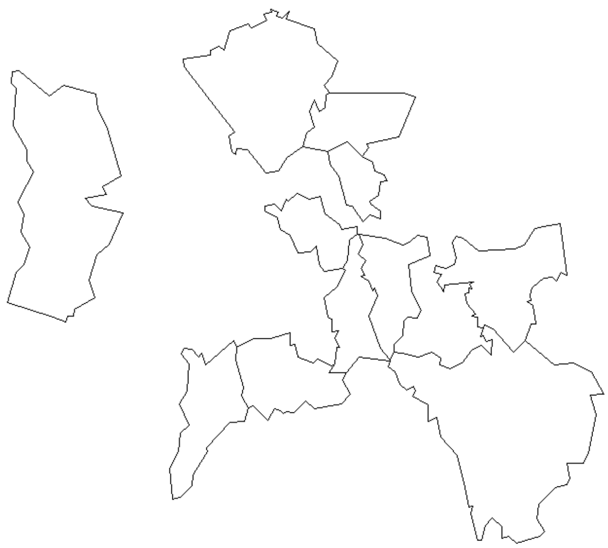
Comment on Code

The above line of code asked R to select only the rows from the lnd object, where sports participation is lower than 15, in this case rows 17, 21 and 32, which are Harrow, Newham and the city centre 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 lnd@data[1:2, 1:3] and test it).

The Geometry Slot

So far we have been interrogating only the attribute data slot (@data) of the lnd 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 zones with high sports participation. Select zones where sports participation is between 20 and 25%

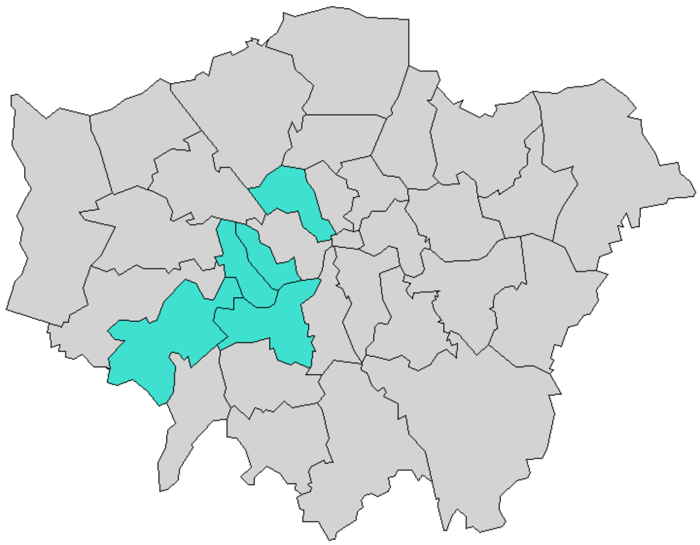
```
sel <- lnd$Partic_Per > 20 & lnd$Partic_Per < 25
plot(lnd[sel, ])
```



```
head(sel)

## [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

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(lnd, col = "lightgrey")
sel <- lnd$Partic_Per > 25
plot(lnd[sel, ], col = "turquoise", add = TRUE)
```



Geographic Centroids

We now find the geographic centroid of Lodon, which we will use in subsequent steps to divide the city into divisions, like quadrants.

Geographic Centroids - Method 1

```
library(rgeos)
plot(lnd, col = "grey")
cent_lnd <- gCentroid(lnd[lnd$name == "City of London",])
points(cent_lnd, cex = 3)
```

Method 1:

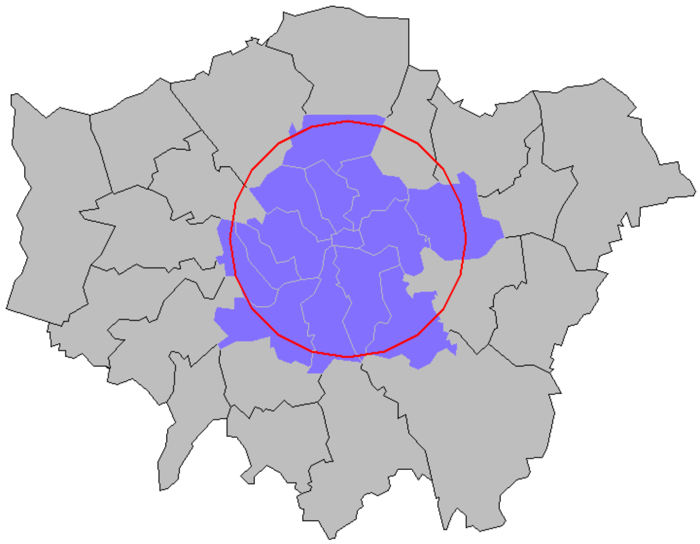
```
lnd_buffer <- gBuffer(spgeom = cent_lnd, width = 10000)
```

Geographic Centroids - Method 2

Method 2 of subsetting selects only points within the buffer:

```
lnd_cents <- SpatialPoints(coordinates(lnd),
                           proj4string = CRS(proj4string(lnd))) # create
spatialpoints
sel <- lnd_cents[lnd_buffer,] # select points inside buffer
points(sel) # show where the points are located
lnd_central <- lnd[sel,] # select zones intersecting w. sel
plot(lnd_central, add = T, col = "lightslateblue",
     border = "grey")
plot(lnd_buffer, add = T, border = "red", lwd = 2)
```

Plot the Centroids



Selecting quadrants

The code below should help understand the way spatial data work in R. It is used to find the centre of the london area

```
lat <- coordinates(gCentroid(lnd))[[1]]
lng <- coordinates(gCentroid(lnd))[[2]]
```

Test for NE Quadrant Inclusion

The following comprise 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[east & north] <- "northeast"
```

Test for SE Quadrant Inclusion

```
east <- sapply(coordinates(lnd)[,1], function(x) x > lat)
south <- sapply(coordinates(lnd)[,2], function(x) x < lng)

lnd@data$quadrant[east & south] <- "southeast"
```

Test for NW Quadrant Inclusion

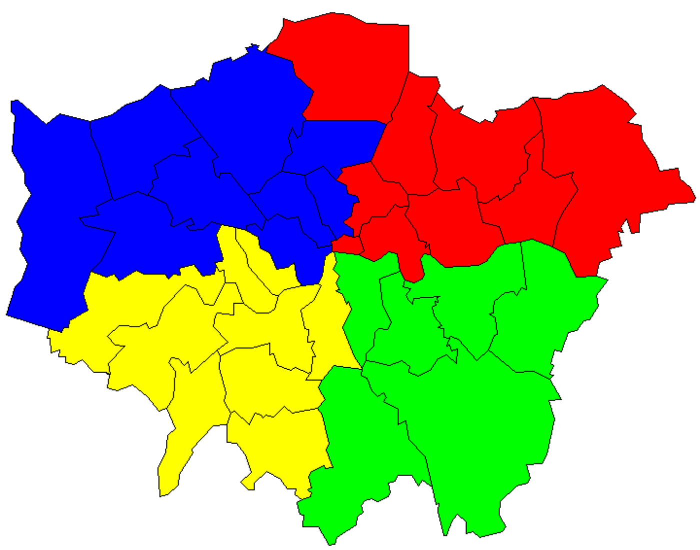
```
west <- sapply(coordinates(lnd)[,1], function(x) x < lat)
north <- sapply(coordinates(lnd)[,2], function(x) x > lng)

lnd@data$quadrant[west & north] <- "northwest"
```

Test for SW Quadrant Inclusion

```
west <- sapply(coordinates(lnd)[,1], function(x) x < lat)
south <- sapply(coordinates(lnd)[,2], function(x) x < lng)

lnd@data$quadrant[west & south] <- "southwest"
```

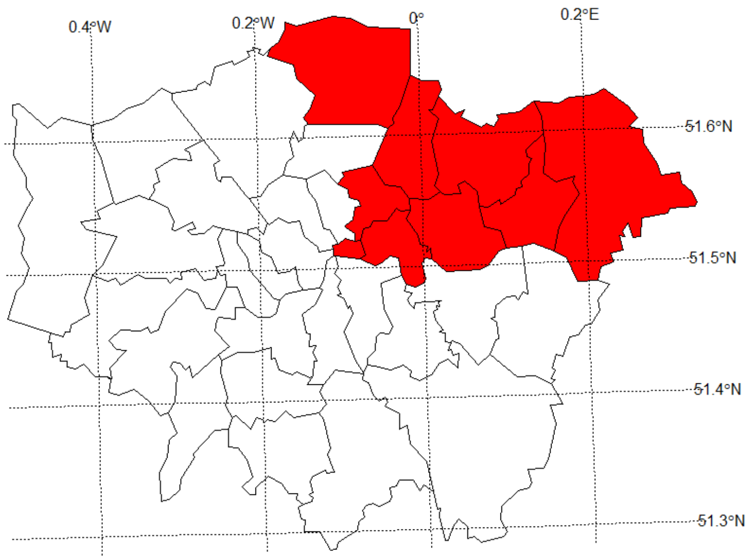
Plot Quadrants with Colors

```
plot(lnd)
plot(lnd[east & north,], col = "red", add = TRUE)
plot(lnd[east & south,], col = "green", add = TRUE)
plot(lnd[west & north,], col = "blue", add = TRUE)
plot(lnd[west & south,], col = "yellow", add = TRUE)
```



Plot with SE Quadrant Colored & add Gridlines

```
lnd$quadrant[!east & north] <- "northwest"
lnd$quadrant[east & !north] <- "southeast"
lnd$quadrant[!east & !north] <- "southwest"
plot(lnd)
plot(lnd[east & north,], add = TRUE, col = "red" )
llgridlines(lnd, lty= 3, side ="EN", offset = -0.5)
```



Create and Modify Spatial Data

R's spatial packages provide a very wide and powerful suite of functionality for processing and crea. Alongside visualisation and interrogation, a GIS must also be able to create and modify spatial data. R's spatial packages provide a very wide and powerful suite of functionality for processing and creating spatial data.

Creating New Data

R objects can be created by entering the name of the class we want to make. vector and data.frame objects for example, can be created as follows:

```
vec <- vector(mode = "numeric", length = 3)
df <- data.frame(x = 1:3, y = c(1/2, 2/3, 3/4))
```

We can check the class of these new objects using class():

```
class(vec)

## [1] "numeric"

class(df)

## [1] "data.frame"
```

Creating New Spatial Data

The same logic applies to spatial data. The input must be a numeric matrix or data.frame:

```
sp1 <- SpatialPoints(coords = df)
```

We have just created a spatial points object, one of the fundamental data types for spatial data. (The others arelines,polygonsandpixels,whichcanbecreatedbySpatialLines,SpatialPolygonsandSpatialPixels, respectively.) Each type of spatial data has a corollary that can accepts non-spatial data, created by adding DataFrame. SpatialPointsDataFrame(), for example, creates points with an associated data.frame.

Spatial Data Constraints

The number of rows in this dataset must equal the number of features in the spatial object, which in the case of sp1 is 3.

```
class(sp1)

## [1] "SpatialPoints"
## attr(,"package")
## [1] "sp"

spdf <- SpatialPointsDataFrame(sp1, data = df)
class(spdf)

## [1] "SpatialPointsDataFrame"
## attr(,"package")
## [1] "sp"
```

Comments on Spatial Objects

The above code extends the pre-existing spatial point object (sp1) by adding data from the data frame (df). To see how strict spatial classes are, try replacing df with mat in the above code: it causes an error. All spatial data classes can be created in a similar way, although SpatialLines and SpatialPolygons are much more complicated (Bivand et al. 2013). More frequently your spatial data will be read-in from an externally-created file, e.g. using readOGR(). Unlike the spatial objects we created above, most spatial data comes with an associate 'CRS'.

Projections: setting and transforming CRS in R

The Coordinate Reference System (CRS) of spatial objects defines where they are placed on the surface of the Earth. You may have noticed 'proj4string 'in the summary of lnd above: the information that follows represents its CRS. Spatial data should always have a CRS. If no CRS information is provided, and the correct CRS is known, it can be set as follow:

```
proj4string(lnd) <- NA_character_ # remove CRS information from lnd
proj4string(lnd) <- CRS("+init=epsg:27700") # assign a new CRS
```

CSRs and EPSG codes

Under this system 27700 represents the British National Grid. 'WGS84' (epsg:4326) is a very commonly used CRS worldwide. The following code shows how to search the list of available EPSG codes and create a new version of lnd in WGS84:3

```
EPSG <- make_EPSG() # create data frame of available EPSG codes
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

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

Using spTransform

Above, spTransform converts the coordinates of lnd into the widely used WGS84CRS. Now we've transformed lnd into a more widely used CRS, it is worth saving it. R stores data eficiently in .RData or .Rds formats. The former is more restrictive and maintains the object's name, so we use the latter. Save lnd84 object (we will use it in Part IV)

Removing Objects

Now we can remove the lnd84 object with the rm command. It will be useful later. (In RStudio, notice it also disappears from the Environment in the top right panel.)

```
rm(lnd84)
```

We will load it back in later with readRDS(file = "data/lnd84.Rds")

Attribute joins

Attribute joins are used to link additional pieces of information to our polygons. In the lnd object, for example, we have 4 attribute variables - that can be found by typing names(lnd). But what happens when we want to add more variables from an external source? We will use the example of recorded crimes by London boroughs to demonstrate this. To reafirm our starting point, we re-load the "london_sport" shapefile as a new object and plot it

```
library(rgdal) # ensure rgdal is loaded
lnd <- readOGR(dsn = mydir, "london_sport")

## OGR data source with driver: ESRI Shapefile
## Source: "C:\Users\jeff\Documents\Crime Analysis\Creating-maps-in-R-master\data", layer: "london_sport"
## with 33 features
## It has 4 fields
## Integer64 fields read as strings:  Pop_2001
```

```r
plot(lnd)
```

```r
nrow(lnd)

## [1] 33
```

Joining Non-Spatial Data

The non-spatial data we are going to join to the lnd object contains records of crimes in London. This is stored in a comma separated values (.csv) file called “mps-recordedcrime-borough”. If you open the file in a separate spreadsheet application first, we can see each row represents a single reported crime. We are going to use a function called aggregate to aggregate the crimes at the borough level, ready to join to our spatial lnd dataset. A new object called crime_data is created to store this data.

Create and Look at New crime_data Object

```r
crime_data <- read.csv("C:/Users/jeff/Documents/Crime Analysis/Creating-maps-in-R-master/data/mps-recordedcrime-borough.csv", stringsAsFactors = FALSE)
head(crime_data$CrimeType) # information about crime type

## [1] "Violence Against The Person" "Burglary"
## [3] "Other Notifiable Offences"   "Robbery"
## [5] "Theft & Handling"            "Theft & Handling"
```

We extract “Theft & Handling” Crimes:

```
crime_theft <- crime_data[crime_data$CrimeType == "Theft & Handling", ]
head(crime_theft, 2) # take a look at the result (replace 2 with 10 to see more rows)
```

```
##   X.1 X  Month        CrimeType                CrimeDetails CrimeCount
## 5   5 5 201104 Theft & Handling       Handling Stolen Goods          3
## 6   6 6 201104 Theft & Handling Theft/Taking Of Pedal Cycle         59
##                   Borough
## 5 Kensington and Chelsea
## 6 Kensington and Chelsea
```

We calculate the Sum of the Crime Count by District:

```
crime_ag <- aggregate(CrimeCount ~ Borough, FUN = sum, data = crime_theft)
```

Now, we show the First Two Rows of Aggregated Data:

```
head(crime_ag, 2)
```

```
##                 Borough CrimeCount
## 1 Barking and Dagenham      12222
## 2                Barnet      19821
```

Exploration Comments

You should not expect to understand all of this upon first try: simply typing the commands and thinking briefiy about the outputs is all that is needed at this stage. Here are a few things that you may not have seen before that will likely be useful in the future: In the first line of code when we read in the file we specify its location (check in your file browser to be sure). The == function is used to select only those observations that meet a specific condition i.e. where it is equal to, in this case all crimes involving "Theft and Handling". The ~ symbol means "by": we aggregated the CrimeCount variable by the district name.

London Boroughs

Now, that we have crime data at the borough level, the challenge is to join it to the lnd object. We will base our join on the Borough variable from the crime_ag object and the name variable from the lnd object. It is not always straight-forward to join objects based on names as the names do not always match. Now, we will see which names in the crime_ag object match the spatial data object, lnd:

```
lnd$name %in% crime_ag$Borough
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [23]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

```
lnd$name[!lnd$name %in% crime_ag$Borough]
```

```
## [1] City of London
## 33 Levels: Barking and Dagenham Barnet Bexley Brent Bromley ...
Westminster
```

Comments on the Code

The first line of code above uses the %in% command to identify which values in lnd$name are also contained in the Borough names of the aggregated crime data. The results indicate that all but one of the borough names matches. The second line of code tells us that it is 'City of London'. This does not exist in the crime data. This may be because the City of London has its own Police Force.

The borough name in the crime data does not match lnd$name is 'NULL'. Check this by typing

```
crime_ag$Borough[!crime_ag$Borough %in% lnd$name]

## [1] "NULL"
```

Joining Spatial and Non-Spatial Tables

Having checked the data found that one borough does not match, 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:

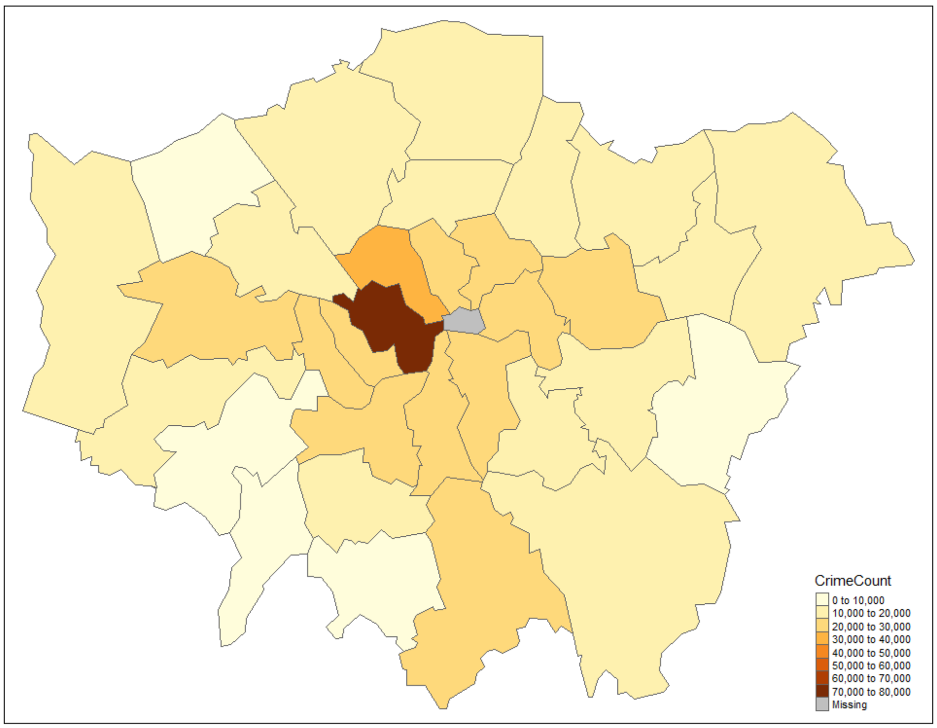
```
library(dplyr) # load dplyr
head(lnd$name) # dataset to add to (results not shown)

## [1] Bromley              Richmond upon Thames Hillingdon
## [4] Havering             Kingston upon Thames Sutton
## 33 Levels: Barking and Dagenham Barnet Bexley Brent Bromley ...
Westminster

head(crime_ag$Borough) # the variables to join

## [1] "Barking and Dagenham" "Barnet"               "Bexley"
## [4] "Brent"                "Bromley"              "Camden"

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

## Warning: Column `name`/`Borough` joining factor and character vector,
## coercing into character vector
```

Explore the New Dataset

Take a look at the new lnd@data object. You should see new variables added, meaning the attribute join was successful. You can now plot the rate of theft crimes in London by borough

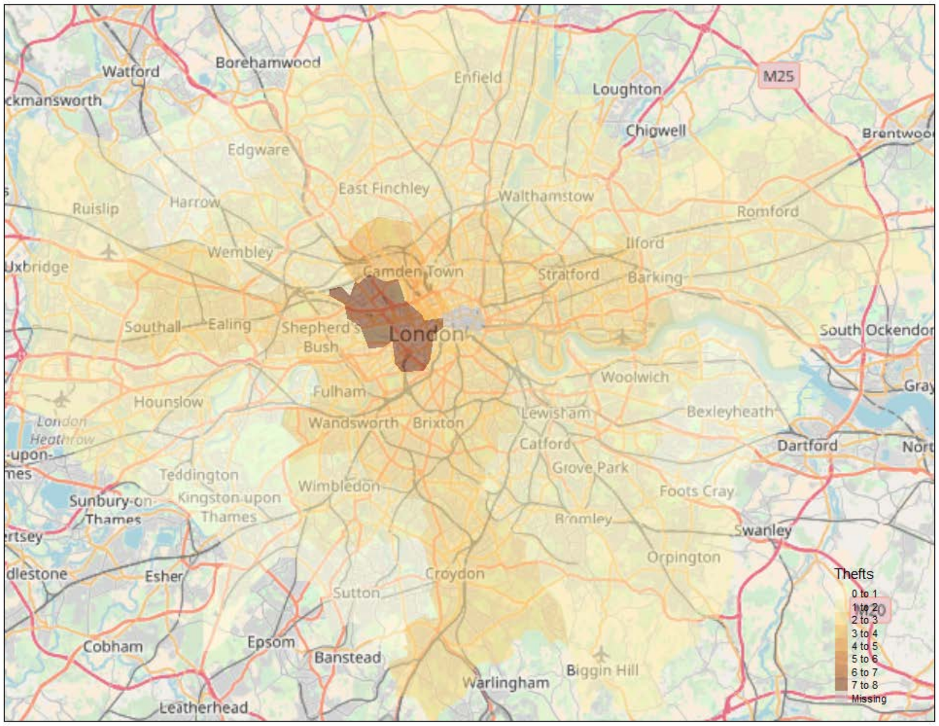
Plot a Basic Map

```
library(tmap) # load tmap package (see Section IV)
qtm(lnd, "CrimeCount") # plot the basic map
```



Plot and Enhanced Map

```
library(tmap)
library(tmaptools)
lnd_wgs = spTransform(lnd, CRS("+init=epsg:4326"))
osm_tiles = read_osm(bbox(lnd_wgs))

## Warning: Current projection unknown. Long lat coordinates (wgs84) assumed.

lnd_wgs$Thefts <- lnd$CrimeCount / 10000
tm_shape(osm_tiles) +
  tm_raster() +
  tm_shape(lnd_wgs) +
  tm_fill("Thefts", fill.title = "Thefts\n(10000)", scale = 0.8, alpha = 0.5)
+
  tm_layout(legend.position = c(0.89,0.02))
```



Clipping and Spatial Joins

In addition to joining by attribute (e.g. Borough name), it is also possible to do spatial joins in R. We use transport infrastructure points as the spatial data to join, with the aim of finding out about how many are found in each London borough.

Create New Stations Object Using a Shapefile

```
library(rgdal)
library(sf)

## Linking to GEOS 3.6.1, GDAL 2.2.3, proj.4 4.9.3

stations <- readOGR(dsn = mydir, layer = "lnd-stns")

## OGR data source with driver: ESRI Shapefile
## Source: "C:\Users\jeff\Documents\Crime Analysis\Creating-maps-in-R-
master\data", layer: "lnd-stns"
## with 2532 features
## It has 27 fields
## Integer64 fields read as strings:  CODE IMPERIAL METRIC

proj4string(stations) # this is the full geographical detail.

## [1] "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"

proj4string(lnd) # what's the coordinate reference system (CRS)
```

```
## [1] "+proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000 +y_0=-
100000 +ellps=airy +units=m +no_defs"
```

```
bbox(stations) # the extent, 'bounding box' of stations
```

```
##                    min        max
## coords.x1 -1.199066  0.9358515
## coords.x2 50.984598 51.9398978
```

```
bbox(lnd) # return the bounding box of the lnd object
```

```
##         min      max
## x 503571.2 561941.1
## y 155850.8 200932.5
```

The proj4string() Function

This function shows that the Coordinate Reference System (CRS) of stations difiers from that of our lnd object. OSGB 1936 (or EPSG 27700) is the oficial CRS for the UK, so we will convert the 'stations' object to this:

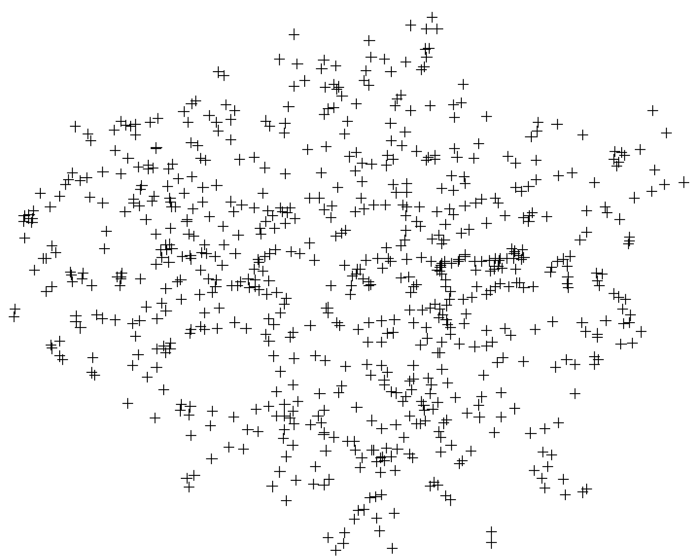
Create Reprojected Stations Object

```
stations <- spTransform(stations, CRSobj = CRS(proj4string(lnd)))
plot(lnd) # plot London
points(stations) # overlay the station points
```

Commenst on Stations Objects

Note the stations points now overlay the boroughs but that the spatial extent of stations is greater than that of lnd. To clip the stations so that only those falling within London boroughs are retained we can use sp::over, or simply the square bracket notation for subsetting tabular data. Enter ?gIntersects to find out another way to do this

```
stations <- stations[lnd, ]
plot(stations) # test the clip succeeded
```



Comments on the Code

The above line of code says: "output all `stations` within the `lnd` object bounds", a concise way of clipping that is consistent with Rs syntax for non-spatial clipping. To prove it worked, only stations within the London boroughs appear in the plot. gIntersects can achieve the same result, but with more lines of code (see www.rpubs.com/RobinLovelace for more on this). It may seem confusing that two different functions can be used to generate the same result. However, this is a common issue in R; the question is finding the most appropriate solution.

Viewing the Summary

Typing `summary(sel)` should provide insight into how this worked: it is a data frame with 1801 NA values, representing zones outside of the London polygon. Note that the preceding two lines of code is equivalent to the single line of code, `stations <- stations[lnd, ]`.

The next section demonstrates spatial aggregation, a more advanced version of spatial subsetting.

```
summary(sel)
```

```
## Object of class SpatialPoints
## Coordinates:
##                  min      max
## coords.x1 523592.6 541466.2
## coords.x2 174026.6 189667.2
## Is projected: TRUE
## proj4string :
## [+proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000
## +y_0=-100000 +ellps=airy +units=m +no_defs]
## Number of points: 14
```

Spatial aggregation

As with Rs very terse code for spatial subsetting, the base function `aggregate` (which provides summaries of variables based on some grouping variable) also behaves differently when the inputs are spatial objects.

```
stations_agg <- aggregate(x = stations["CODE"], by = lnd, FUN = length)
head(stations_agg@data)
```

```
##   CODE
## 0   48
## 1   22
## 2   43
## 3   18
## 4   12
## 5   13
```

Comments about the Code

The above code performs a number of steps in just one line: `aggregate` identifies which `lnd` polygon (borough) each `station` is located in and groups them accordingly. The use of the syntax `stations["CODE"]` tells R that we are interested in the spatial data from `stations` and its `CODE` variable (any variable could have been used here as we are merely counting how many points exist). It counts the number of `stations` points in each borough, using the function `length`. A new spatial object is created, with the same geometry as `lnd`, and assigned the name `stations_agg`, the count of stations.

Extract the Raw Count Data

To extract the raw count data, one could enter `stations_agg$CODE`. This variable could be added to the original `lnd` object as a new field, as follows:

```
lnd$n_points <- stations_agg$CODE
```

Spatial Implementation of `aggregate`

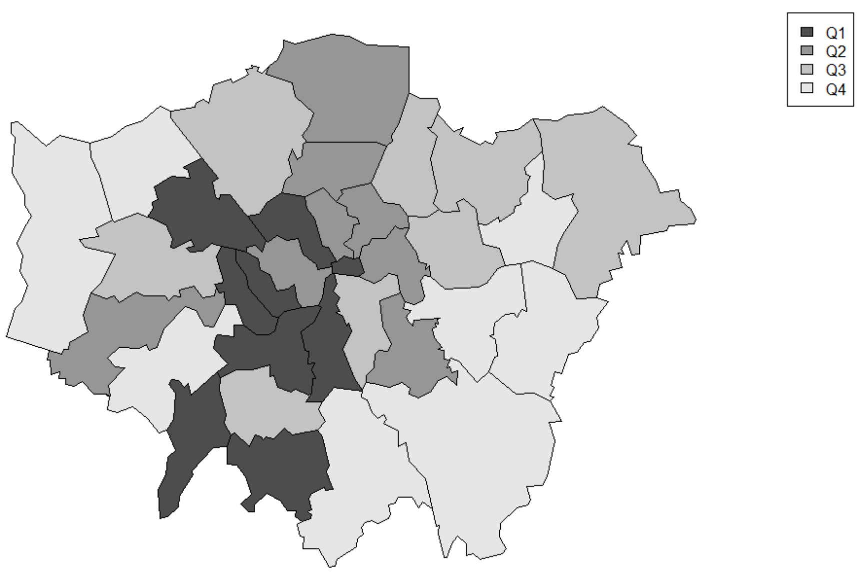
As shown below, the spatial implementation of `aggregate` can provide summary statistics of variables, as well as simple counts. In this case we take the variable `NUMBER` and find its mean value for the stations in each ward. [See the miniature Vignette 'Clipping and aggregating spatial data with gIntersects' for more information on this: http://rpubs.com/RobinLovelace/83834 .]

```
lnd_n <- aggregate(stations["NUMBER"], by = lnd, FUN = mean)
```

Analyze and Plot the Result

For an optional advanced task, let us analyse and plot the result.

```
library(ggplot2)
q <- cut_number(lnd_n$NUMBER,4) # a nice little function from ggplot2
q <- factor(q, labels = grey.colors(n = 4))
summary(q)

## #4D4D4D #969696 #C3C3C3 #E6E6E6
##       9       8       8       8

qc <- as.character(q) # convert to character class to plot
plot(lnd_n, col = qc) # plot (not shown in printed tutorial)
legend(legend = paste0("Q", 1:4), fill = levels(q), "topright")
```

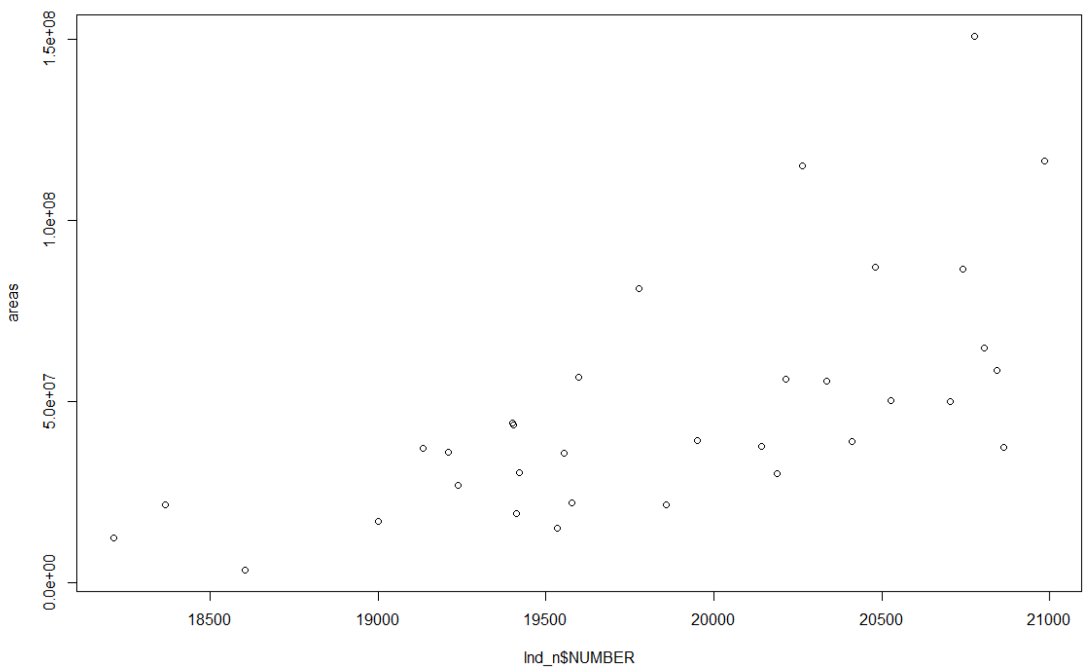



```
areas <- sapply(lnd_n@polygons, function(x) x@area)
```

Comments on the Plot

This results in a simple choropleth map and a new vector containing the area of each borough. As an additional step, try comparing the mean area of each borough with the mean value of `stations` points within it:

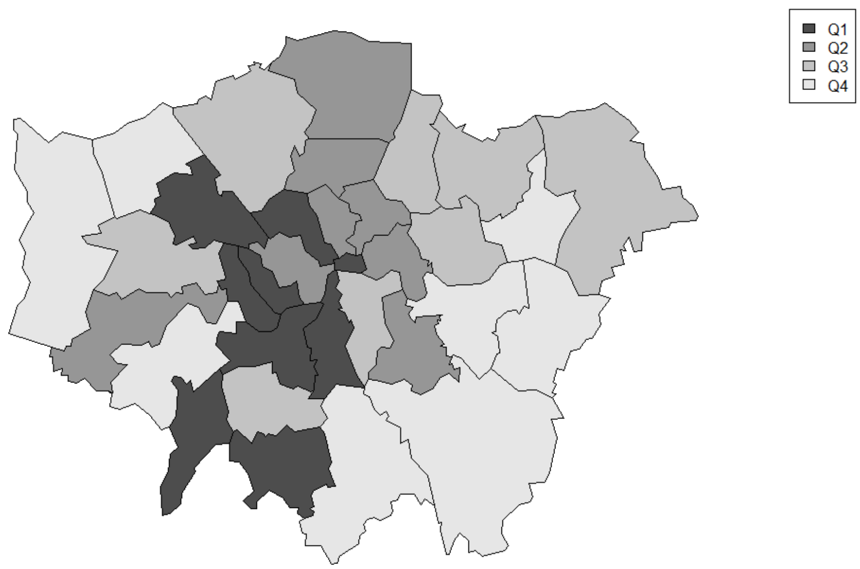
```
plot(lnd_n$NUMBER, areas)
```

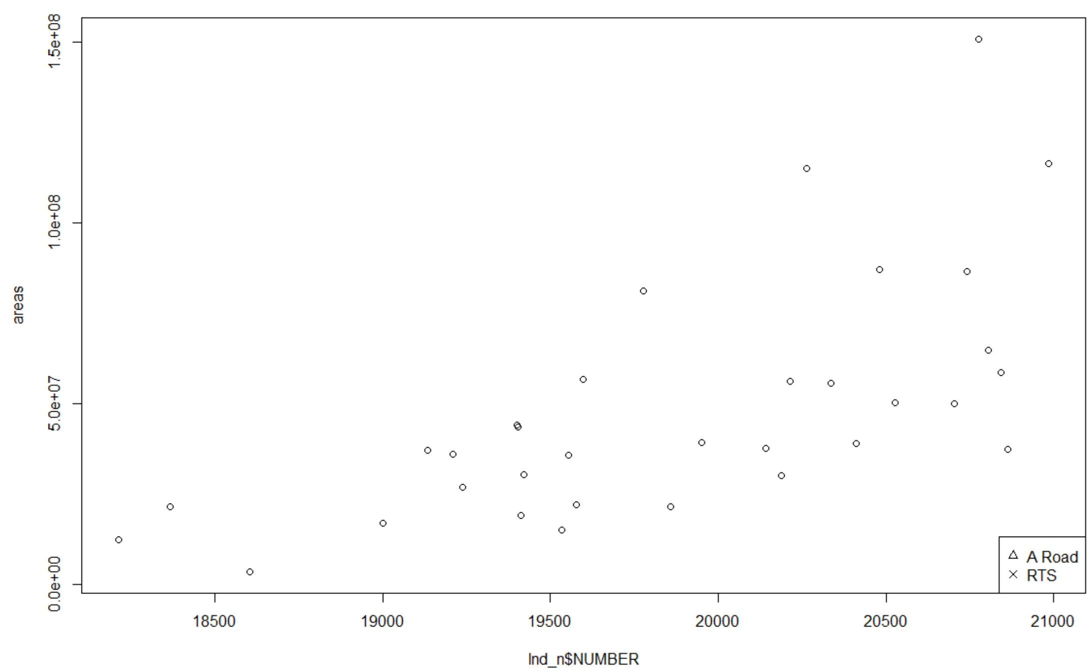


Adding symbols for tube and train stations

Imagine now that we want to display all tube and train stations on top of the previously created choropleth map. How would we do this? The shape of points in R is determined by the `pch` argument, as demonstrated by the result of entering the following code: `plot(1:10, pch=1:10)`. To apply this knowledge to our map, try adding the following code to the chunk above (output not shown):

```
levels(stations$LEGEND) # see A roads and rapid transit stations (RTS) (not
shown)
sel <- grepl("A Road Sing|Rapid", stations$LEGEND) # selection for plotting
sym <- as.integer(stations$LEGEND[sel]) # symbols
points(stations[sel,], pch = sym)
legend(legend = c("A Road", "RTS"), "bottomright", pch = unique(sym))
```






```
## #4D4D4D #969696 #C3C3C3 #E6E6E6
##       9       8       8       8
## [1] "Railway Station"
## [2] "Rapid Transit Station"
## [3] "Roundabout, A Road Dual Carriageway"
## [4] "Roundabout, A Road Single Carriageway"
```

```
## [5] "Roundabout, B Road Dual Carriageway"
## [6] "Roundabout, B Road Single Carriageway"
## [7] "Roundabout, Minor Road over 4 metres wide"
## [8] "Roundabout, Primary Route Dual Carriageway"
## [9] "Roundabout, Primary Route Single C'way"
```

Plot Code

```
library(ggplot2)
q <- cut_number(lnd_n$NUMBER,4) # a nice little function from ggplot2
q <- factor(q, labels = grey.colors(n = 4))
summary(q)
qc <- as.character(q) # convert to character class to plot
plot(lnd_n, col = qc) # plot (not shown in printed tutorial)
legend(legend = paste0("Q", 1:4), fill = levels(q), "topright")
areas <- sapply(lnd_n@polygons, function(x) x@area)
plot(lnd_n$NUMBER, areas)
levels(stations$LEGEND)
sel <- grepl("A Road Sing|Rapid", stations$LEGEND) # selection for plotting
sym <- as.integer(stations$LEGEND[sel]) # symbols
points(stations[sel,], pch = sym)
legend(legend = c("A Road", "RTS"), "bottomright", pch = unique(sym))
```

Comments on the Code

The above block of code first identifies which types of transport points are present in the map with `levels` (this command only works on factor data). Next we select a subset of `stations` using a new command, `grepl`, to determine which points we want to plot. Note that `grepl`s first argument is a text string (hence the quote marks) and the second is a factor (try typing `class(stations$LEGEND)` to test this). grepl`uses *regular expressions* to match whether each element in a vector of text or factor names match the text pattern we want. In this case, because we are only interested in roundabouts that are A roads and Rapid Transit systems (RTS). Note the use of the vertical separator`|`to indicate that we want to match`LEGEND`names that contain either "A Road" *or* "Rapid". Based on the positive matches (saved as`sel`, a vector of`TRUE`and`FALSE` values), we subset the stations. Finally we plot these as points, using the integer of their name to decide the symbol and add a legend.

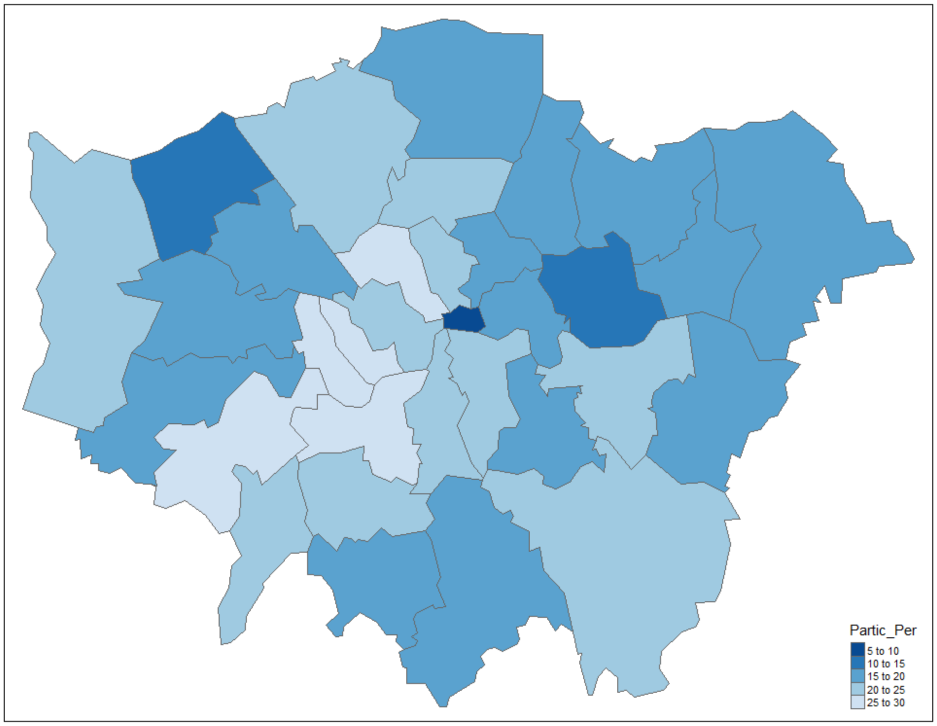
Making maps with tmap

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:

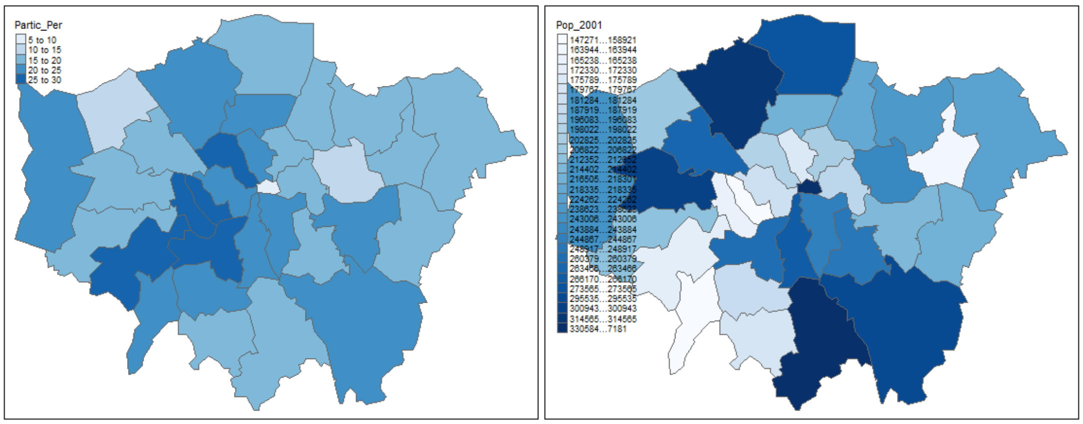
```
library(tmap)
vignette("tmap-getstarted")

## starting httpd help server ... done
```

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

```
qtm(shp = lnd, fill = "Partic_Per", fill.palette = "-Blues") # not shown
```




```
qtm(shp = lnd, fill = c("Partic_Per", "Pop_2001"), fill.palette = "Blues",
nrow = 1)

## Warning: Number of levels of the variable "Pop_2001" is 33, which is
## larger than max.categories (which is 30), so levels are combined. Set
## tmap_options(max.categories = 33) in the layer function to show all
levels.
```

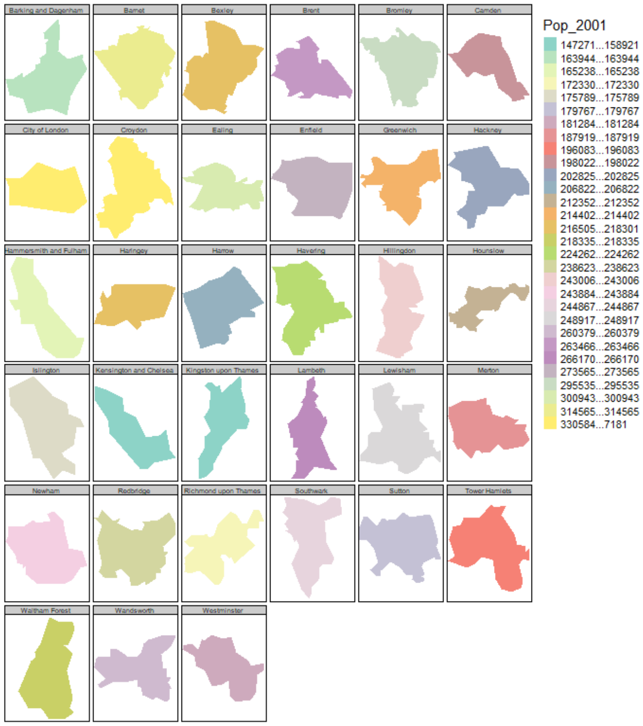


Comments on the Plot

The plot above shows the ease with which tmap can create maps next to each other for difierent variables. The plot produced by the following code chunk (not shown) demonstrates the power of the tm_facets command. Note that all the maps created with the qtm function can also be created with tm_shape, followed by tm_fill (or another tm_ function).

Enhance the Plot

```
tm_shape(lnd) +
  tm_fill("Pop_2001", thres.poly = 0) +
  tm_facets("name", free.coords = TRUE, drop.units = TRUE)

## Warning: Number of levels of the variable "Pop_2001" is 33, which is
## larger than max.categories (which is 30), so levels are combined. Set
## tmap_options(max.categories = 33) in the layer function to show all
levels.
```



Create a Basemap.

To create a basemap with tmap, you can use the read_osm function, from the tmaptools package as follows. Note that you must first transform the data into a geographical CRS:

```
library(OpenStreetMap)
lnd_wgs = spTransform(lnd, CRS("+init=epsg:4326"))
osm_tiles = tmaptools::read_osm(bbox(lnd_wgs)) # download images from OSM

## Warning: Current projection unknown. Long lat coordinates (wgs84) assumed.

tm_shape(osm_tiles) + tm_raster() + tm_shape(lnd_wgs) +
  tm_fill("Pop_2001", fill.title = "Population, 2001", scale = 0.8, alpha =
0.5) +
  tm_layout(legend.position = c(0.89,0.02))

## Warning: Number of levels of the variable "Pop_2001" is 33, which is
## larger than max.categories (which is 30), so levels are combined. Set
## tmap_options(max.categories = 33) in the layer function to show all
levels.
```

Another way to make tmap maps have a basemap is by entering tmap_mode("view"). This will make the maps appear on a zoomable webmap powered by leafiet. There are many other intuitive and powerful functions in tmap. Check the documentation to find out more:

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

Making Maps with ggmap.

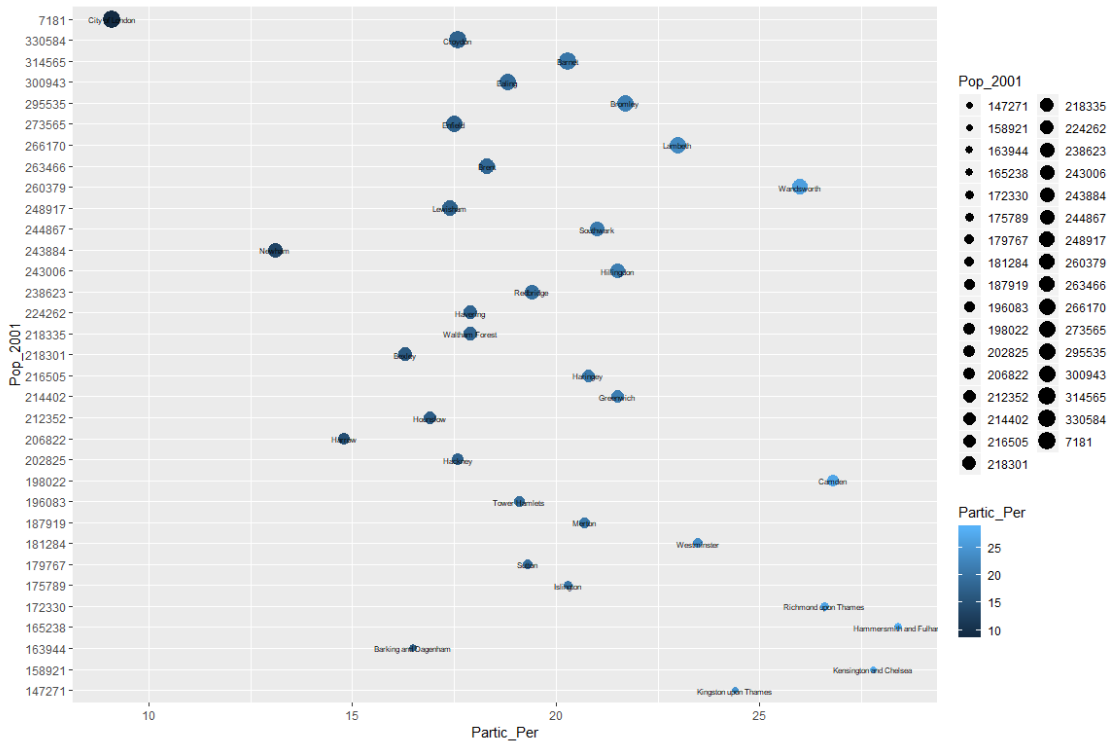
ggmap is based on the ggplot2 package, an implementation of the Grammar of Graphics (Wilkinson 2005). ggplot2 can replace the base graphics in R (the functions you have been plotting with so far). It contains default options that match good visualisation practice and is well-documented: http://docs.ggplot2.org/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

```
library(ggplot2)
p <- ggplot(lnd@data, aes(Partic_Per, Pop_2001))
```

Add Layers to a Plot.

The real power of ggplot2 lies in its ability to add layers to a plot. In this case we can add text to the plot.

```
p + geom_point(aes(colour = Partic_Per, size = Pop_2001)) +
  geom_text(size = 2, aes(label = name))

## Warning: Using size for a discrete variable is not advised.
```



Comments about Layers.

This idea of layers (or geoms) is quite difierent from the standard plot functions in R. You will find that each of the functions does a lot of clever stufi to make plotting much easier (see the documentation for a full list).

Creating Dataframes.

In the following steps we will create a map to show the percentage of the population in each London Borough who regularly participate in sports activities. ggmap requires spatial data to be supplied as data.frame, using fortify(). The generic plot() function can use Spatial* objects directly; ggplot2 cannot. Therefore we need to extract them as a data frame. The fortify function was written specifically for this purpose. For this to work, either the maptools or rgeos packages must be installed.

```
library(rgeos)
lnd_f <- fortify(lnd) ## Regions defined for each Polygons

## Regions defined for each Polygons
```

This step has lost the attribute information associated with the lnd object. We can add it back using the left_join function from the dplyr package (see ?left_join).

```
head(lnd_f, n = 2) # peak at the fortified data
```

```
##       long      lat order  hole piece id group
## 1 541177.7 173555.7     1 FALSE     1  0   0.1
## 2 541872.2 173305.8     2 FALSE     1  0   0.1
```

```
lnd$id <- row.names(lnd) # allocate an id variable to the sp data
head(lnd@data, n = 2) # final check before join (requires shared variable
name)
```

```
##   ons_label                 name Partic_Per Pop_2001 CrimeCount n_points
## 1      00AF              Bromley       21.7   295535      15172       48
## 2      00BD Richmond upon Thames       26.6   172330       9715       22
##   id
## 1  0
## 2  1
```

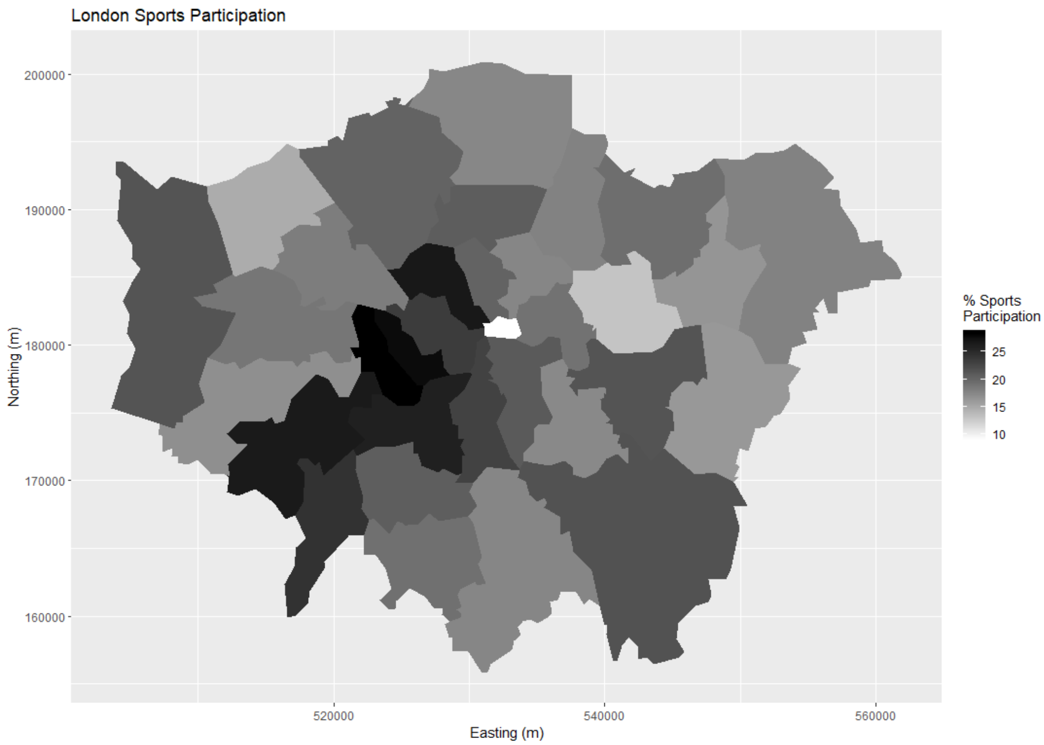
```
lnd_f <- left_join(lnd_f, lnd@data) # join the data
```

```
## Joining, by = "id"
```

Plotting the New Object

The new lnd_f object contains coordinates alongside the attribute information associated with each London Borough. It is now straightforward to produce a map with ggplot2. coord_equal() is the equivalent of asp = T in regular plots with R:

```
map <- 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")
map + scale_fill_gradient(low = "white", high = "black")
```



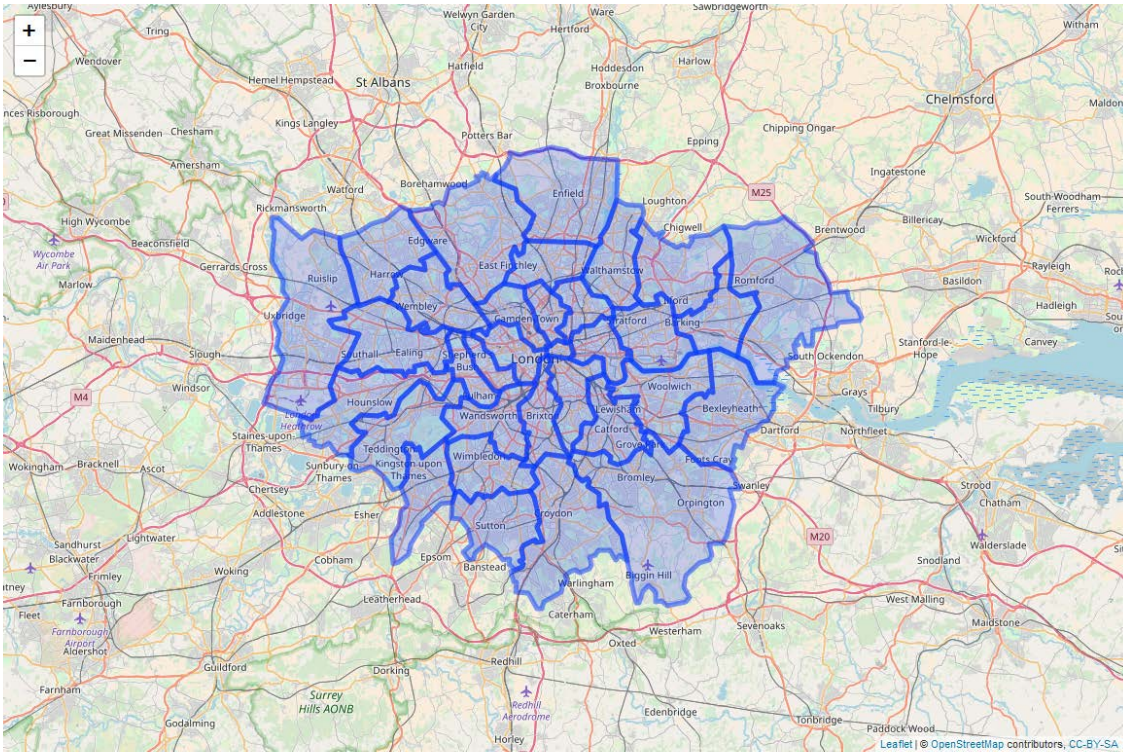

```
## Creating interactive maps with leafiet
```

Using Leaflet

Leafiet is the worlds premier web mapping system, serving hundreds of thousands of maps worldwide each day. The JavaScript library actively developed at github.com/Leafiet/Leafiet, has a strong user community. It is fast, powerful and easy to learn. The leafiet 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/leafiet, see rstudio.github.io/leafiet/ and the following on-line tutorial: robinlovelace.net/r/2015/02/01/leafiet-r-package.html.

We now generate a map from lnd84.rds file we created earlier.

```
library(leaflet)
lnd84 <- readRDS('C:/Users/jeff/Documents/Crime Analysis/Creating-maps-in-R-master/data/lnd84.Rds')
leaflet() %>%
  addTiles() %>%
  addPolygons(data = lnd84)
```



Advanced Task: Faceting for Maps

The below code demonstrates how to read in the necessary data for this task and 'tidy' it up. The data file contains historic population values between 1801 and 2001 for London, again from the London data store. We tidy the data so that the columns become rows. In other words, we convert the data from 'fiat' to 'long' format. This is the form required by ggplot2 for faceting graphics: the date of the population survey becomes a variable in its own right, rather than being strung-out over many columns.

Tidy up the data

```
london_data <- read.csv("C:/Users/jeff/Documents/Crime Analysis/Creating-
maps-in-R-master/data/census-historic-population-borough.csv")
library(tidyr) # if not install it, or skip the next two steps
library(dplyr)
ltidy <- gather(london_data, date, pop, -Area.Code, -Area.Name)
head(ltidy, 2) # check the output

##   Area.Code            Area.Name     date    pop
## 1      00AA       City of London Pop_1801 129000
## 2      00AB Barking and Dagenham Pop_1801   3000
```

Comments on Tidy

In the above code we take the london_data object and create the column names 'date' (the date of the record, previously spread over many columns) and 'pop' (the population which varies). The minus (-) symbol in this context tells gather not to include the Area.Name and

Area.Code as columns to be removed. In other words, "leave these columns be". Data tidying is an important subject: more can be read on the subject in Wickham (2014) or in a vignette about the package, accessed from within R by entering vignette("tidy-data").

Merge the Population Data

Now we merge with the London borough geometry contained within our lnd_f object, using the left_join function from the dplyr package:

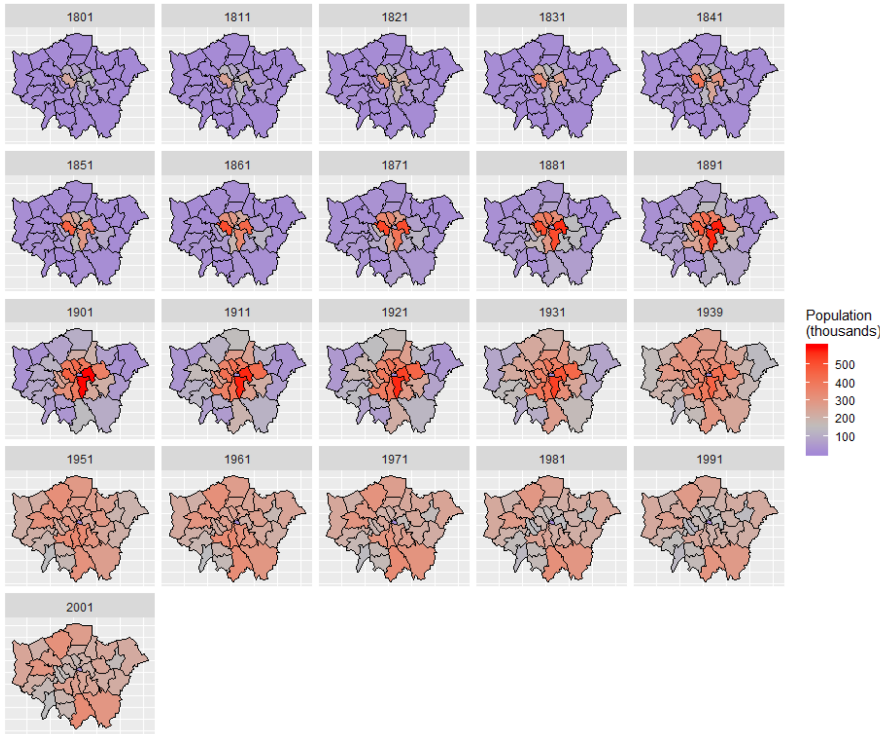
```
head(lnd_f, 2) # identify shared variables with ltidy

##       long      lat order  hole piece id group ons_label    name
## 1 541177.7 173555.7     1 FALSE     1  0   0.1      00AF Bromley
## 2 541872.2 173305.8     2 FALSE     1  0   0.1      00AF Bromley
##   Partic_Per Pop_2001 CrimeCount n_points
## 1       21.7   295535      15172       48
## 2       21.7   295535      15172       48

ltidy <- rename(ltidy, ons_label = Area.Code) # rename Area.code variable
lnd_f <- left_join(lnd_f, ltidy)

## Joining, by = "ons_label"

## Warning: Column `ons_label` joining factors with different levels,
coercing
## to character vector
```

We use ?gsub and Google 'regex' to find out more.

```
lnd_f$date <- gsub(pattern = "Pop_", replacement = "", lnd_f$date)
```

Finally, we use faceting to Produce One Map per year.

```
library(ggplot2)
ggplot(data = lnd_f, # the input data
       aes(x = long, y = lat, fill = pop/1000, group = group)) + # define
variables
  geom_polygon() + # plot the boroughs
  geom_path(colour="black", lwd=0.05) + # borough borders
  coord_equal() + # fixed x and y scales
  facet_wrap(~ date) + # one plot per time slice
  scale_fill_gradient2(low = "blue", mid = "grey", high = "red", # colors
                       midpoint = 150, name = "Population\n(thousands)") + #
legend options
  theme(axis.text = element_blank(), # change the theme options
        axis.title = element_blank(), # remove axis titles
        axis.ticks = element_blank()) # remove axis ticks
```


1801
1811
1821
1831
1841
1851
1861
1871
1881
1891
1901
1911
1921
1931
1939
1951
1961
1971
1981
1991
2001
Population (thousands)
500
400
300
200
100