An Introduction to Spatial Data Analysis and Visualisation in R

Guy Lansley and James Cheshire

2016

Contents

Introduction . . . 3

Practical 1: An Introduction to R **4**
Downloading data from the CDRC data website . . . 4
Loading data and data formatting in R . . . 6
Loading data into R . . . 8
Viewing data . . . 8
Observing column names . . . 8
Selecting columns . . . 9
Renaming column headers . . . 9
Joining data in R . . . 9
Exporting Data . . . 10

Practical 2: Data exploration in R **11**
Exploring the data . . . 11
Descriptive statistics . . . 13
Univariate plots . . . 13
Histograms . . . 13
Boxplots . . . 15
Installing the vioplot package and creating a violin plot . . . 16
Exporting images . . . 19

Practical 3: Bivariate Plots in R **21**
Simple scatter plots . . . 21
Symbols plot . . . 24
Adding a regression line . . . 25
Using the ggplot2 package . . . 27

Practical 4: Finding Relationships in R **30**
Bivariate correlations . . . 30
Regression analysis . . . 32
Mutliple regression . . . 37

Practical 5: Making maps in R **39**
Loading shapefiles into R . . . 39
Joining data . . . 40
Setting a coordinate system . . . 41
Mapping data in R . . . 41
Creating a quick map . . . 41
Creating more advanced maps with tmap . . . 42
Creating a simple map . . . 42
Adding borders . . . 48
Adding a north arrow . . . 49

Editing the layout of the map . . . 50
Saving the shapefile . . . 51

Practical 6: Mapping Point Data in R **52**
Loading point data into R . . . 52
Mapping point data . . . 53
Creating proportional symbol maps . . . 57
Saving the shapefile . . . 59

Practical 7: Using R as a GIS **60**
Buffers . . . 63
Union . . . 63
Adding Backing Maps . . . 64
Creating Interactive Maps . . . 66

Practical 8: Representing Densities in R **70**
Running a kernel density estimation . . . 70
Identifying homeranges . . . 74

Practical 9: Measuring Spatial Autocorrelation in R **76**
Running a spatial autocorrelation . . . 77
Finding neighbours . . . 77
Running a global spatial autocorrelation . . . 79
Running a local spatial autocorrelation . . . 80
Getis-Ord . . . 83

Practical 10: Geographically Weighted Regression in R **86**
Run a linear model . . . 86
Mapping the resididuals . . . 92
Run a GWR . . . 93

Practical 11: Interpolating Point Data in R **98**
Thiessen polygons . . . 99
Inverse Distance Weighting (IDW) . . . 102
3D surfaces . . . 104
Masking a raster . . . 108
Geostatistical interpolation . . . 109

Practical 12: Functions and Loops in R **111**
User-written functions . . . 111
Loops . . . 117
For loop . . . 117
If. else statements . . . 118

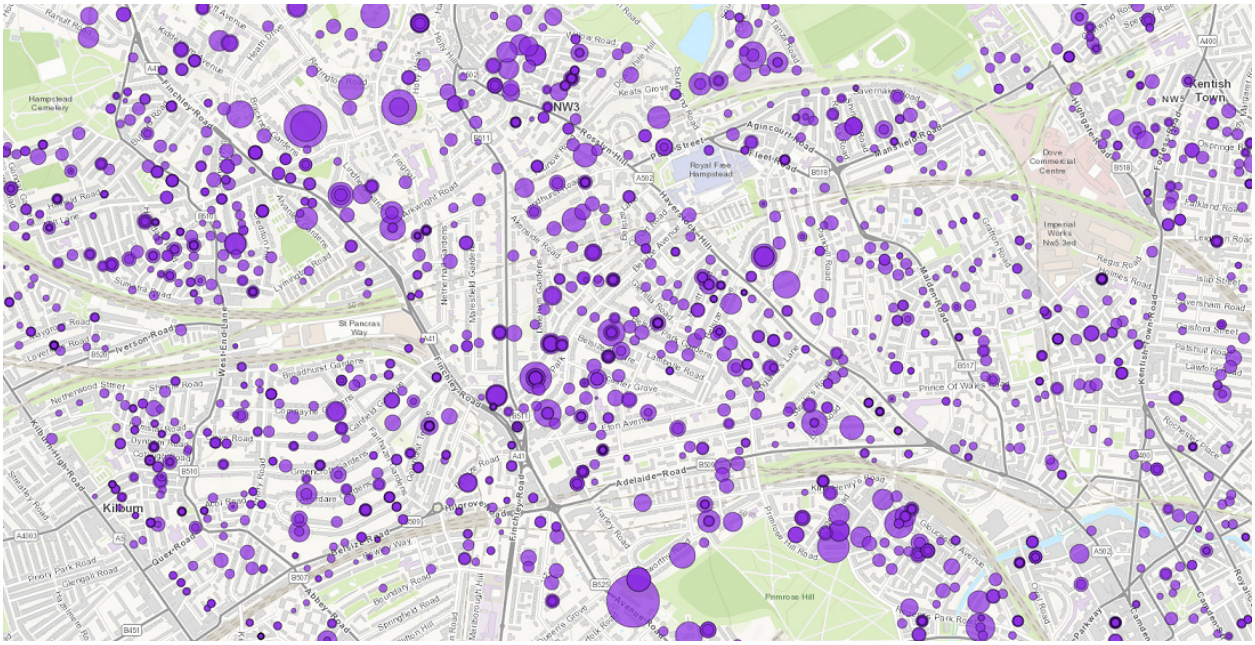
Acknowledgements **121**

References **121**

An Introduction to Spatial Data Analysis and Visualisation in R

Guy Lansley[1] and James Cheshire[2]

University College London



Introduction

This tutorial series is designed to provide an accessible introduction to techniques for handling, analysing and visualising spatial data in R. R is an open source software environment for statistical computing and graphics. It has a range of bespoke packages which provide additional functionality for handling spatial data and performing complex spatial analysis operations. The practical series uses open data which has been made readily available and demonstrates range of techniques useful in social sciences including multivariate analysis, mapping and spatial interpolation. While this series provides an introduction to implementing a number of useful statistical and geospatial techniques in R, we have also provided hyperlinks to external sources which provide comprehensive details about science behind such methods. Noteable useful resources are the Statistical Analysis Handbook (www.statsref.com) and Geospatial Analysis (www.spatialanalysisonline.com) online text books, both of which are free to access.

The data for the practicals have been made freely available on the Consumer Data Research Centre data service website at: https://data.cdrc.ac.uk/tutorial/an-introduction-to-spatial-data-analysis-and-visualisation-in-r

For any questions or concerns please email Guy at g.lansley@ucl.ac.uk.

---

[1] g.lansley@ucl.ac.uk
[2] james.cheshire@ucl.ac.uk

Practical 1: An Introduction to R

This practical is the first in a series of tutorials which will introduce you to a range of useful techniques for handling, analysing and visualising spatial data in R. All of the data and software used are freely available as open data and open software.

> "Open data is data that can be freely used, re-used and redistributed by anyone - subject only, at most, to the requirement to attribute and sharealike." *The Open Data Handbook*

Prior to working in R, we will first take you through obtaining the data from the Consumer Data Research Centre's (CDRC) data portal which stores consumer-related data from a large number of sources within the UK. Consumer data are data generated by retailers and other service organisations as part of their routine business processes. They are commonly used within the private sector to monitor the needs, preferences and behaviours of customers. However, the data is also of high value to public institutions. Census data is also useful to consumer insight, all leading retailers rely on accurate spatial data on population in order to guide the planning of their store locations and how their stock is distributed and marketed across the country.

The CDRC offers an open data service through which members of the public can register online and freely download data pertaining to consumers. The service can be accessed by visiting https://data.cdrc.ac.uk/

The following practicals will use the London Borough of Camden as their case study area.

In this tutorial we will:

- Download a Census datapack from the CDRC Data website
- Load the data into R using RStudio
- View the raw data in R
- Subset data in R
- Merge data in R

Downloading data from the CDRC data website

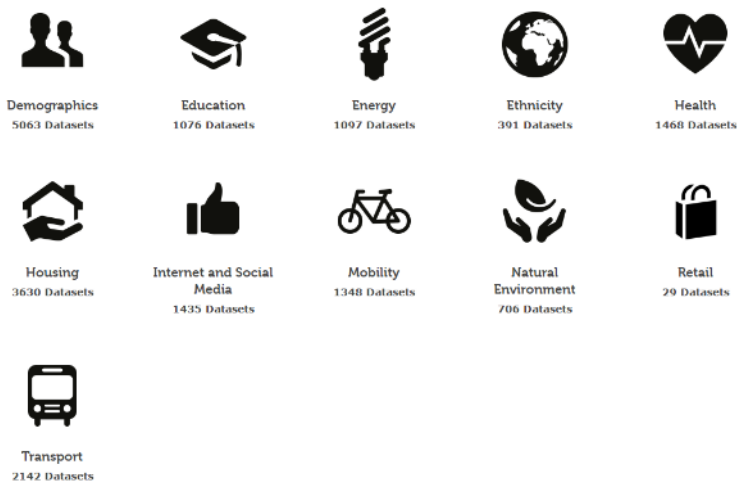
Before we introduce you to R and Rstudio, we will first download some data from the CDRC Data Service. On an internet browser go to https://data.cdrc.ac.uk/

In the top right of the screen you will see options to log in or register for an account. If you have not yet registered please sign up to an account now.

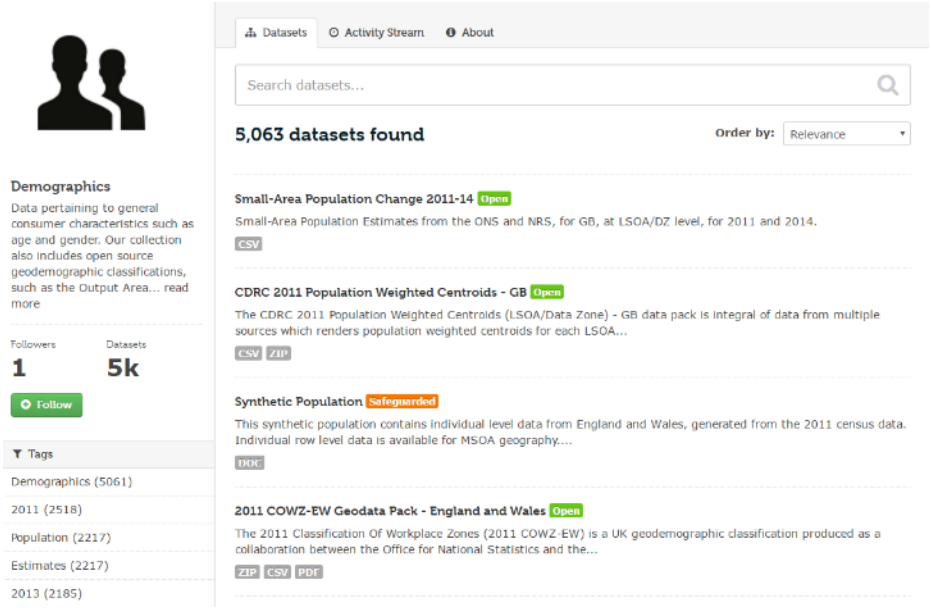
We are going to be interested in small area Census data for the Borough of Camden. There are several routes to this dataset which include a search bar at the top of each page.

On the tab panel at the top of the page, click on **Topics**.

The data available on the CDRC data website can be broadly grouped into 11 key themes. Most of these pertain to the population and human activities. All of these themes are important to a wide range of industries, notably including retail. Census data can be found by clicking on the **Demographics** topic.



However, within this option there is still a very large number of datasets as the CDRC stores individual data packs for every district within the UK. In the search bar at the top of the page, enter 'Camden'.



You now want to scroll down to **CDRC 2011 Census Data Packs for Local Authority District: Camden (E09000007)**. It can also be found by clicking on the Census tab on the side of the screen.

The following page describes the content of the data and disclosure controls. The data is provided as a zipped folder of several different tables of Census data which encompass a wide variety of variables on the population. In addition, data is also provided at three different geographic scales of data units - Output Area, Lower Super Output Area and Middle Super Output Area. The geographic units have been described on the ONS website.

To download the file you must click on **Camden.zip** under Data and Resources. If you have not already done so, here you will be asked to freely register as a new user.

It is recommended that you move the downloaded **Camden.zip** file to somewhere appropriate in your directory and then unzip the folder. To unzip on a windows computer simply right-click on the zipped file in windows explorer and click on **"Extract All."**.

The folder includes lots of useful data. The **tables** subfolder is where the Census data is stored in its various

forms. However, each table is given a code name which is not informative to us, so the **datasets_description** file is provided so we can lookup their names. In addition to this, a **variables_description** is provided so we can look up the variable name codes within each data table. GIS shapefiles are also available from the **shapefiles** subfolder, these will be useful for mapping the data.

The Census data pack includes a large number of different datasets, all stored as Comma Separated Values (.csv) files. CSVs are a simple means of storing data so that it can be easily read and written on a computer. They are simply plain text documents where commas are used to separate the fields of data (you can observe this if you open a CSV in notepad).

For the forthcoming practicals we will be considering three variables, each from a different census dataset. These will be:

ColumnVariableCode	ColumnVariableDescription	DatasetId	DatasetTitle
KS201EW0020	White: English/Welsh/Scottish/Northern Irish/British	KS201EW	Ethnic group
KS403EW0012	Occupancy rating (bedrooms) of -1 or less	KS403EW	Rooms, bedrooms and central heating
KS501EW0019	Highest level of qualification: Level 4 qualifications and above[1]	KS501EW	Qualifications and students
KS601EW0019	Economically active: Unemployed	KS601EW	Economic activity

[1]Level 4 qualifications refer to a Certificate of Higher Education Higher National Certificate (awarded by a degree-awarding body

Loading data and data formatting in R

R is a free software environment for statistical computing and graphics. It is extremely powerful and as such is now widely used for academic research as well as in the commercial sector. Unlike software such as Excel or SPSS, the user has to type commands to get it to execute tasks such as loading in a dataset or performing a calculation. The biggest advantage of this approach is that you can build up a document, or script, that provides a record of what you have done, which in turn enables the straightforward repetition of tasks. Graphics can be easily modified and tweaked by making slight changes to the script or by scrolling through past commands and making quick edits. Unfortunately command-line computing can also be off-putting at first. It is easy to make mistakes that aren't always obvious to detect. Nevertheless, there are good reasons to stick with R. These include:

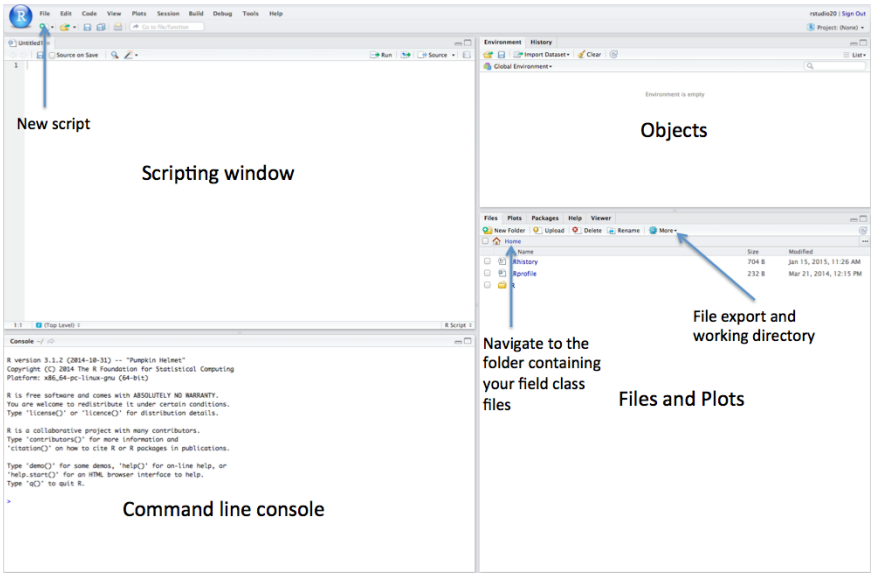
- It's broadly intuitive with a strong focus on publishable-quality graphics. It's 'intelligent' and offers in-built good practice - it tends to stick to statistical conventions and present data in sensible ways.
- It's free, cross-platform, customisable and extendable with a whole swathe of libraries ('add ons') including those for discrete choice, multilevel and longitudinal regression, and mapping, spatial statistics, spatial regression and geostatistics.
- It is well respected and used at the world's largest technology companies (including Google, Microsoft and Facebook), the largest pharmaceutical companies (including Johnson & Johnson, Merck, and Pfizer), and at hundreds of other companies.
- It offers a transferable skill that demonstrates experience of both statistics and computing.

R has a steep learning curve, but the benefits of using it are well worth the effort. Take your time and think through every piece of code you type in. The best way to learn R is to take the basic code provided in tutorials and experiment with changing parameters - such as the colour of points in a graph - to really get "under the hood" of the software. Take lots of notes as you go along and if you are getting really frustrated

take a break! To open R click on the start menu and open RStudio. You should see a screen resembling the image below (if it prompts you to update just ignore it for now).

R can be downloaded from https://www.r-project.org/ if it is not on your computer already. Although it is possible to conduct analysis on R directly, you may find it easier to run it via Rstudio which provides a user-friendly graphical user interface. After downloading R, Rstudio can be obtained for free from https://www.rstudio.com/

To open R click on the start menu and open RStudio. You should see a screen resembling the image below (if it prompts you to update just ignore it for now).



It is recommended that you enter your commands into the scripting window of RStudio and use this area as your work space. When you wish to run your commands either hold control Ctrl and enter on your keyboard for each line or select the line you wish to run and click Run at the top of the scripting window.

Our first step is to set the working directory. This is so R knows where to open and save files to. It is recommended that you set the working directory to an appropriate space in your computers work space. In this example it is the same folder as where the Census data pack has been stored. To set the working directory, go to the **Files** table in the Files and Plots window in RStudio. If you click on this tab you can then navigate to the folder you wish to use. You can then click on the **More** button and then **Set as Working Directory**. You should then see some code similar to the below appear in the command line.

Alternatively, you can type in the address of the working directory manually using the `setwd()` function as demonstrated below. This requires you to type in the full address of where your data are stored.

Lines which commence with hashtags (#) are comments. R will not read these through the console but these are useful for annotating your code for your own benefit.

```
#Set the working directory. The bit between the "" needs to specify the path to the
#folder you wish to use you will see my file path below as am example
setwd("C:/Users/Guy/Documents/Teaching/CDRC/Practicals")
# Note the single / (\\ will also work)
```

Our next steps are to load the data into R.

Loading data into R

One of R's great strengths is its ability to load in data from almost any file format. Comma Separated Value (CSV) files are often a preferred choice for data due to their small file sizes and simplicity. We are going to open three different datasets from the Census database. Their codes and dataset names are written in the table in the previous page. We will be downloading Output Area level data, so only files with "oa11" included in their filenames.

We can read CSVs into R using the `read.csv()` function as demonstrated below. This requires us to identify the file location within our workspace, and also assign an object name for our data in R.

```
# loads a csv, remember to correctly input the file location within your working directory
Ethnicity <- read.csv("camden/Camden/tables/KS201EW_oa11.csv")
Rooms <- read.csv("camden/Camden/tables/KS403EW_oa11.csv")
Qualifications <-read.csv("camden/Camden/tables/KS501EW_oa11.csv")
Employment <-read.csv("camden/Camden/tables/KS601EW_oa11.csv")
```

Viewing data

With the data now loaded into RStudio, they can be observed in the objects window. Alternatively, you can open them with the View function as demonstrated below.

```
# to view the top 1000 cases of a data frame
View(Employment)
```

All functions need a series of arguments to be passed to them in order to work. These arguments are typed within the brackets and typically comprise the name of the object (in the examples above its the DOB) that contains the data followed by some parameters. The exact parameters required are listed in the functions' help files. To find the help file for the function type `?` followed by the function name, for example - `?View`

There are two problems with the data. Firstly, the column headers are still codes and are therefore uninformative. Secondly, the data is split between three different data objects.

First let's reduce the data. The Key Statistics tables in the CDRC Census data packs contain both counts and percentages. We will be working with the percentages as the populations of Output Areas are not identical across our sample.

Observing column names

To observe the column names for each dataset we can use a simple `names()` function. It is also possible to work out their order in the columns from observing the results of this function.

```
# view column names of a dataframe
names(Employment)
```

```
##  [1] "GeographyCode" "KS601EW0001"   "KS601EW0002"   "KS601EW0003"
##  [5] "KS601EW0004"   "KS601EW0005"   "KS601EW0006"   "KS601EW0007"
##  [9] "KS601EW0008"   "KS601EW0009"   "KS601EW0010"   "KS601EW0011"
## [13] "KS601EW0012"   "KS601EW0013"   "KS601EW0014"   "KS601EW0015"
## [17] "KS601EW0016"   "KS601EW0017"   "KS601EW0018"   "KS601EW0019"
## [21] "KS601EW0020"   "KS601EW0021"   "KS601EW0022"   "KS601EW0023"
## [25] "KS601EW0024"   "KS601EW0025"   "KS601EW0026"   "KS601EW0027"
## [29] "KS601EW0028"   "KS601EW0029"
```

From using the *variables_description* csv from our datapack, we know the *Economically active: Unemployed* percentage variable is recorded as *KS601EW0019*. This is the 19th column in the Employment dataset.

We will cover more data exploration techniques in the forthcoming practical.

Selecting columns

Next we will create new data objects which only include the columns we require. The new data objects will be given the same name as the original data, therefore overriding the bigger file in R. Using the *variable_description* csv to lookp the codes, we have isolated only the columns we are interested in. Remember we are downloading percentages, not raw counts.

```
# selecting specific columns only
# note this action overwrites the labels you made for the original data,
# so if you make a mistake you will need to reload the data into R

Ethnicity <- Ethnicity[, c(1, 21)]
Rooms <- Rooms[, c(1, 13)]
Employment <- Employment[, c(1, 20)]
Qualifications <- Qualifications[, c(1, 20)]
```

Renaming column headers

Next we want to change the names of the codes to ease our interpretation. We can do this using the `names()`.

If we wanted to change an individual column name we could follow the approach detailed below. In this example, we tell R that we are interested in setting to the name *Unemployed* to the 2nd column header in the data.

```
# to change an individual column name
names(Employment)[2] <- "Unemployed"
```

However, we want to name both column headers in all of our data. To do this we can enter the following code. The `c()` function allows us to concatenate multiple values within one command.

```
# to change both column names
names(Ethnicity)<- c("OA", "White_British")
names(Rooms)<- c("OA", "Low_Occupancy")
names(Employment)<- c("OA", "Unemployed")
names(Qualifications)<- c("OA", "Qualification")
```

Joining data in R

We next want to combine the data into a single dataset. Joining two data frames together requires a common field, or column, between them. In this case it is the OA field. In this field each OA has a unique ID (or OA name), this IDs can be used to identify each OA between each of the datasets. In R the `merge()` function joins two datasets together and creates a new object. As we are seeking to join four datasets we need to undertake multiple steps as follows.

```
#1 Merge Ethnicity and Rooms to create a new object called "merged_data_1"
merged_data_1 <- merge(Ethnicity, Rooms, by="OA")

#2 Merge the "merged_data_1" object with Employment to create a new merged data object
merged_data_2 <- merge(merged_data_1, Employment, by="OA")

#3 Merge the "merged_data_2" object with Qualifications to create a new data object
Census.Data <- merge(merged_data_2, Qualifications, by="OA")
```

```
#4 Remove the "merged_data" objects as we won't need them anymore
rm(merged_data_1, merged_data_2)
```

Our newly formed *Census.Data* object contains all four variables.

Exporting Data

You can now save this file to your workspace folder. We will use this data in the forthcoming practicals. Remember R is case sensitive so take note of when object names are capitalised.

```
# Writes the data to a csv named "practical_data" in your file directory
write.csv(Census.Data, "practical_data.csv", row.names=F)
```

Practical 2: Data exploration in R

This practical will introduce you to the data exploration techniques which are useful for deriving an understanding of large numerical variables in R. We will introduce you to some useful R commands which allow you to observe the data efficiently and also create descriptive statistics. We will then visualise the distribution(s) of our data through the creation of univariate plots. Data for the practical can be downloaded from the **Introduction to Spatial Data Analysis and Visualisation in R** homepage.

In this tutorial we will:

- View and explore the data
- Create descriptive statistics
- Observe and compare the data using univariate plots
- Install and load an R package to use bespoke functions

Before we start, we need to do two things. First, we need to set the working directory.

```
# Set the working directory (remember to change this to your file path)
setwd("C:/Users/Guy/Documents/Teaching/CDRC/Practicals")
```

Second, load the data we saved in the previous practical.

```
# Load the data created in the previous practical
Census.Data <-read.csv("practical_data.csv")
```

Exploring the data

There are several ways to view data, some of been exemplified below. Remember R is case sensitive.To view the *Census.Data* object type:

```
# prints the data within the console
print(Census.Data)
```

We can also select which columns and rows we wish to print by entering the numerical ranges of the array within square brackets after the variable name (i.e. `Census.Data[n,n]`) where the comma separates the rows and columns. In this example we are selecting rows 1 to 20 and columns 1 to 5. As we are selecting all of the columns in the data we could just leave the space after the comma in the square brackets blank.

```
# prints the selected data within the console
print(Census.Data[1:20,1:5])
```

```
##           OA White_British Low_Occupancy Unemployed Qualification
## 1  E00004120      42.35669     6.2937063  1.8939394      73.62637
## 2  E00004121      47.20000     5.9322034  2.6881720      69.90291
## 3  E00004122      40.67797     2.9126214  1.2121212      67.58242
## 4  E00004123      49.66216     0.9259259  2.8037383      60.77586
## 5  E00004124      51.13636     2.0000000  3.8167939      65.98639
## 6  E00004125      41.41791     3.9325843  3.8461538      74.20635
## 7  E00004126      48.54015     5.5555556  4.5454545      62.44726
## 8  E00004127      48.67925     8.8709677  0.9389671      60.35242
## 9  E00004128      45.39249     2.4844720  2.1645022      70.07874
## 10 E00004129      49.05660     3.5211268  4.3103448      66.66667
## 11 E00004130      38.80597     6.2500000  0.9174312      66.66667
## 12 E00004131      39.64286     7.5630252  1.8691589      64.47368
## 13 E00004132      55.88235     4.3478261  3.7974684      73.49398
## 14 E00004133      41.96078     7.6271186  1.9900498      65.38462
## 15 E00004134      53.19149     6.0000000  2.7027027      72.89157
```

```
## 16 E00004135      46.85315     4.7619048  3.7313433      74.82014
## 17 E00004136      59.64912     0.9090909  2.7322404      73.68421
## 18 E00004137      48.16176     5.4421769  2.7522936      69.06780
## 19 E00004138      42.22222     2.8169014  4.9723757      58.16327
## 20 E00004139      17.71772    64.2857143 15.9420290      22.96651
```

You can also open the data in R using the `View()` function. This will create a clearly formatted table in a new window which displays the top 1000 cases.

```
# to view the top 1000 cases of a data frame
View(Census.Data)
```

If the data is very large and opening it could be computationally intensive - you could just opt to open the top or bottom n cases.

The `head()` and `tail()` commands open the top and bottom n cases respectively.

```
head(Census.Data)
```

```
##          OA White_British Low_Occupancy Unemployed Qualification
## 1 E00004120      42.35669     6.2937063   1.893939      73.62637
## 2 E00004121      47.20000     5.9322034   2.688172      69.90291
## 3 E00004122      40.67797     2.9126214   1.212121      67.58242
## 4 E00004123      49.66216     0.9259259   2.803738      60.77586
## 5 E00004124      51.13636     2.0000000   3.816794      65.98639
## 6 E00004125      41.41791     3.9325843   3.846154      74.20635
```

```
tail(Census.Data)
```

```
##            OA White_British Low_Occupancy Unemployed Qualification
## 744 E00174675     37.354086      9.401709   2.714932      52.81385
## 745 E00174676      7.881773      9.868421   0.500000      37.12871
## 746 E00174677     22.520107      8.125000   4.528302      50.67568
## 747 E00174678     23.949580      6.194690   1.421801      53.21101
## 748 E00174679     24.271845      4.081633   1.663894      45.34884
## 749 E00174680     36.514523     25.274725   8.108108      24.74227
```

It is also easy to observe the number of rows and columns, and the column headers.

```
#Get the number of columns
ncol(Census.Data)
```

```
## [1] 5
```

```
#Get the number of rows
nrow(Census.Data)
```

```
## [1] 749
```

```
#List the column headings
names(Census.Data)
```

```
## [1] "OA"            "White_British" "Low_Occupancy" "Unemployed"
## [5] "Qualification"
```

Whilst it is informative to open data, it is often difficult to generalise key trends just by looking at the numbers.

Descriptive statistics

Descriptive statistics are a useful means of deriving quick information about a collective dataset. A good introduction to descriptive statistics is available in the online version of Statistical Analysis Handbook (de Smith, 2015) which includes detailed descriptions of measures of central tendecy and measures of spread

Notice that below we use the $ symbol to select a single variable from the *Census.Data* object. If you type in `Census.Data$` (so the name of your data object followed by a $ sign), then press tab on your keyboard, RStudio will let you select a variable from a drop down window. Repeat this step for your qualifications variable.

```
mean(Census.Data$Unemployed)
```

```
## [1] 4.510309
```

```
median(Census.Data$Unemployed)
```

```
## [1] 4.186047
```

```
range(Census.Data$Unemployed)
```

```
## [1]  0.00000 18.62348
```

A useful function for descriptive statistics is `summary()` which will produce multiple descriptive statistics as a single output. It can also be run for multiple variables or an entire data object.

```
#mean, median, 25th and 75th quartiles, min, max
summary(Census.Data)
```

```
##          OA      White_British    Low_Occupancy      Unemployed
##  E00004120:  1   Min.   : 7.882   Min.   : 0.000   Min.   : 0.000
##  E00004121:  1   1st Qu.:35.915   1st Qu.: 6.015   1st Qu.: 2.500
##  E00004122:  1   Median :44.541   Median :10.000   Median : 4.186
##  E00004123:  1   Mean   :44.832   Mean   :11.597   Mean   : 4.510
##  E00004124:  1   3rd Qu.:54.472   3rd Qu.:16.107   3rd Qu.: 6.158
##  E00004125:  1   Max.   :78.035   Max.   :64.286   Max.   :18.623
##  (Other)  :743
##  Qualification
##  Min.   :11.64
##  1st Qu.:36.32
##  Median :55.10
##  Mean   :51.43
##  3rd Qu.:66.23
##  Max.   :88.07
##
```

Univariate plots

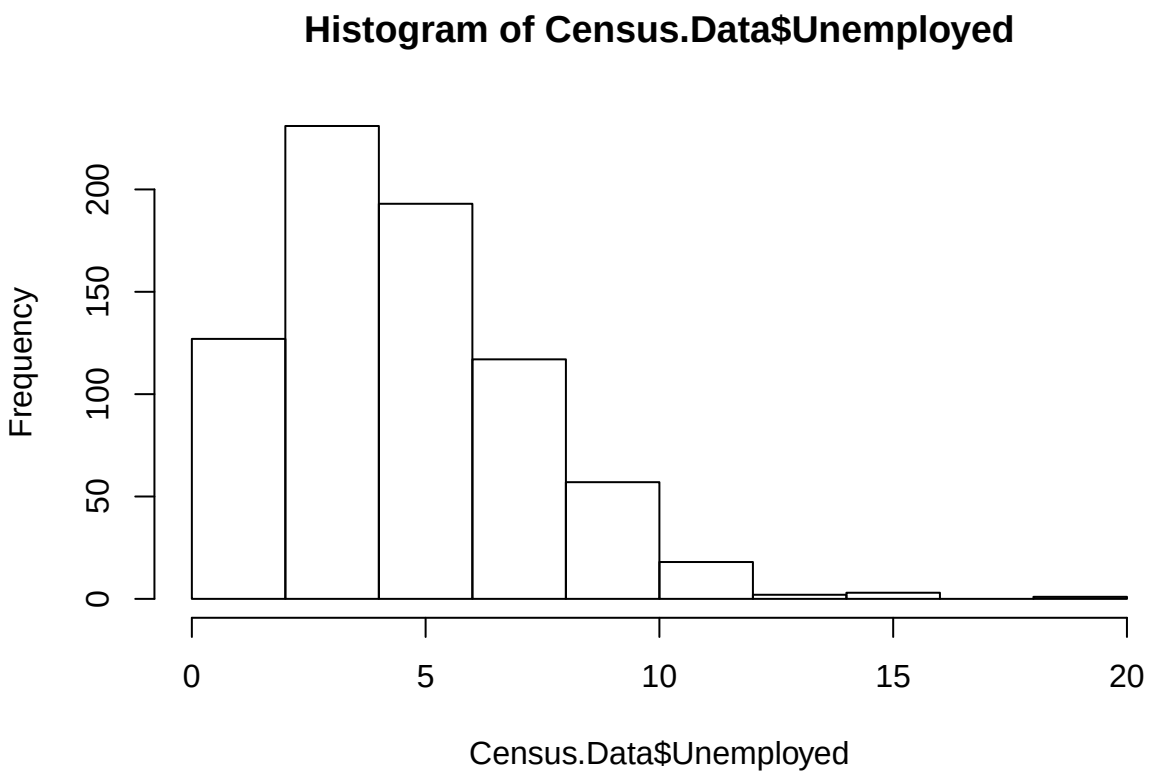
Univariate plots are a useful means of conveying the distribution of a particular variable. Many of these can be produced very simply in R.

Histograms

Histograms are perhaps the most informative means of visualising a univariate distribution. In the example below we will create a histogram for the *unemployed* variable using the `hist()` function (remember if you

put a $ symbol followed by the column header after the data object, the function in R will read that column only). Repeat this step for your qualifications variable.

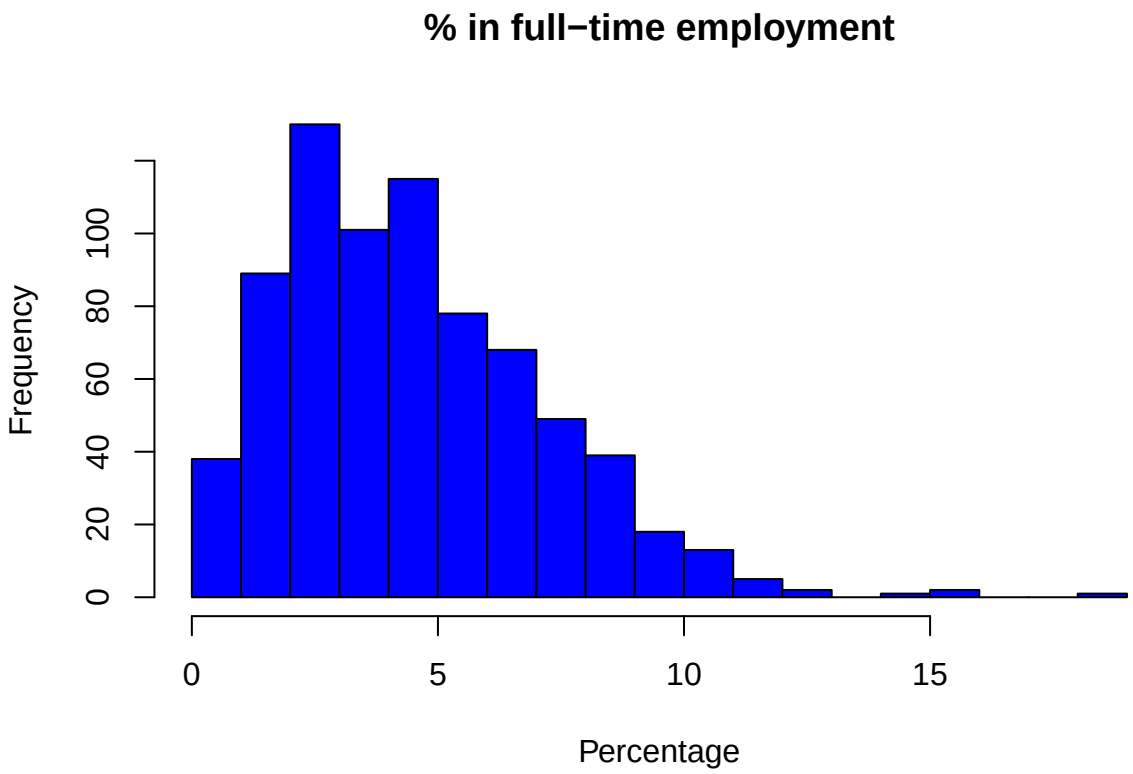
```
# Creates a histogram
hist(Census.Data$Unemployed)
```



The histogram should appear in the *Plots* window of RStudio.

We can tidy up the histogram by including the following parameters within the `hist()` function. In the example below we specify the number of data breaks in the chart (`breaks`), colour the chart blue (`col`), create a title (`main`) and label the x-axis (`xlab`). For more information on all of the parameters of the function just type `?hist` into R and run it.

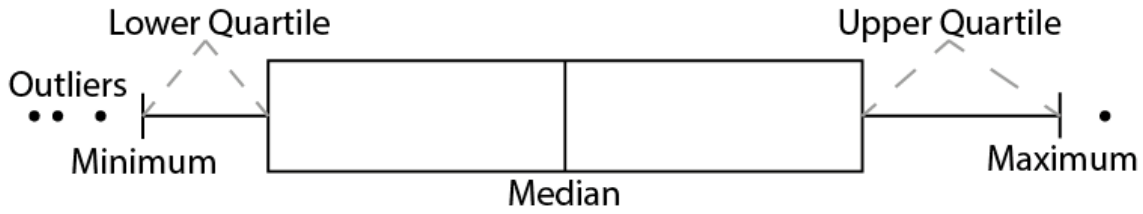
```
# Creates a histogram, enters more commands about the visualisation
hist(Census.Data$Unemployed, breaks=20, col= "blue",
     main="% in full-time employment", xlab="Percentage")
```



Notice that in the example above, we have specified that there are 20 breaks (or columns in the graph). The higher the number of breaks, the more intricate and complex a histogram becomes. Try producing a histogram with 50 breaks to observe the difference.

Boxplots

In addition to histograms, another type of plot that shows the core characteristics of the distribution of values within a dataset, and includes some of the `summary()` information we generated earlier, is a box and whisker plot (box plot for short).



Box plots can be created in R by sampling running the `boxplot()` function. For more information on the parameters of this function type `?boxplot` into R.

In the example below we are creating multiple box plots in order to compare the four main variables. To

select the variables we have used an array after *Census.Data* so that the function doesn't read the row names too. We could also call indiviudal rows i.e. `boxplot(Census.Data$Unemployed)` or even pairs of variables i.e. `boxplot(Census.Data$Unemployed, Census.Data$Qualification)`.

```
# box and whisker plots
boxplot(Census.Data[,2:5])
```

Installing the vioplot package and creating a violin plot

One of the main benefits of R is that as an open source tool, there is much documentation on how to complete more advanced tasks online. In addition to R's core functions, there are also a large volume of bespoke packages which included their own niche functions. These packages can be downloaded and installed for free using R.

In this case we want to create a type of univariate plot known as a violin plot. In its simplest form, a violin plot combines both box plots and histograms in one graphic. It is also possible to input multiple plots in a single image in order to draw comparisons. However, there is no function to create these plots in the standard R library so we need to install a new package.

Step 1: Install the package

To install go to **Tools > Install packages.** in RStudio and enter *vioplot*. Alternatively, run `install.packages()` as demonstrated below,

```
# When you hit enter R will ask you to select a mirror to download the package contents
# from. It doesn't really matter which one you choose, I tend to pick the UK based ones.
```

```
install.packages("vioplot")
```

The `install.packages` step only needs to be performed once. You don't need to install a package every time you want to use it. However, each time you open R and wish to use a package you need to use the `library()` command to tell R that it will be required.

Step 2: Open the package

To ensure that R connects to the package and the new functions are activated you need to activate the package. This can be done using the `library()` or `require()` packages. Simply enter the name of the downloaded package within the brackets of either function.

```
# loads a package
library(vioplot)
```

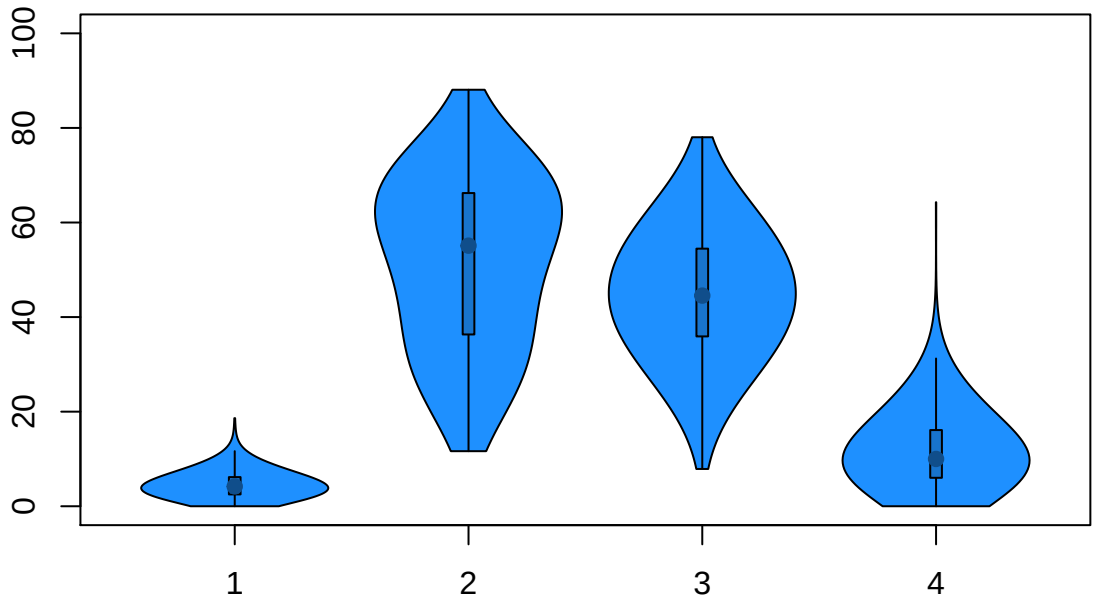
```
## Loading required package: sm
```

```
## Package 'sm', version 2.2-5.4: type help(sm) for summary information
```

Step 3: Using newly installed functions

Now we are ready to use the newly available `vioplot()` function. Remember you can run `?vioplot` to explore the parameters of the function. In its very simplest form you can just write `vioplot(Census.Data$Unemployed)` to create a simple plot for the Unemployed variable. However, the example below includes all four variables and a couple of extra parameters. The `ylim` command allows you to set the upper and lower limits of the y-axis (in this case 0 and 100 as all data is percentages). Three colours were also assigned for different parts of the plot.

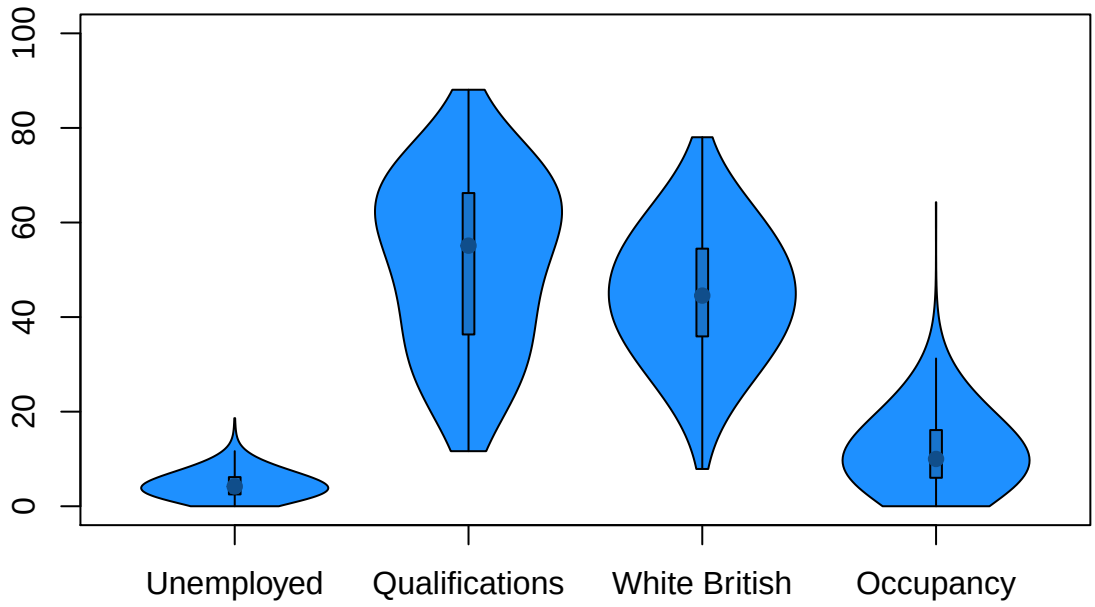
```
# creates a violin plot for 4 variables, uses 3 shades of blue
vioplot(Census.Data$Unemployed, Census.Data$Qualification, Census.Data$White_British,
        Census.Data$Low_Occupancy, ylim=c(0,100),
        col = "dodgerblue", rectCol="dodgerblue3", colMed="dodgerblue4")
```



Recreate the violin plots using different colours. Colours can be specified in various different forms such as predefined names (as demonstrated above) or using RGB or HEX colour codes. This PDF outlines the names of many colours in R and may be useful for you: http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf. For more information on graphical parameters please see: http://www.statmethods.net/advgraphs/parameters.html

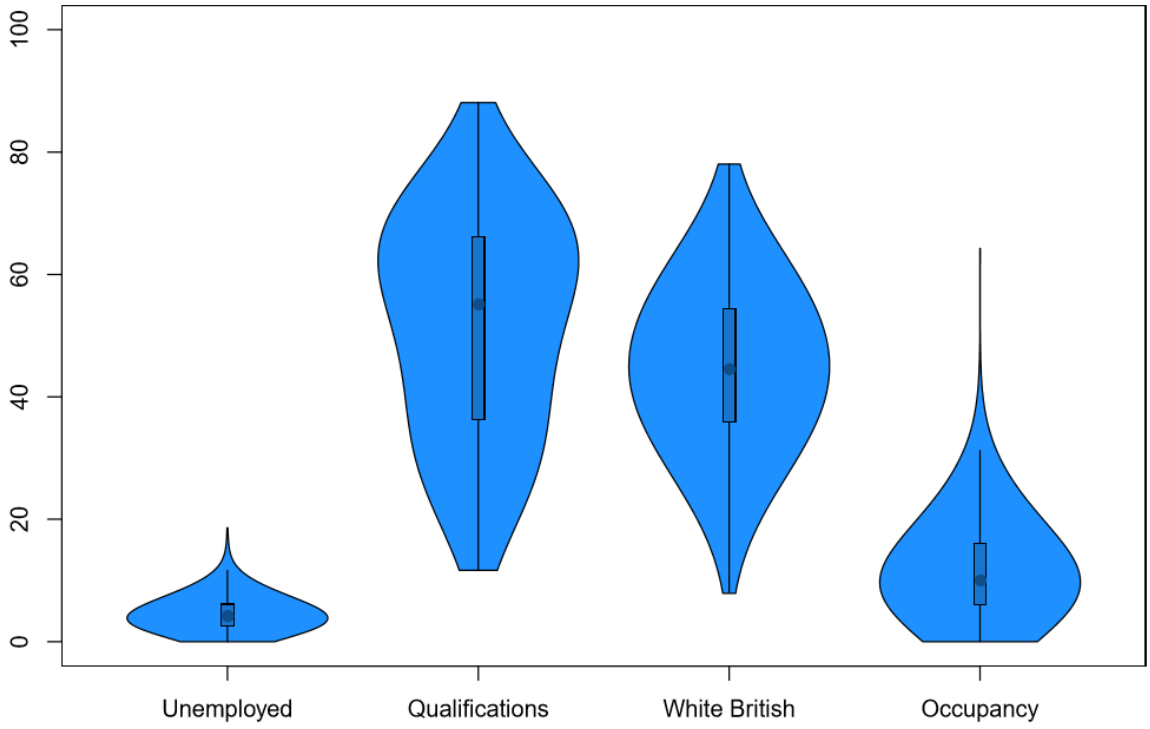
Finally, create labels for each of the data in the graphic. We can do this using the names command within the `vioplot()` function as demonstrated below.

```
# add names to the plot
vioplot(Census.Data$Unemployed, Census.Data$Qualification, Census.Data$White_British,
        Census.Data$Low_Occupancy, ylim=c(0,100),
        col = "dodgerblue", rectCol="dodgerblue3", colMed="dodgerblue4",
        names=c("Unemployed", "Qualifications", "White British", "Occupancy"))
```



Exporting images

You can export the images from R if you want to observe them in a much higher quality. You can do this by clicking on **Export** within the **Plots** window in RStudio. PDF versions are exported in a very high quality. Below is an example of an exported version of the image above.


100
80
60
40
20
0
Unemployed
Qualifications
White British
Occupancy

Practical 3: Bivariate Plots in R

This practical is intended to introduce you to some of the basic techniques used to create two-dimensional plots in R. Data for the practical can be downloaded from the **Introduction to Spatial Data Analysis and Visualisation in R** homepage.

In this tutorial we will:

- Create a simple scatter plot
- Create a symbols plot
- Create plots in the ggplot package

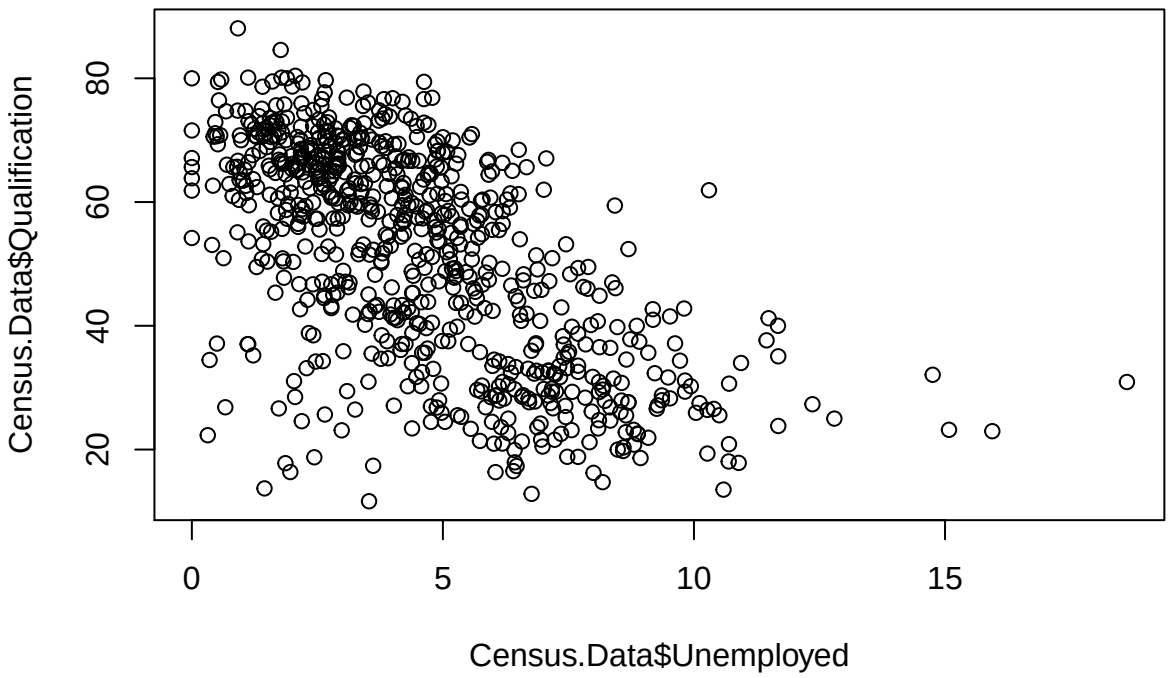
First, we must set the working directory and load the practical data.

```
#Set the working directory.
setwd("C:/Users/Guy/Documents/Teaching/CDRC/Practicals")

#Load the data. You may need to alter the file directory
Census.Data <-read.csv("practical_data.csv")
```

Simple scatter plots

The most basic scatter plots in R can be made using the `plot()` function which only requires you to identify two variables within the function's parameters. To do this follow the example below.

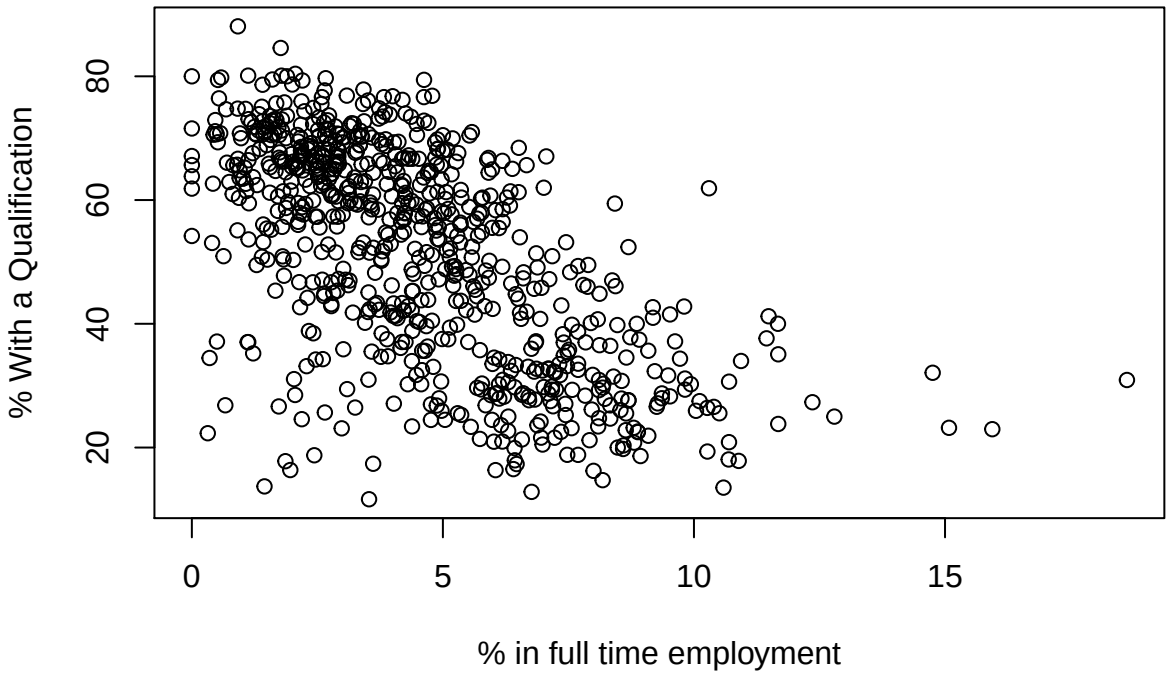
```
#left of the comma is the x-axis, right is the y-axis. Also note how we are using the
#$ command to select the columns of the data frame we want.

plot(Census.Data$Unemployed,Census.Data$Qualification)
```



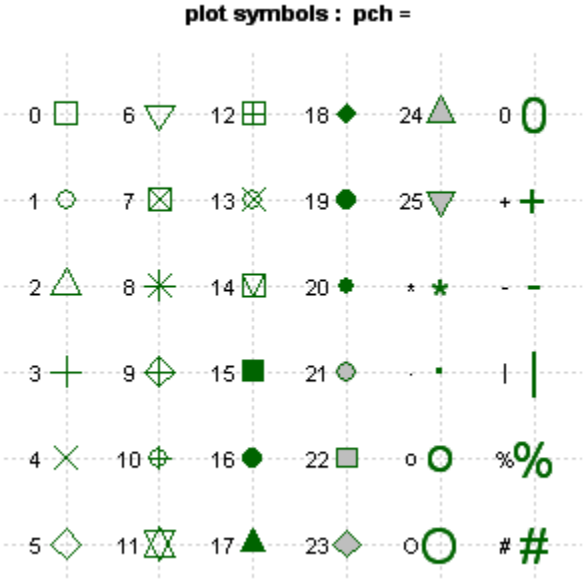
Each point represents data from the two variables for a unique output area. From this chart it is possible to infer that there is a distinctive negative relationship between the percentage of those with *level 4 qualifications and above* and the percentage of those *unemployed* at the output area level. As level 4 qualifications refer to certficates of higher education we could expect there to be an inverse relationship between this variable and local unemployment rates.

Remember to observe the available parameters for a function we can enter `?plot` into R. We will now include axis labels.

```
# includes axis labels
plot(Census.Data$Unemployed,Census.Data$Qualification, xlab="% in full time employment",
     ylab="% With a Qualification")
```

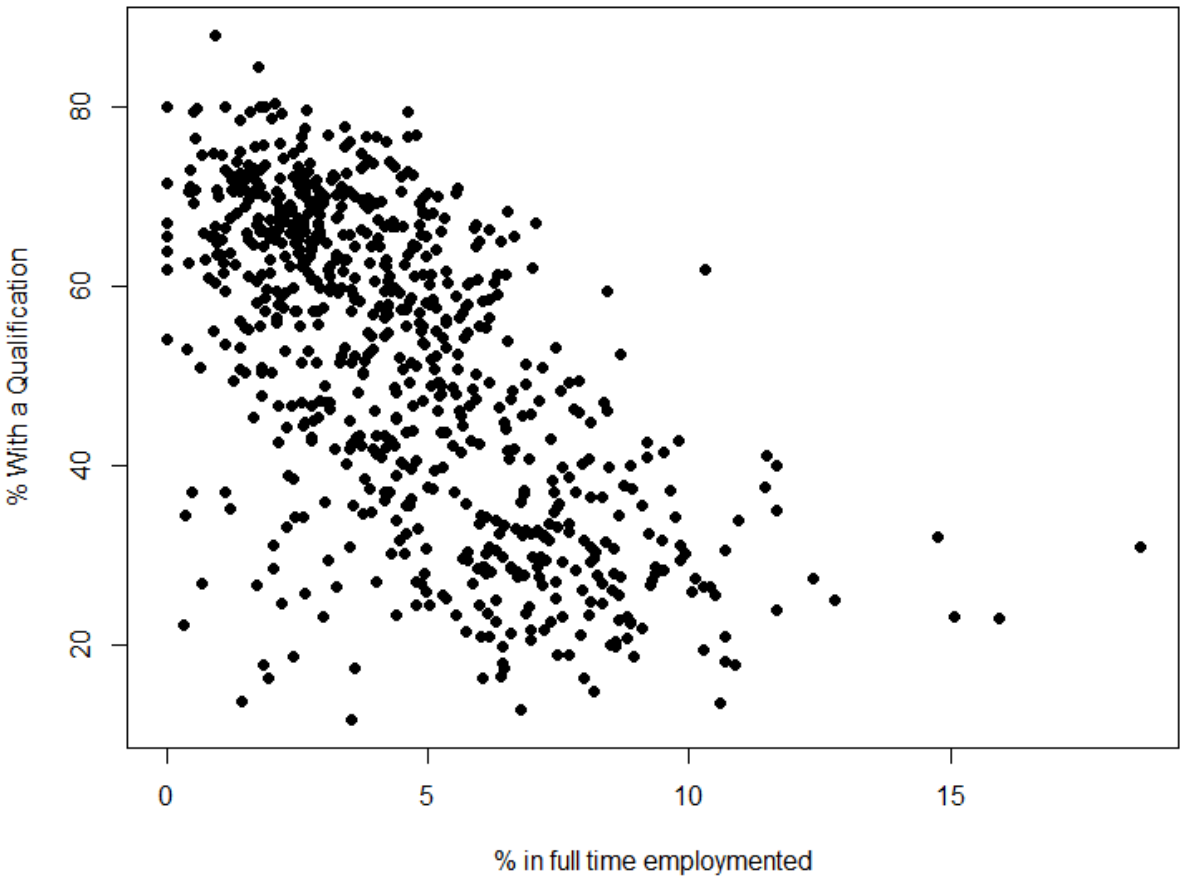


One of the parameters for the `plot()` function is plotting symbols (pch). Details of what symbols are available can be found in the following link: http://www.statmethods.net/advgraphs/parameters.html



If you want to change the default of hollow circles to filled circles, you can just add `pch = 19` to the end of the function (remember to insert a comma after the previous command). The output of this is demonstrated

below. Try changing the plots to triangles with your data.



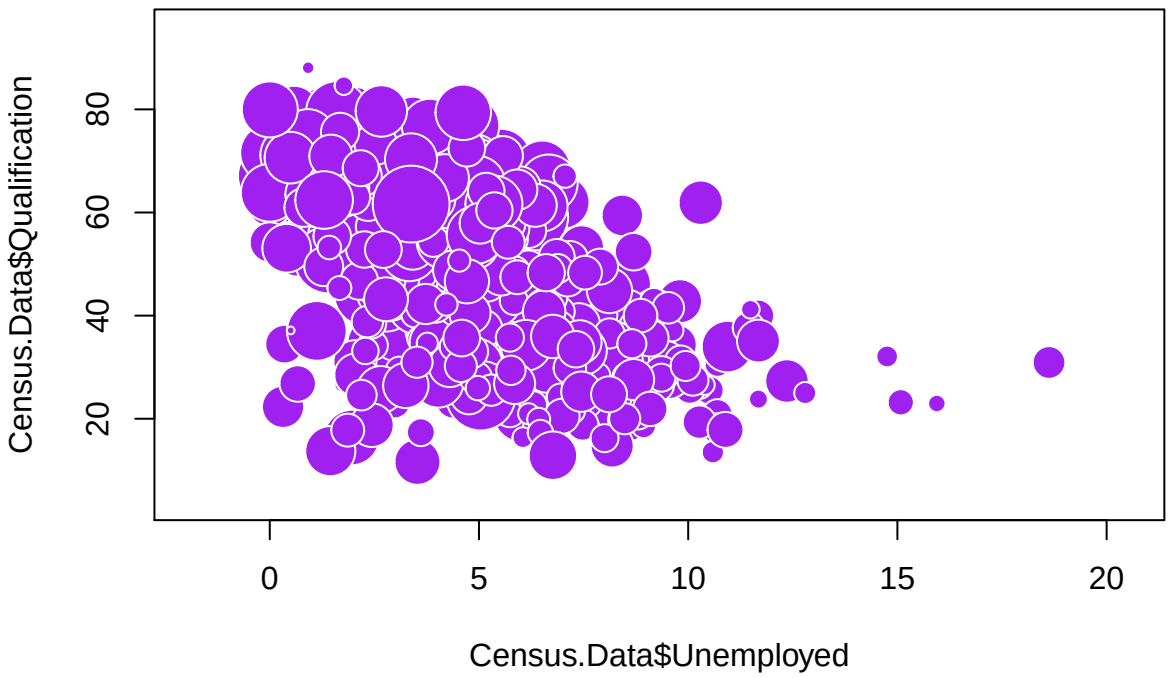
Symbols plot

There is also the possibility of creating a proportional symbols plot using the `symbols()` function which shares much of the parameters as the `plots()` function. This allows us to consider a third dimenion (aka a third variable) in our two-dimensional plot.

In this example we will set the *percentage of White British persons* as the size dependent variable.

Now using the `symbols()` function we will set the symbols as circles and use the *percentage of White British persons* to define the proportional sizes of them in the chart. We have also defined the foreground(`fg`) and background(`bg`) colours of the symbols. The inches command allows you to restrict the overall size of all the symbols. Try changing some of these commands below.

```
# Create a proportional symbols plot
symbols(Census.Data$Unemployed,Census.Data$Qualification,
        circles = Census.Data$White_British,
        fg="white", bg ="purple", inches = 0.2)
```



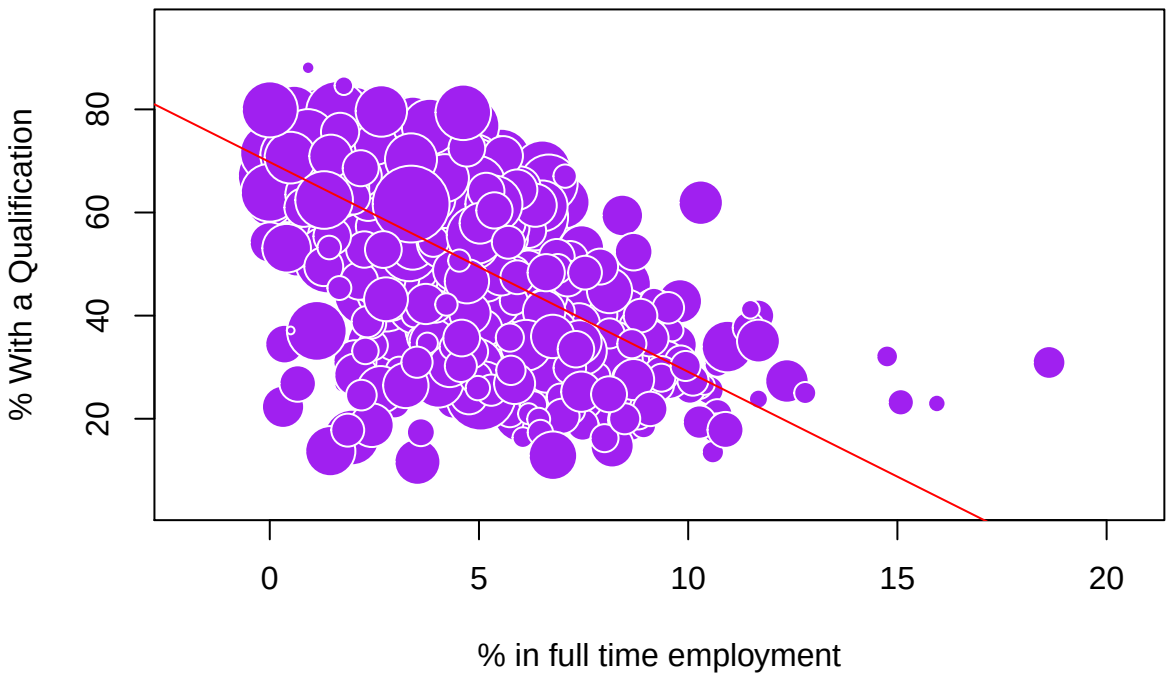
In the graphic, larger circles represent areas of greater percentages of the White British population. Can you depict a relationship between this variable and the other two in the plot?

There are still improvements that can be made to the plot. For instance adding x-axis and y-axis labels (`xlab`, `ylab`).

Adding a regression line

It is also possible to insert a regression line to plots in R. This requires you to run a quick linear regression model which you can use to plot a straight line of best fit in the chart. This can be done in R by adding (+) a linear model (`lm()`) function to our plot. The outputted regression line is then represented using the 'abline()' function. We will cover the regression model in more detail in a later practical.

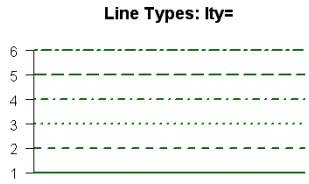
```
# bubble plot
symbols(Census.Data$Unemployed, Census.Data$Qualification,
        circles = Census.Data$White_British,
        fg="white", bg ="purple", inches = 0.2,  xlab="% in full time employment",
        ylab="% With a Qualification") +
# adds a regression line, sets the colour to red
abline(lm(Census.Data$Qualification~ Census.Data$Unemployed), col="red")
```



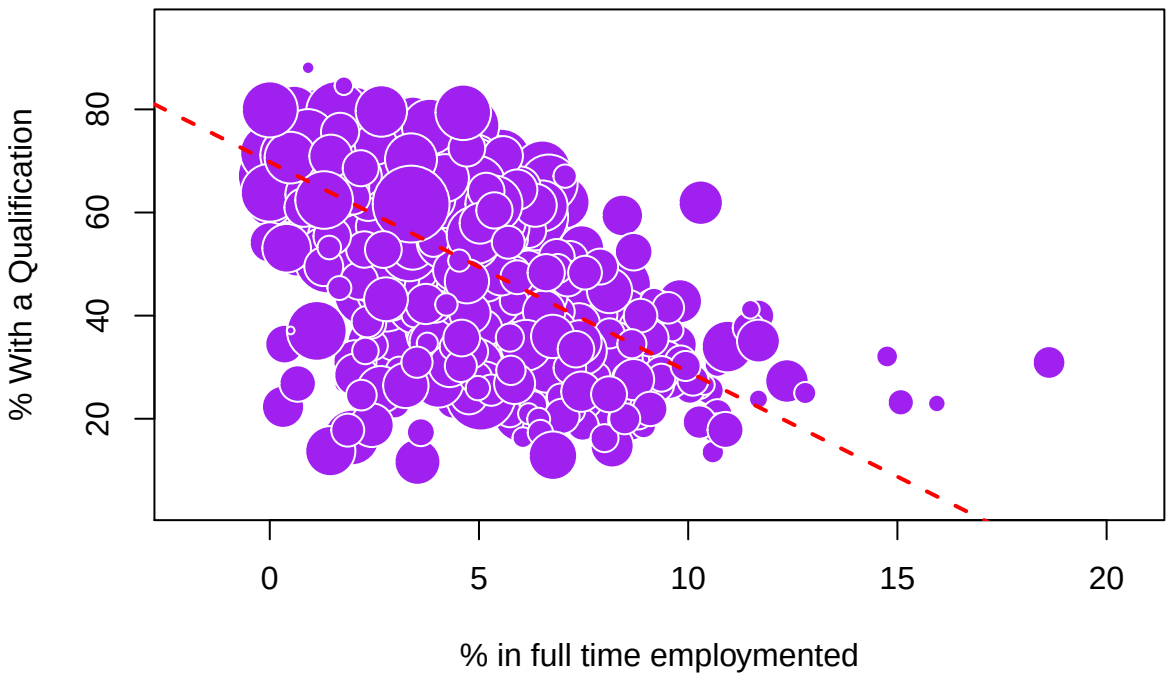

```
## numeric(0)
```

We can also edit the line using the line type (`lty`) and line width (`lwd`) commands from the `abline()` function. These are demonstrated below using an image from http://www.statmethods.net/advgraphs/parameters.html

option	description
lty	line type. see the chart below.
lwd	line width relative to the default (default=1). 2 is twice as wide.




```
# a bubble plot with a dotted regression line
symbols(Census.Data$Unemployed, Census.Data$Qualification,
        circles = Census.Data$White_British,
        fg="white", bg ="purple", inches = 0.2,  xlab="% in full time employmented",
        ylab="% With a Qualification") +
  abline(lm(Census.Data$Qualification~ Census.Data$Unemployed), col="red", lwd=2, lty=2)
```




```
## numeric(0)
```

Using the ggplot2 package

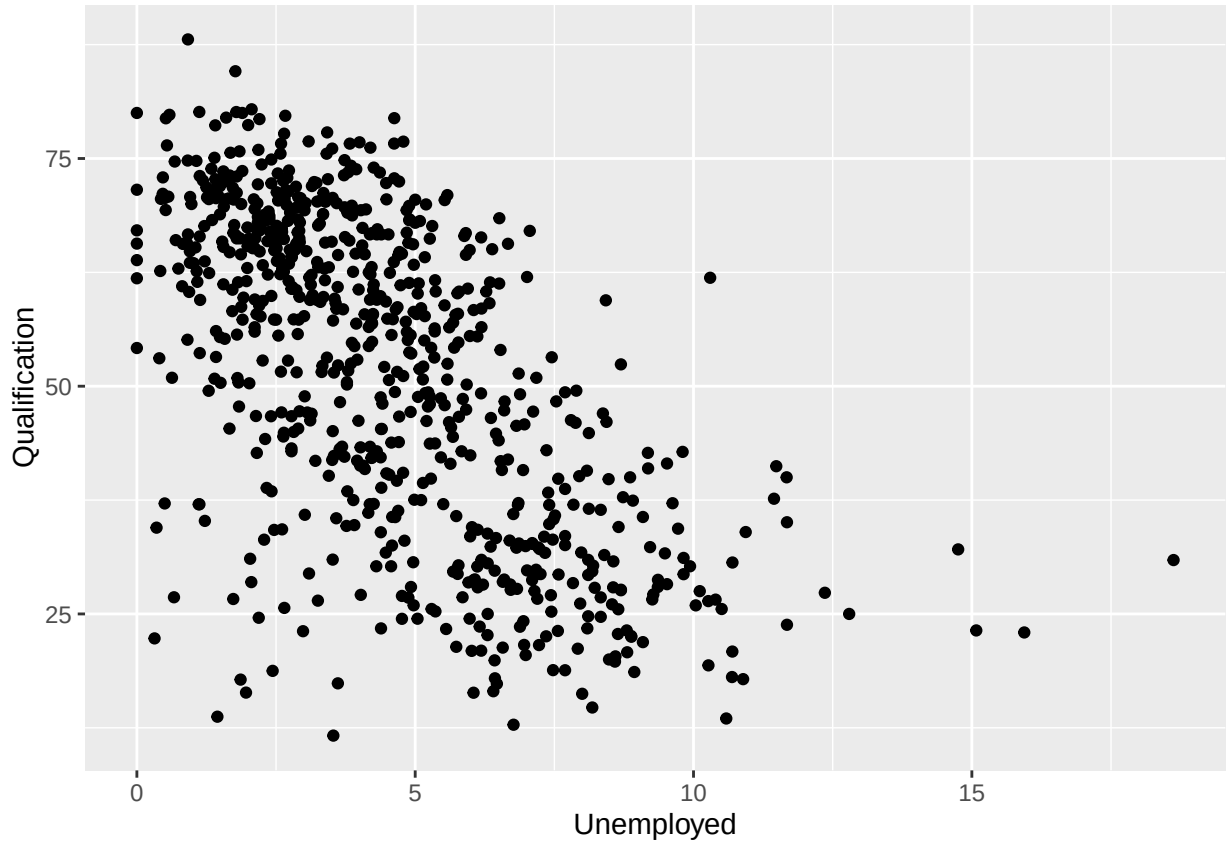
A slightly different method of creating plots in R requires the ggplot2 package. As the commands from this package are not already in R, we need to install ggplot2 and then load it. Look at your notes from Practical 2 about how to install packages.

```
# Loads an installed package
library("ggplot2")
```

The package is an implementation of the *Grammar of Graphics* (Wilkinson, 2005) - a general scheme for data visualisation that breaks up graphs into semantic components such as scales and layers. ggplot2 can serve as a replacement for the base graphics in R and contains a number of default options that match good visualisation practice.

We will first create a simple scatter plot using `ggplot()`

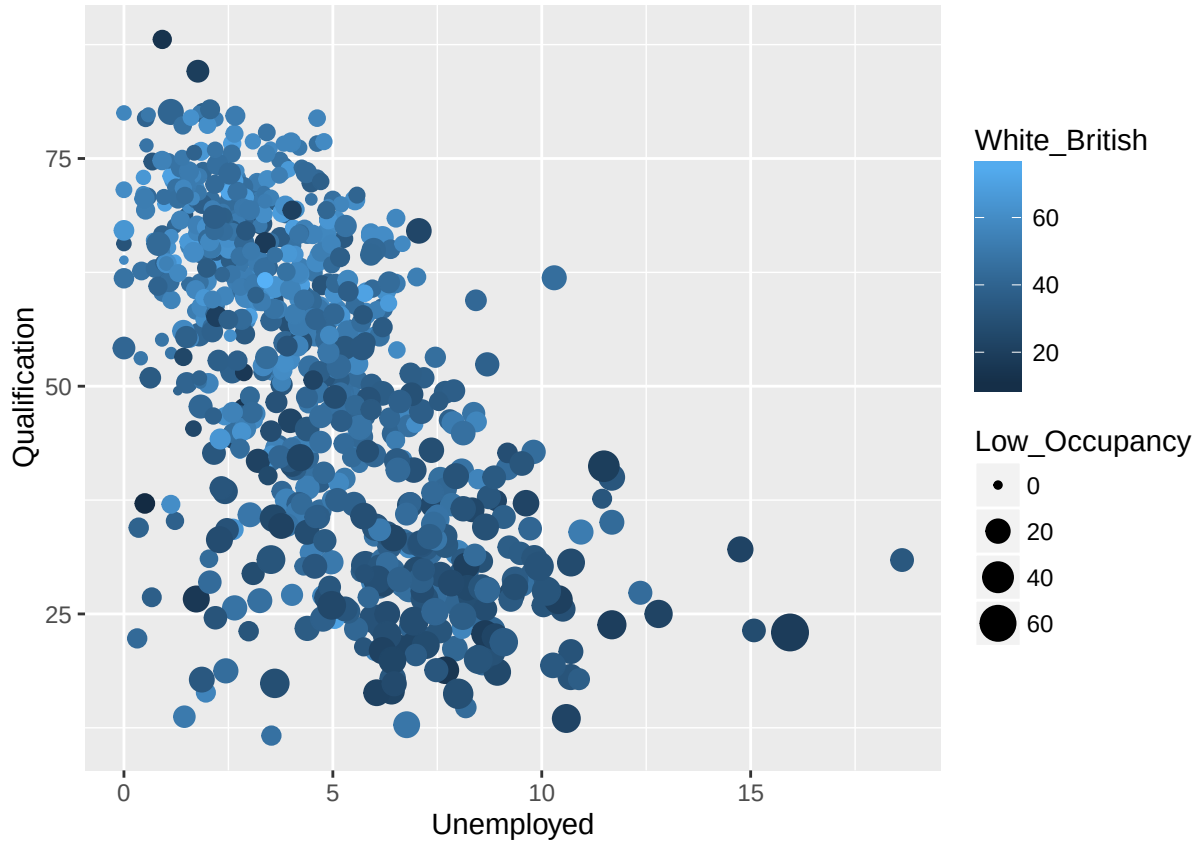
```
p <- ggplot(Census.Data, aes(Unemployed,Qualification))
p + geom_point()
```



Notice that here we assign the data object, then in the `aes()` command we set the two variables. The points are also inputted as a separate function to the `ggplot()` function.

We can also set various parameters for the points such size and colour. For example in the code below the colours are proportional to the percentage of the White British population, whilst the size is proportional to the percentage of homes with a low occupation rating (i.e. overcrowded). Here it is possible to observe four different variables in one two-dimensional chart.

```
p <- ggplot(Census.Data, aes(Unemployed,Qualification))
p + geom_point(aes(colour = White_British, size = Low_Occupancy))
```


Qualification
75
50
25
0
5
10
15
Unemployed
White_British
60
40
20
Low_Occupancy
0
20
40
60

Practical 4: Finding Relationships in R

This practical is intended to introduce you to some of the most commonly used means of statistically identifying and measuring bivariate and multivariate relationships in R. Data for the practical can be downloaded from the **Introduction to Spatial Data Analysis and Visualisation in R** homepage.

In this tutorial we will:

- Run a Pearson's correlation test
- Run a Spearman's correlation test
- Run a linear regression model

First we must set the working directory and load the practical data. Remember to change the working directory to the correct file path on your computer

```
#Set the working directory. Remember to alter the file path below.
setwd("C:/Users/Guy/Documents/Teaching/CDRC/Practicals")

#Load the data. You may need to alter the file directory
Census.Data <-read.csv("practical_data.csv")
```

Bivariate correlations

One common means of identifying and measuring the relationship between two variables is a correlation. In R this can simply be done by using the `cor()` function and inputting two variables within the parameters - i.e:

```
# Runs a Pearson's correlation
cor(Census.Data$Unemployed, Census.Data$Qualification)
```

This will return a Pearson's correlation coefficient(r). A Pearson's (or Product Moment Correlation) coefficient is a measure of linear association between two variables. Greater values represent a stronger relationship between the pair. *1* represents a perfect positive relationship, *0* represents no linear correlation and *-1* represents a perfect negative relationship.

In R, a better option is to use `cor.test()` as this also reports significance statistics. If a test is not statistically significant, its results cannot be regarded as reliable. Here is an example below.

```
# Runs a Pearson's correlation
cor.test(Census.Data$Unemployed, Census.Data$Qualification)
```

```
##
##  Pearson's product-moment correlation
##
## data:  Census.Data$Unemployed and Census.Data$Qualification
## t = -21.85, df = 747, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  -0.6662641 -0.5786800
## sample estimates:
##       cor
## -0.624431
```

The final value is the Pearson's correlation coefficient. A score of -0.62 identifies that there is a negative relationship between the unemployment and qualification variables. From the model we also get the 95% confidence intervals. Confidence intervals display the range of values of which there is a defined probability that the coefficient falls within. The output also returns the result of the t-test. We can use this to determine if the results were statistically significant.

A Pearson's correlation is only suitable when the relationship between the two variables is linear. It is not sensitive to relationships that are non-linear. In these circumstances it is worth using Spearman's rank correlation. This statistic is obtained by simply replacing the observations by their rank within their sample and computing the correlation, which means it is also suitable for large-scale ordinal variables.

```
# Runs a Spearman's correlation
cor.test(Census.Data$Unemployed, Census.Data$Qualification, method="spearman")
```

```
## Warning in cor.test.default(Census.Data$Unemployed, Census.Data
## $Qualification, : Cannot compute exact p-value with ties
```

```
##
##  Spearman's rank correlation rho
##
## data:  Census.Data$Unemployed and Census.Data$Qualification
## S = 113730000, p-value < 2.2e-16
## alternative hypothesis: true rho is not equal to 0
## sample estimates:
##        rho
## -0.6240406
```

Does your conclusion about the relationship between these two variables change when using a Spearman's correlation compared with a Pearson's correlation?

It is also possible to produce a correlation pair-wise matrix in R. This will display a correlation coefficient for every possible pairing of variables in the data. To do this we need to first format the data to get rid of the ID column as it will not work in a correlation. We only want to include the variables for our correlation matrix.

```
# creates a data1 object which does not include the 1st column from the original data
data1 <- Census.Data[,2:5]
```

Then with our new data1 object we can create a new matrix.

```
# creates correlation matrix
cor(data1)
```

```
##                White_British Low_Occupancy Unemployed Qualification
## White_British      1.0000000    -0.6006639 -0.3984454     0.4992319
## Low_Occupancy     -0.6006639     1.0000000  0.6408021    -0.7347354
## Unemployed        -0.3984454     0.6408021  1.0000000    -0.6244310
## Qualification      0.4992319    -0.7347354 -0.6244310     1.0000000
```

Remember coefficients of 1 will always be produced between identical variables as they display the same patterns.

We can use the `round()` function to round our results to 2 decimal places.

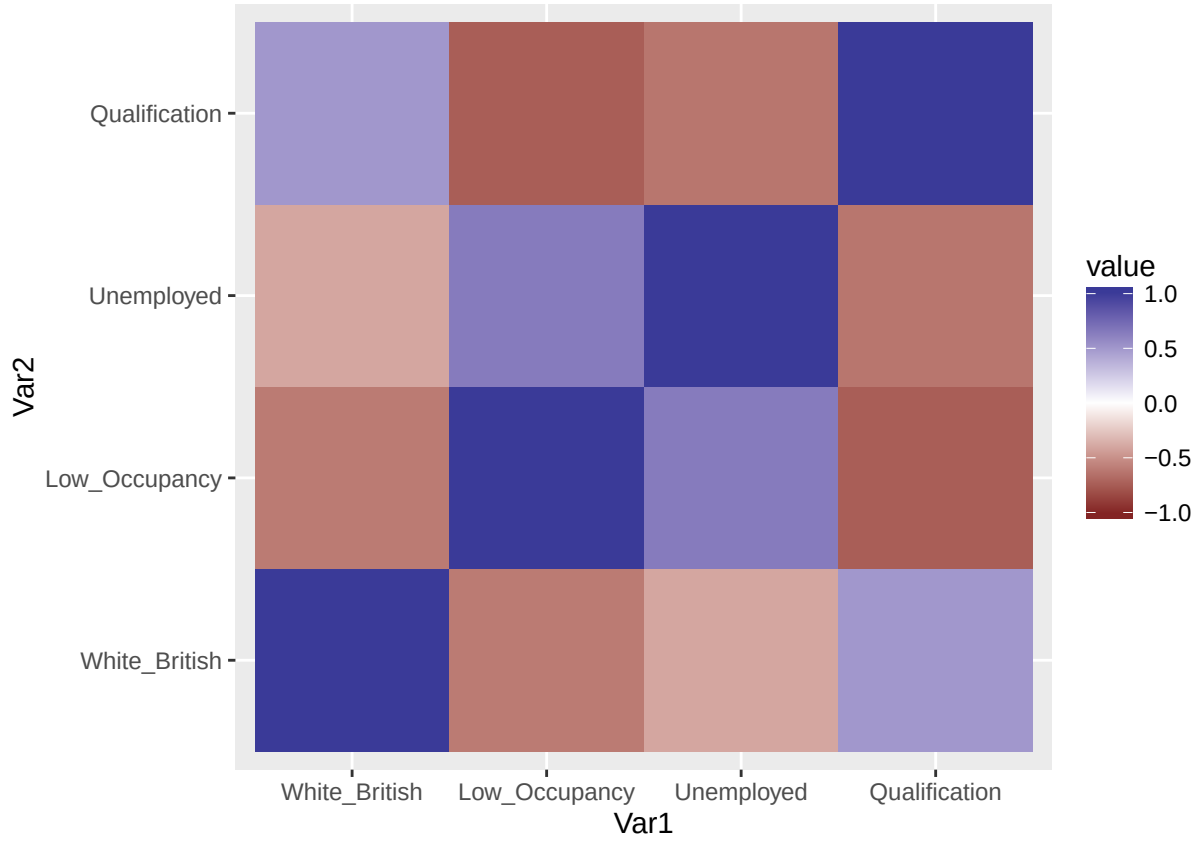
```
# creates correlation matrix
round(cor(data1),2)
```

```
##                White_British Low_Occupancy Unemployed Qualification
## White_British           1.0         -0.60      -0.40          0.50
## Low_Occupancy          -0.6          1.00       0.64         -0.73
## Unemployed             -0.4          0.64       1.00         -0.62
## Qualification           0.5         -0.73      -0.62          1.00
```

We can use the `qplot()` function from the ggplot2 package to create a heat map of this correlation matrix. The code to do this is written below.

```
library(ggplot2) # should already be opened from the previous stage
library(reshape2)
```

```
qplot(x=Var1, y=Var2, data=melt(cor(data1, use="p")), fill=value, geom="tile") +
   scale_fill_gradient2(limits=c(-1, 1))
```



Heat maps like this can be a simple and effective means of conveying lots of information in one graphic. In this case, the correlation matrix is already quite small so obscuring the numerical values with colours is not necessary. But, if you have very large matrices of say 50+ cells this could be a useful technique.

Regression analysis

A simple linear regression plots a single straight line of predicted values as the model for a relationship. It is a simplification of the real world and its processes, that assumes that there is approximately a linear relationship between X and Y.

Another way of thinking about this line is as the best possible summary of the cloud of points that are represented in the scatterplot (if we can assume that a straight line would do a good job doing this). If I were to tell you to draw a straight line that best represents this pattern of points the regression line would be the one that best does it (if certain assumptions are met). The linear model then is a model that takes the form of the equation of a straight line through the data. The line does not go through all the points.

In order to draw a regression line we need to know two things:

(1) We need to know where the line begins - the value ofY(our dependent variable) when X (our independent variable) is 0 - so that we have a point from which to start drawing the line. The technical name for this point is the intercept or the constant.

(2) And we need to know what is the slope of that line.

If you recall from school algebra (and you may not), the equation for any straight line is: y = mx + b. In statistics we use a slightly different notation, although the equation remains the same: y = b0 + b1x.

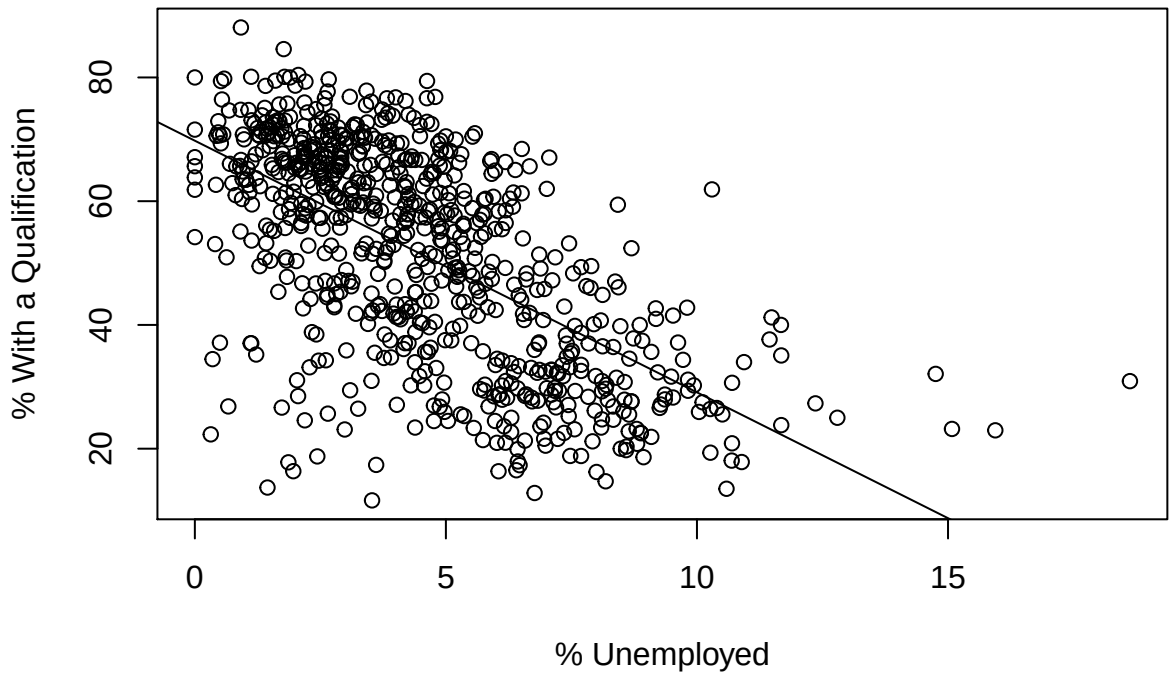
We need the origin of the line (b0) and the slope of the line (b1). How does R get the intercept and the slope for the regression line? How does R know where to draw this line? We need to estimate these parameters (or coefficients) from the data. For linear regression models (like the one we cover here) R tries to minimise the distance from every point in the scatterplot to the regression line using a method called least squares estimation.

In order to fit the model we use the `lm()` function using the formula specification (Y ~ X). Typically you want to store your regression model in a "variable", let's call it model_1:

```
model_1 <- lm(Census.Data$Qualification~ Census.Data$Unemployed)
```

First lets add the regression line from the model to a scatter plot by using the `abline()` function (as we did in the previous practical. Notice that the model orders the x and y the other way round.

```
plot(Census.Data$Unemployed, Census.Data$Qualification, xlab="% Unemployed",
     ylab="% With a Qualification") + abline (model_1)
```




```
## numeric(0)
```

If you want to simply see the basic results from running the model you can use the `summary()` function.

```
summary(model_1)
```

```
##
## Call:
## lm(formula = Census.Data$Qualification ~ Census.Data$Unemployed)
##
```

```
## Residuals:
##     Min      1Q  Median      3Q     Max
## -50.172  -9.635   2.339   9.512  36.887
##
## Coefficients:
##                         Estimate Std. Error t value Pr(>|t|)
## (Intercept)              69.7740     0.9743   71.61   <2e-16 ***
## Census.Data$Unemployed  -4.0672     0.1861  -21.85   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 13.53 on 747 degrees of freedom
## Multiple R-squared:  0.3899, Adjusted R-squared:  0.3891
## F-statistic: 477.4 on 1 and 747 DF,  p-value: < 2.2e-16
```

For now just focus on the numbers in the "Estimate" column. The value of 69.78 estimated for the intercept is the "predicted" value for Y when X equals zero - it is possible to interpret this from observing the scatter plot we just made. This is the predicted value of percentage of people with degrees when the percentage of people who are unemployed is zero.

We then need the b1 regression coefficient for our independent variable (Unemployed), the value that will shape the slope in this scenario. This value is -4.0672. This estimated regression coefficient for our independent variable has a convenient interpretation. When the value is positive, it tells us that for every one unit increase in X there is a b1 increase on Y. If the coefficient is negative then it represents a decrease on Y. Here, we can read it as "for every one unit increase in the percentage of people who are unemployed, there is a -4.0672 unit decrease in the percentage of people with a degree."

Knowing these two parameters not only allows us to draw the line, we can also solve Y for any given value of X. If the percentage of people who are unemployed is 15% in a given area, we can simply go back to our regression line equation and insert the estimated parameters:

y = b0 + b1x or y = 69.78 + (-4.0672 x 15)

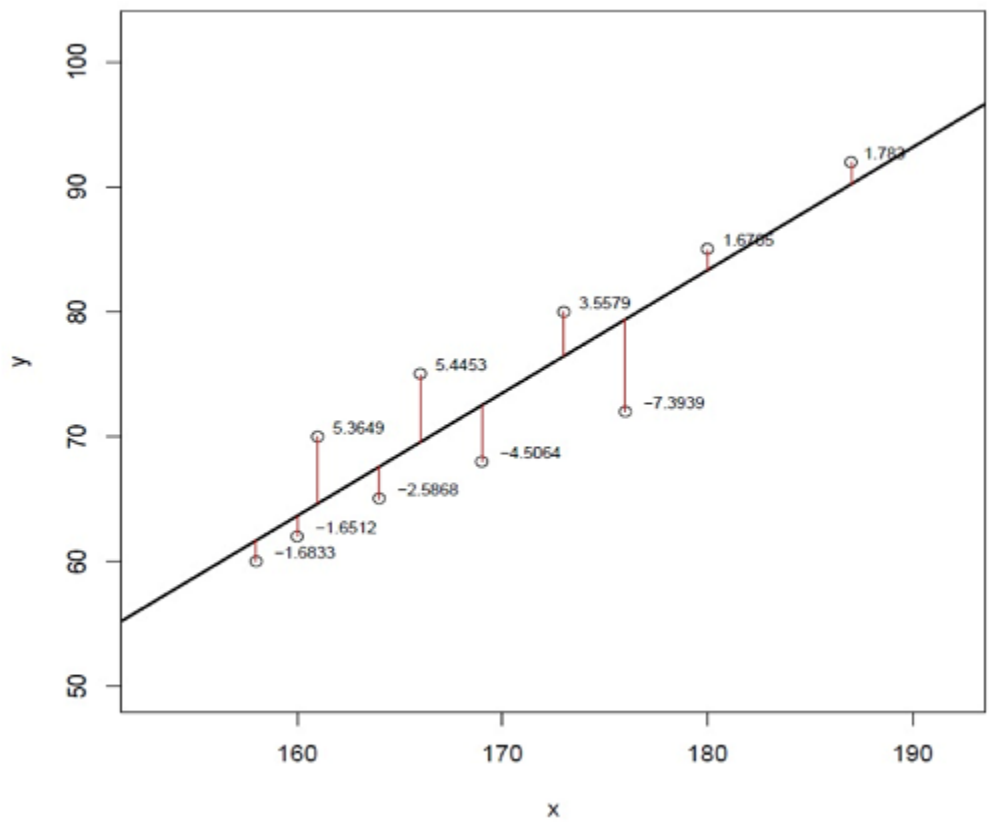
If you don't want to do the calculation yourself, you can use the predict function:

```
predict(model_1, data.frame(Unemployed = c(15)))
```

Of course this model is simplification of reality and only considers the influence of one variable.

R squared

In the output above we saw there was something called the residuals. The residuals are the differences between the observed values of Y for each case minus the predicted or expected value of Y, in other words the distances between each point in the dataset and the regression line (see the visual example below).

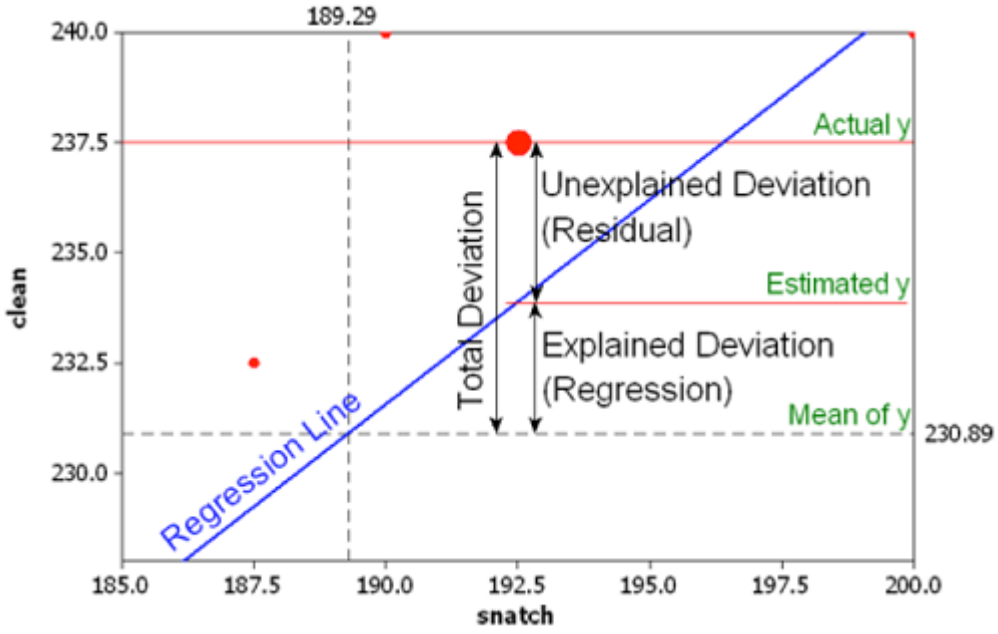


We see here points indicating the observed values (the data points in our sample), red lines indicating their distance to the regression lines (the residuals), and numerical values indicating the distance from each of the points to the regression line. Least square estimation essentially aims to reduce the squared average of all these distances: that's how it draws the line.

We have residuals because our line is not a perfect representation of the cloud of points. You cannot predict perfectly what the value of Y is for every area just by looking ONLY at the value of X. There are other things that may influence the values of Y which are not being taken into account by our model. There are other things that surely matter in terms of understanding of the relationship between health and levels of education. And then, of course, we have measurement error and other forms of noise.

We can re-write our equation like this if we want to represent each value of Y (rather than the predicted value of Y) then: $y = b0 + b1x + error$

The residuals capture how much variation is unexplained, how much we still have to learn if we want to understand variation in Y. A good model tries to maximise explained variation and reduce the magnitude of the residuals. We can use information from the residuals to produce a measure of effect size, of how good our model is in predicting variation in our dependent variables. If we did not have any information about X our best bet for Y would be the mean of Y. The regression line aims to improve that prediction. By knowing the values of X we can build a regression line that aims to get us closer to the actual values of Y (look at the Figure below).



The distance between the mean (our best guess without any other piece of information) and the observed value of Y is what we call the total variation. The residual is the difference between our predicted value of Y and the observed value of Y. This is what we cannot explain (i.e, variation in Y that is unexplained). The difference between the mean value of Y and the expected value of Y (the value given by our regression line) is how much better we are doing with our prediction by using information about X. How much closer the regression line gets us to the observed values. We can then contrast these two different sources of variation (explained and unexplained) to produce a single measure of how good our model is. The formula is as follows:

$$R^2 = \frac{SSR}{SST} = \frac{\sum(\hat{y}_i - \bar{y})^2}{\sum(y_i - \bar{y})^2}$$

All this formula is doing is taking a ratio of the explained variation (the squared differences between the regression line and the mean of Y for each observation) by the total variation (the squared differences of the observed values of Y for each observation from the mean of Y). This gives us a measure of the percentage of variation in Y that is "explained" by X.

We can take this value as a measure of the strength of our model. If you look at the R output you will see that the R2 for our model was 0.3899 (look at the multiple R2 value in the output). We can say that our model explains about 40% of the variance in the percentage of people with a degree in our study area.

Knowing how to interpret this is important. R2 ranges from 0 to 1. The greater it is, the more powerful our model is, the more explaining we are doing, and the better we are able to account for variation in our outcome Y with our input. In other words, the stronger the relationship is between Y and X. As with all the other measures of effect size, interpretation is a matter of judgement. You are advised to see what other researchers report in relation to the particular outcome that you may be exploring. We can use the R2 to compare against other models we might fit to see which is most powerful.

Inference with regression

In real world applications, we have access to a set of observations from which we can compute the least squares line, but the population regression line is unobserved. So our regression line is one of many that could

be estimated. A different set of Output Areas would produce a different regression line. If we estimate b0 and b1 from a particular sample, then our estimates won't be exactly equal to b0 and b1 in the population. But if we could average the estimates obtained over a very large number of data sets, the average of these estimates would equal the coefficients of the regression line in the population.

In the same way that we can compute the standard error when estimating the mean, we can compute standard errors for the regression coefficients to quantify our uncertainty about these estimates. These standard errors can in turn be used to produce confidence intervals. This would require us to assume that the residuals are normally distributed. For a simple regression model, you are assuming that the values of Y are approximately normally distributed for each level of X:

You can also then perform standard hypothesis test on the coefficients. As we saw before when summarising the model, R will compute the standard errors and a t-test for each of the coefficients.

In our example, we can see that the coefficient for our predictor here is statistically significant, as represented by the p-value. Notice that the t-statistics and p-value are the same as the correlation coefficient.

```
summary(model_1)
```

```
##
## Call:
## lm(formula = Census.Data$Qualification ~ Census.Data$Unemployed)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -50.172  -9.635   2.339   9.512  36.887
##
## Coefficients:
##                          Estimate Std. Error t value Pr(>|t|)
## (Intercept)               69.7740     0.9743   71.61   <2e-16 ***
## Census.Data$Unemployed    -4.0672     0.1861  -21.85   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 13.53 on 747 degrees of freedom
## Multiple R-squared:  0.3899, Adjusted R-squared:  0.3891
## F-statistic: 477.4 on 1 and 747 DF,  p-value: < 2.2e-16
```

We can also obtain confidence intervals for the estimated coefficients using the confint() function. The below example will produce a 95% confidence interval for the model. The 95% confidence interval defines a range of values that you can be 95% certain contains the mean slope of the regression line.

```
confint(model_1, level= 0.95)
```

```
##                             2.5 %    97.5 %
## (Intercept)            67.861262 71.686689
## Census.Data$Unemployed -4.432593 -3.701747
```

Mutliple regression

So we have seen our models with just one predictor or explanatory variable. We can build 'better' models by increasing the number of predictors. In our case we can also add another variable into the model for predicting the number of people with degree level qualifications. We have seen from the plots above that there are clearly fewer people living in deprivation in areas where more people have a degree, so let's see if it helps us make better predictions.

Another reason why it is important to think about additional variables in your model is to control for spurious

correlations (although here you may also want to use your common sense when selecting your variables!). You have heard that correlation does not equal causation. Just because two things are associated we cannot assume that one is the cause for the other. Typically we see how the pilots switch the "secure the belt" sign on when there is turbulence during a flight. These two things are associated, they tend to come together. But the pilots are not causing the turbulence by pressing a switch! The world is full of spurious correlations, associations between two variables that should not be taken too seriously.

It's not an exaggeration to say that most quantitative explanatory research is about trying to control for the presence of confounders - variables that may explain away observed associations. Think about any social science question: Are married people less prone to depression? Or is it that people that get married are different from those that don't (and are there pre-existing differences that are associated with less depression)? Are ethnic minorities more likely to vote for centre-left political parties? Or, is it that there are other factors (e.g. socioeconomic status, area of residence, sector of employment) that, once controlled, would mean there is no ethnic group difference in voting?

Multiple regression is one way of checking the relevance of competing explanations. You could, for example, set up a model where you try to predict voting behaviour with an indicator of ethnicity and an indicator of structural disadvantage. If, after controlling for structural disadvantage, you see that the regression coefficient for ethnicity is still significant you may be onto something, particularly if the estimated effect is still large. If, on the other hand, the t-test for the regression coefficient of your ethnicity variable is no longer significant, then you may be tempted to think structural disadvantage is a confounder for vote selection.

It could not be any easier to fit a multiple regression model. You simply modify the formula in the `lm()` function by adding terms for the additional inputs. Here the 2nd predictor (or independent) variable is the % of the white British population.

```
# runs a model with two independent variables
model_2 <- lm(Census.Data$Qualification~ Census.Data$Unemployed +
                Census.Data$White_British)

# view the model summary
summary(model_2)
```

```
##
## Call:
## lm(formula = Census.Data$Qualification ~ Census.Data$Unemployed +
##     Census.Data$White_British)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -50.311  -8.014   1.006   8.958  38.046
##
## Coefficients:
##                            Estimate Std. Error t value Pr(>|t|)
## (Intercept)                47.86697    2.33574   20.49   <2e-16 ***
## Census.Data$Unemployed     -3.29459    0.19027  -17.32   <2e-16 ***
## Census.Data$White_British   0.41092    0.04032   10.19   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 12.69 on 746 degrees of freedom
## Multiple R-squared:  0.4645, Adjusted R-squared:  0.463
## F-statistic: 323.5 on 2 and 746 DF,  p-value: < 2.2e-16
```

Now we can consider the influence of multiple vaiables. Also, you will notice that the R2 vaue has improved slightly compared to the first model. Try testing various different combinations of variables. Which model is most efficient at representing the distribution of our qualifications variable across the study area?

Practical 5: Making maps in R

This practical provides a good introduction of how to handle and map spatial polygon data in R. While there are several means of doing this in R, through a large number of bespoke packages, we will focus on the functionality of the tmap package due to its relative simplicity. Data for the practical can be downloaded from the **Introduction to Spatial Data Analysis and Visualisation in R** homepage.

In this tutorial we will:

- Load spatial data files into R
- Join data to GIS spatial data files
- Create a simple choropleth map
- Customise choropleth maps with the tmap package

First we must set the working directory and load the practical data.

```
# Set the working directory. Remember to change the example below.
setwd("C:/Users/Guy/Documents/Teaching/CDRC/Practicals")

# Load the data. You may need to alter the file directory
Census.Data <-read.csv("practical_data.csv")
```

Loading shapefiles into R

A GIS shapefile is a file format for storing the location, shape, and attributes of geographic features. In other words, it contains the geographic information which allow it to be mapped as either points, lines or polygons.

First we need to load some packages. Remember to install them first if you have not used them before. To install go to **Tools > Install packages.** in RStudio and enter the name of the package or use the `install.packages()` function.

The packages are:

- `rgdal` - Bindings for the Geospatial Data Abstraction Library
- `rgeos` - Interface to Geometry Engine - Open Source

```
# Load packages
library("rgdal")
library("rgeos")
```

Next, we need to load the output area shapefile into R.

Shapefiles are made up of multile different files which, when combined by certain software packages, can be mapped using a common projection system. We will be using output area boundaries as our data is at that level. Therefore, the data will be visualised as a polygon file whereby each individual polygon represents the outline of a unique output area from our study area. In this example, our spatial data files can be found in the shapefiles folder of our data pack. Together they comprise:

- Camden_oa11.dbf
- Camden_oa11.prj
- Camden_oa11.shp
- Camden_oa11.shx

Before doing this, please find the *Camden_oa11* files from within the shapefiles folder of your census data pack you which you downloaded. Move these files to your working directory.

```
# Load the output area shapefiles
Output.Areas<- readOGR(".", "Camden_oa11")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "Camden_oa11"
## with 749 features
## It has 1 fields
```

The inputted shapefile should contain an identical number of features as the number of observations in our *Census.Data* file.

We can now explore the shapefile. First, we will plot it as a map to view the spatial dimensions of the shapefile mapped out.

This is very simple, just enter the object name of the shapefile into the standard `plot()` function.

```
# plots the shapefile
plot(Output.Areas)
```

Joining data

We now need to join our Census.Data to the shapefile so the census attributes can be mapped. As our census data contains the unique names of each of the output areas, this can be used a key to merge the data to our output area file (which also contains unique names of each output area). We will use the `merge()` function to join the data.

Notice that this time the column headers for our output area names are not identical, despite the fact they contain the same data. We therefore have to use `by.x` and `by.ya` so the merge function uses the correct columns to join the data.

```
# joins data to the shapefile
OA.Census <- merge(Output.Areas, Census.Data, by.x="OA11CD", by.y="OA")
```

Setting a coordinate system

It is also important to set a coordinate system, especially if you want to map multiple different files. The `proj4string()` and `CRS()` functions allows us to set the coordinate system of a shapefile to a predefined system of our choice. Most data from the UK is projected using the British National Grid (EPSG:27700) produced by the Ordnance Survey, this includes the standard statistical geography produced by the Office for National Statistics.

In this case, the shapefile we originally downloaded from the CDRC Data website already has the correct projection system so we don't need to run this step for our OA.Census object. However, it is worth taking note of this step for future reference.

```
# sets the coordinate system to the British National Grid
proj4string(OA.Census) <- CRS("+init=EPSG:27700")
```

Mapping data in R

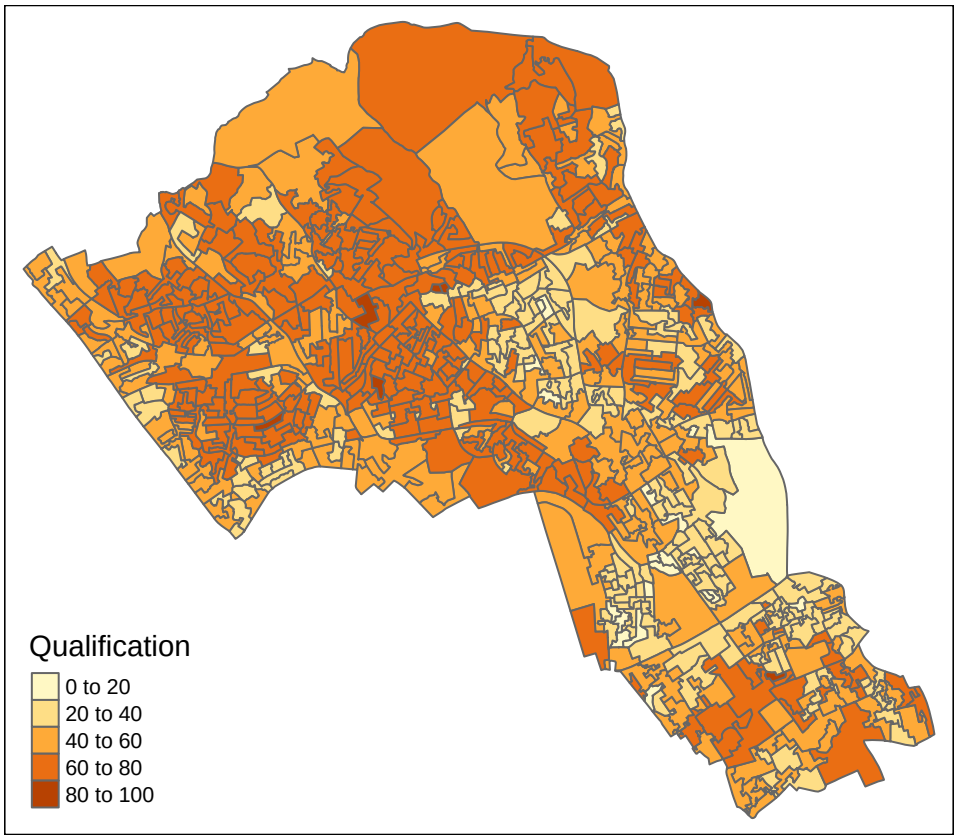
Whilst the plot function is pretty limited in its basic form. Several packages allow us to map data relatively easily. They also provide a number of functions to allow us to tailor and alter the graphic. `ggplot2`, for example, allows you to map spatial data. However, probably the easiest to use are the functions within the `tmap` library.

```
# loads packages
library(tmap)
library(leaflet)
```

Creating a quick map

If you just want to create a map with a legend quickly you can use the `qtm()` function.

```
# this will prodyce a quick map of our qualification variable
qtm(OA.Census, fill = "Qualification")
```



Creating more advanced maps with tmap

Creating maps in tmap involves you binding together several functions that comprise different aspects of the graphic. For instance:

```
polygon + polygon's symbology +
borders +
layout
```

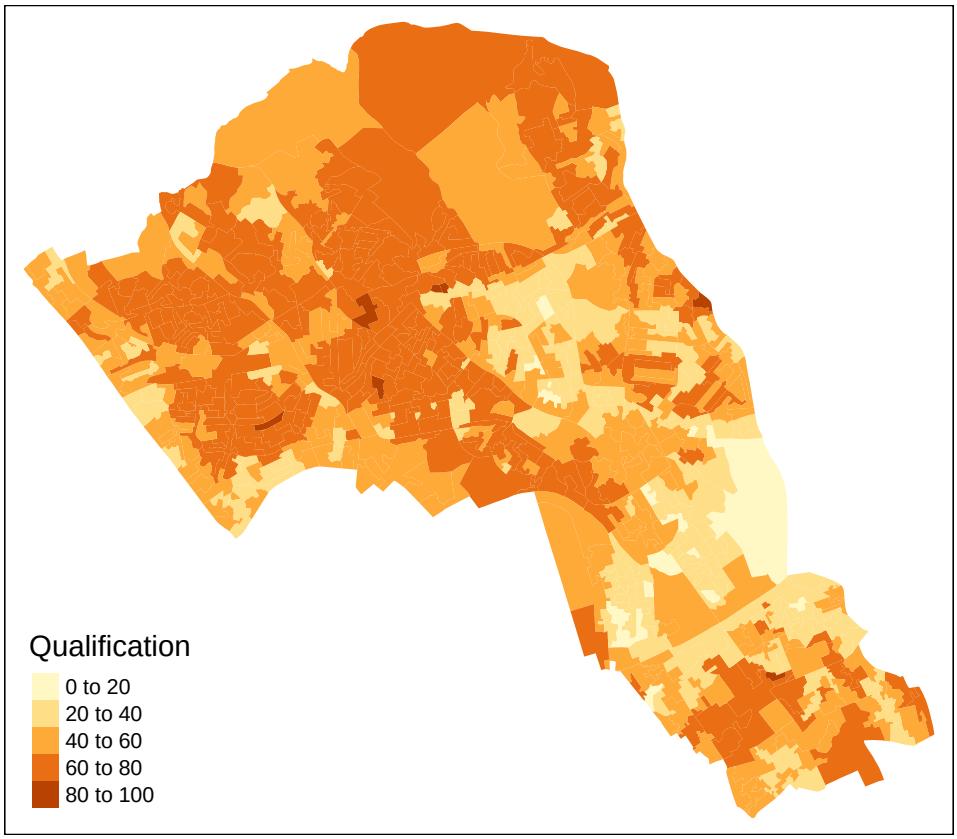
We enter shapefiles (or R spatial data objects) followed by a command to set their symbologies. The objects are then layered in the visualisation in order. The object entered first appears at the bottom of the graphic.

Creating a simple map

Here we load in the shapefile with the `tm_shape()` function then add in the `tm_fill()` function which is where we can decide how the polygons are filled in. All we need to do in the `tm_shape()` function is call the shapefile. We then add (`+`) a separate new function (`tm_fill()`) to enter the parameters which will determine how the polygons are filled in the graphic. By default, we only need to include a variable name within the parameters.

You can explore all of the available functions and how they can be customised by visiting the webpage for tmap, or by entering `?` followed by a function name into r (i.e. `?tm_fill`). Below we will go through some of the basic steps of mapping with tmap.

```
# Creates a simple choropleth map of our qualification variable
tm_shape(OA.Census) + tm_fill("Qualification")
```



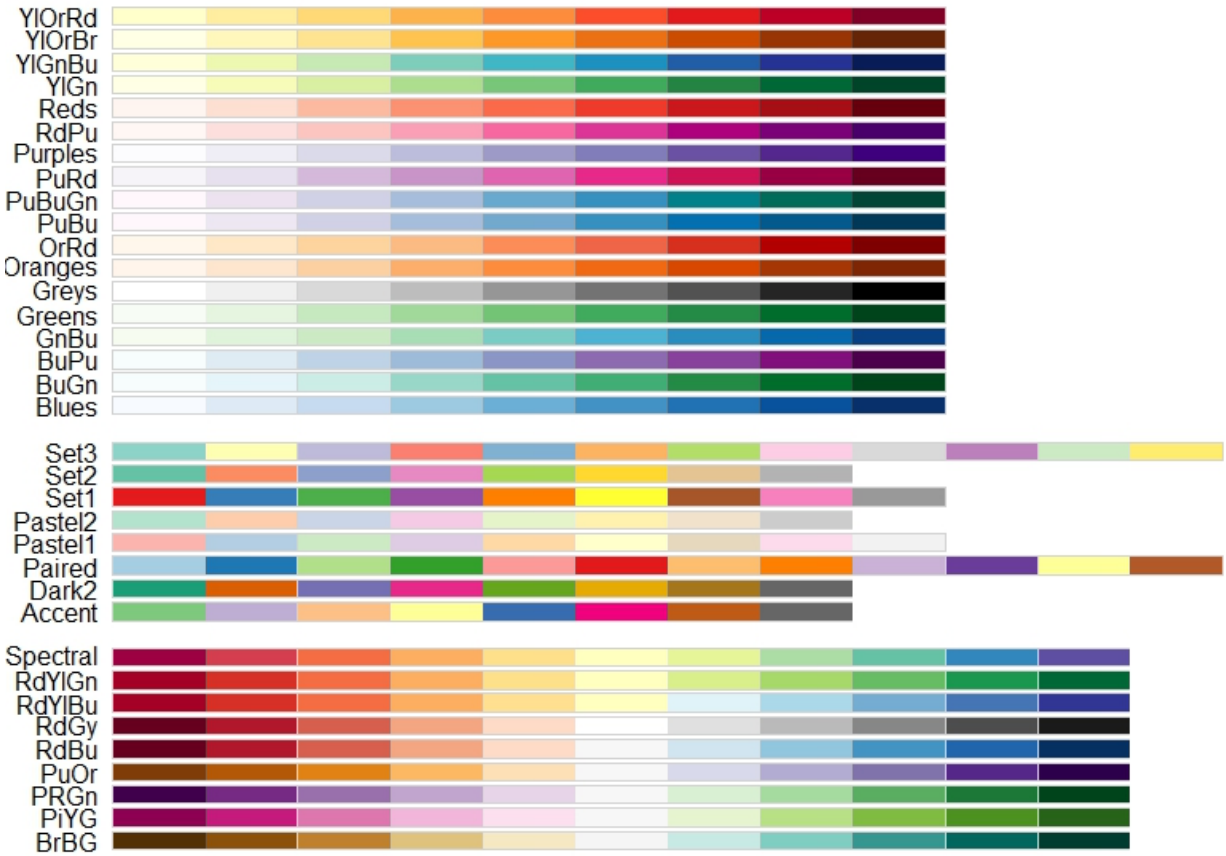
You will notice that the map is very similar to the quick map produced from using the `qtm()` function. However, the advantage of using the advanced functions from tmap is that they provide a wide range of customisation options. The following steps will demonstrate this.

Setting the colour palette

tmap allows you to use colour ramps either defined by the user or a set of predefined colour ramps from the `RColorBrewer()` function.

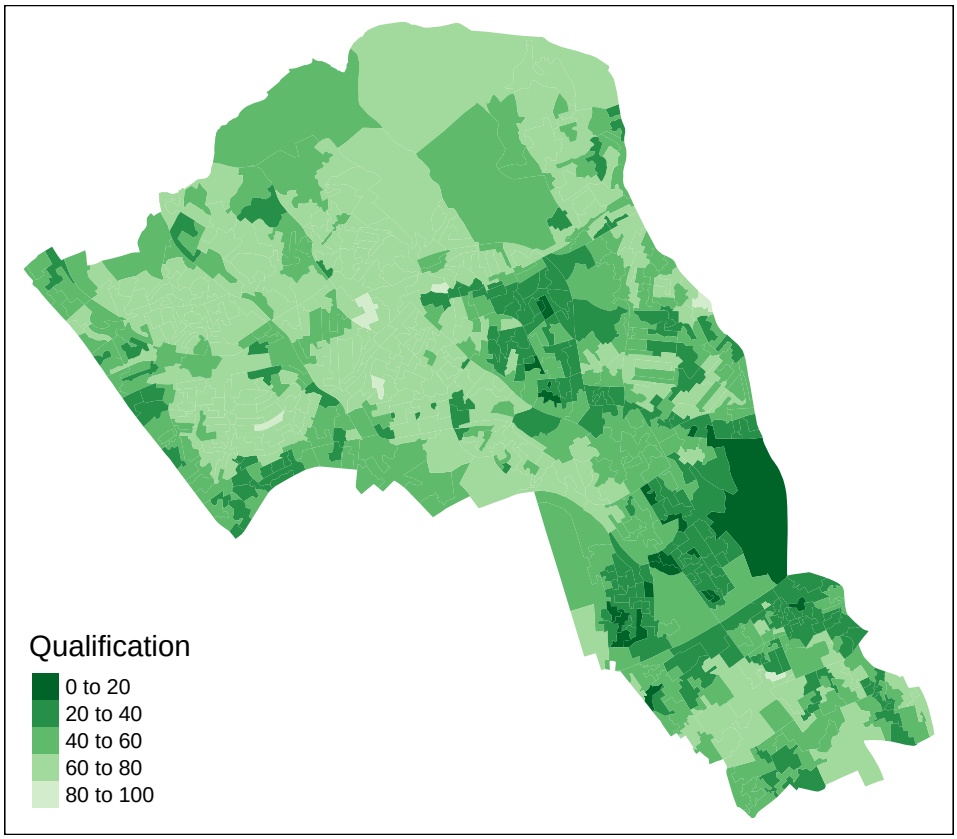
To explore the predefined colour ramps in ColourBrewer enter the following code

```
library(RColorBrewer)

display.brewer.all()
```



This presents a range of previously defined colour ramps (as shown above). The continuous ramps at the top are all appropriate for our data. If you enter a minus sign before the name of the ramp within the brackets (i.e. `-Greens`), you will invert the order of the colour ramp.

```
# setting a colour palette
tm_shape(OA.Census) + tm_fill("Qualification", palette = "-Greens")
```



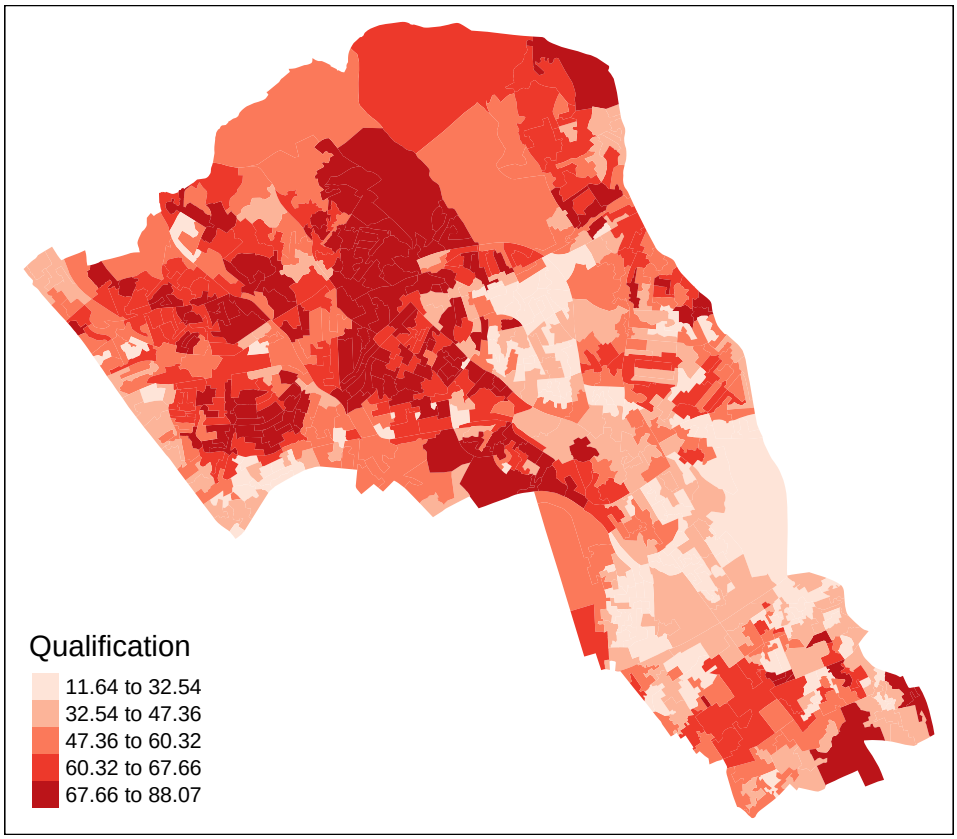
Setting the colour intervals

We have a range of different interval options in the style parameter. Each of them will greatly impact how your data is visualised. To do this you enter "style =" followed by one of the options below.

- **equal** - divides the range of the variable into n parts.
- **pretty** - chooses a number of breaks to fit a sequence of equally spaced 'round' values. So they keys for these intervals are always tidy and memorable.
- **quantile** - equal number of cases in each group
- **jenks** - looks for natural breaks in the data
- **Cat** - if the variable is categorical (i.e. not continuous data)

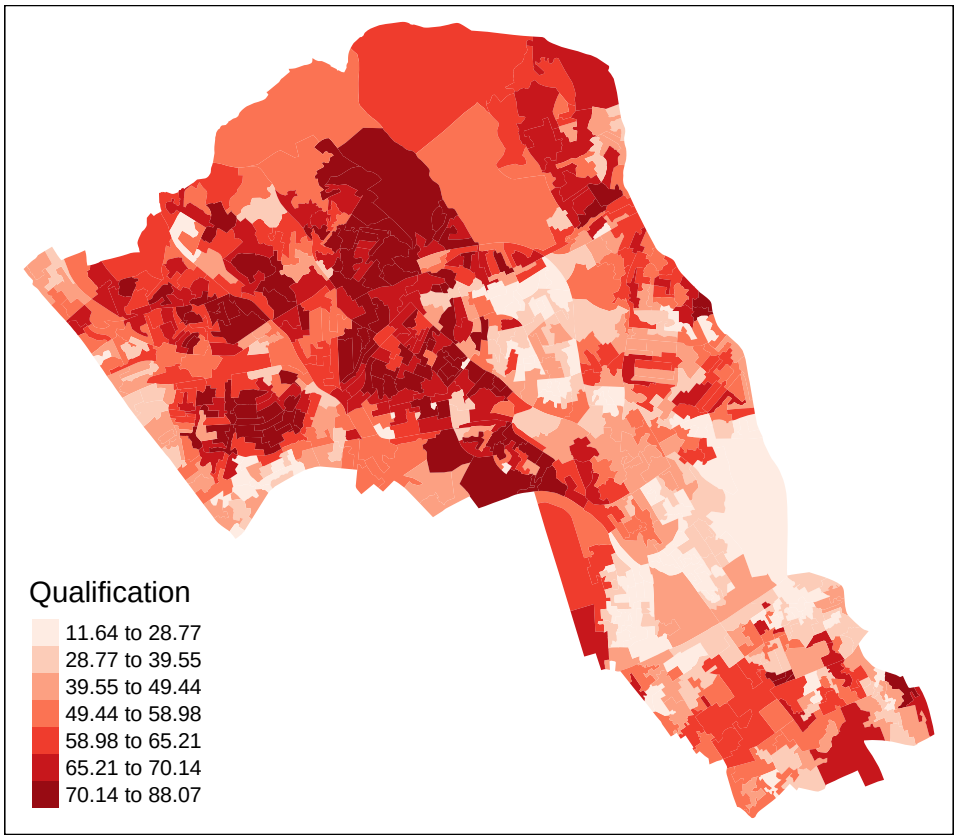
Try a couple of different interval styles to observe how they visualise the data differently. The example below uses the quantile interval scheme

```
# changing the intervals
tm_shape(OA.Census) + tm_fill("Qualification", style = "quantile", palette = "Reds")
```



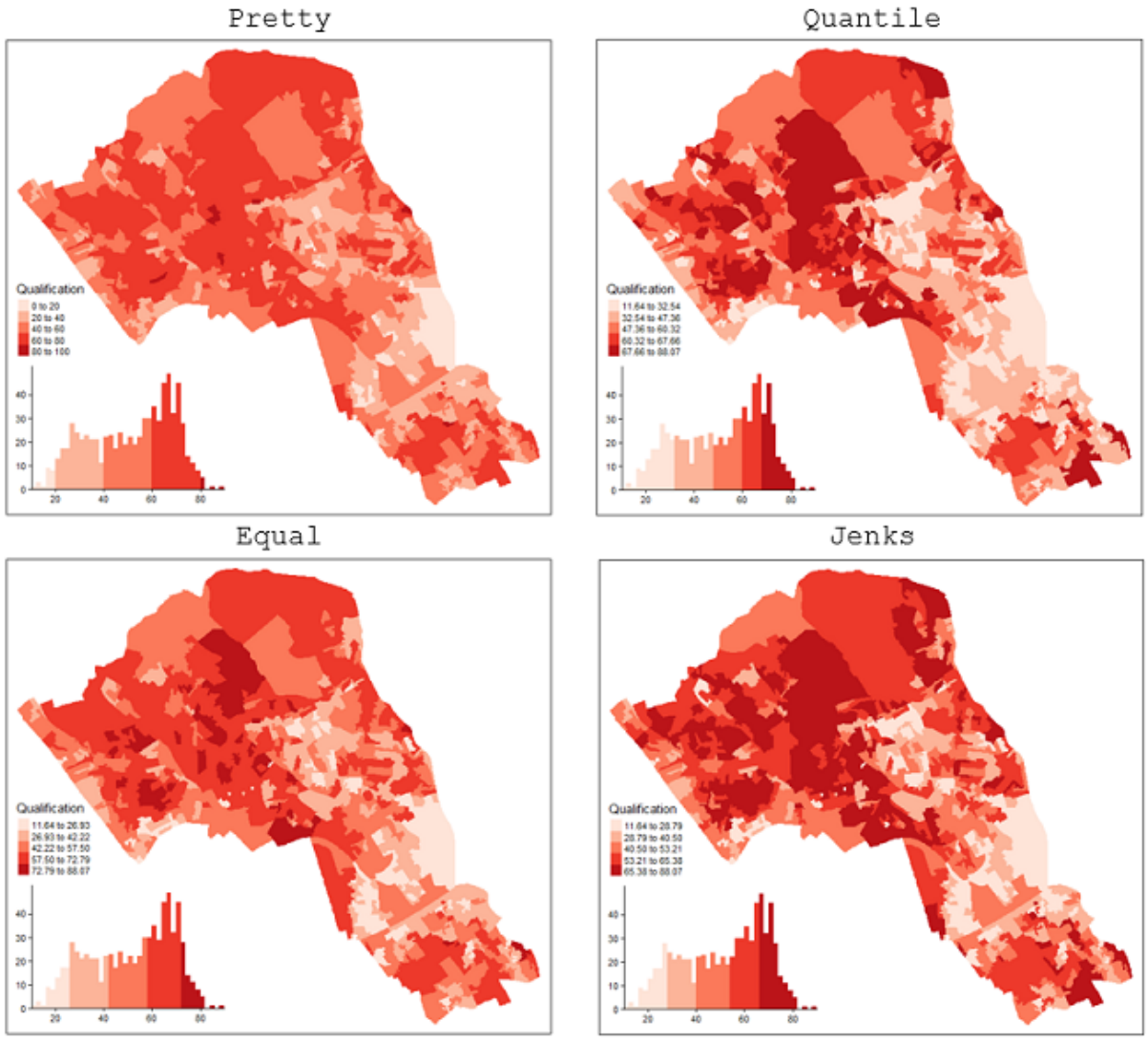
We can also change the number of intervals in the colour scheme and how the intervals are spaced. Changing the number of intervals is straight-forward, just n= n (note that the default interval setting may still round the number). Here we have 7 shades instead of the default 5.

```
# number of levels
tm_shape(OA.Census) + tm_fill("Qualification", style = "quantile", n = 7,
                              palette = "Reds")
```



You can also create a histogram within the legend, simply add `legend.hist = TRUE` within tm_fill. The histogram can be quite informative of how the intervals are defined. Try running the maps with a histogram to observe how different intervals are univariately distributed across our data.

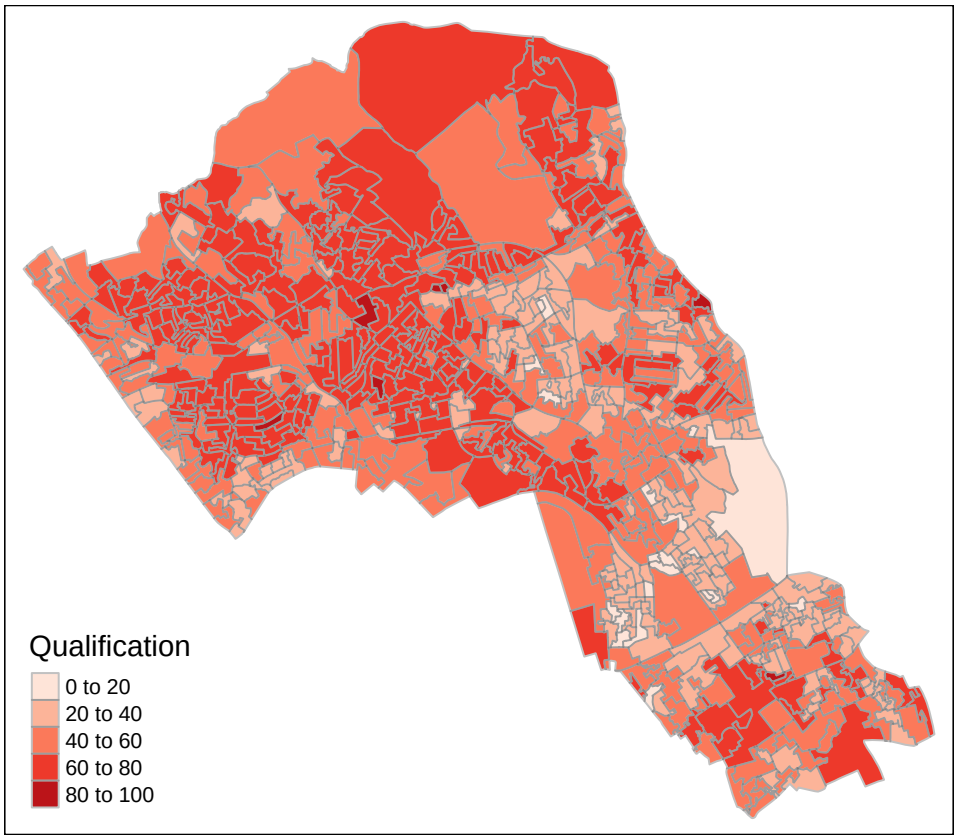
```
# includes a histogram in the legend
tm_shape(OA.Census) + tm_fill("Qualification", style = "quantile", n = 5,
                              palette = "Reds", legend.hist = TRUE)
```



Adding borders

You can edit the borders of the shapefile with the `tm_borders()` function which has many arguments. `alpha` denotes the level of transparency on a scale from 0 to 1 where 0 is completely transparent.

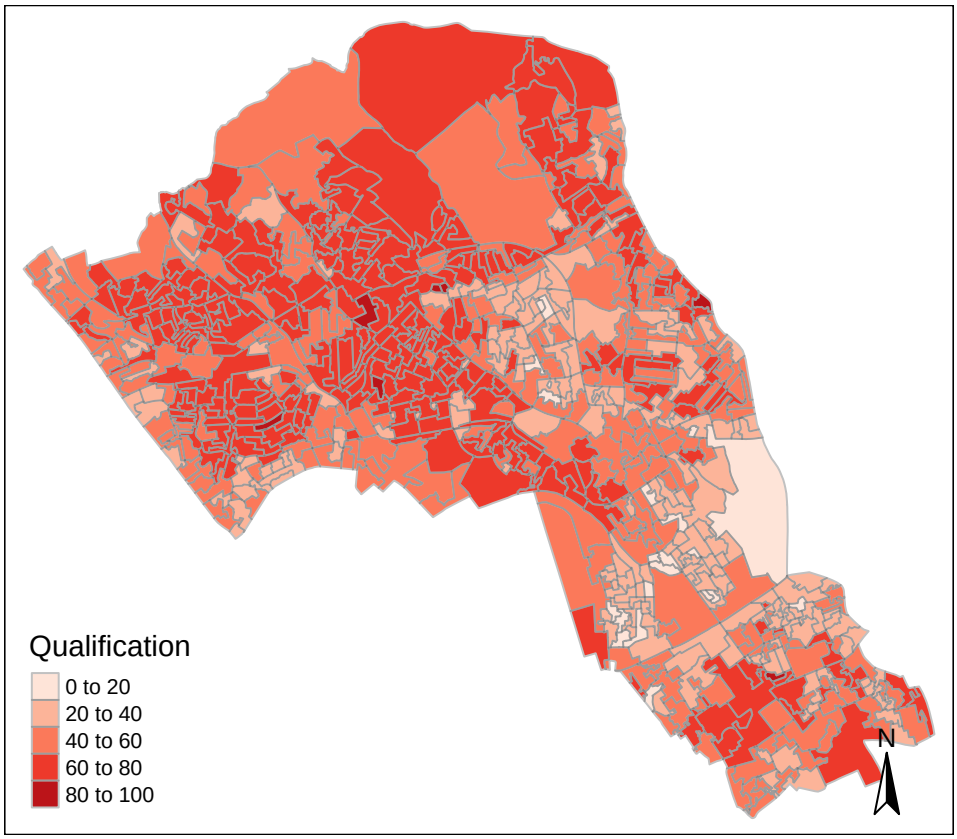
```
# add borders
tm_shape(OA.Census) + tm_fill("Qualification", palette = "Reds") +
tm_borders(alpha=.4)
```



Adding a north arrow

We can also enter a north arrow with `tm_compass()`.

```
# north arrow
tm_shape(OA.Census) + tm_fill("Qualification", palette = "Reds") +
tm_borders(alpha=.4) +
tm_compass()
```

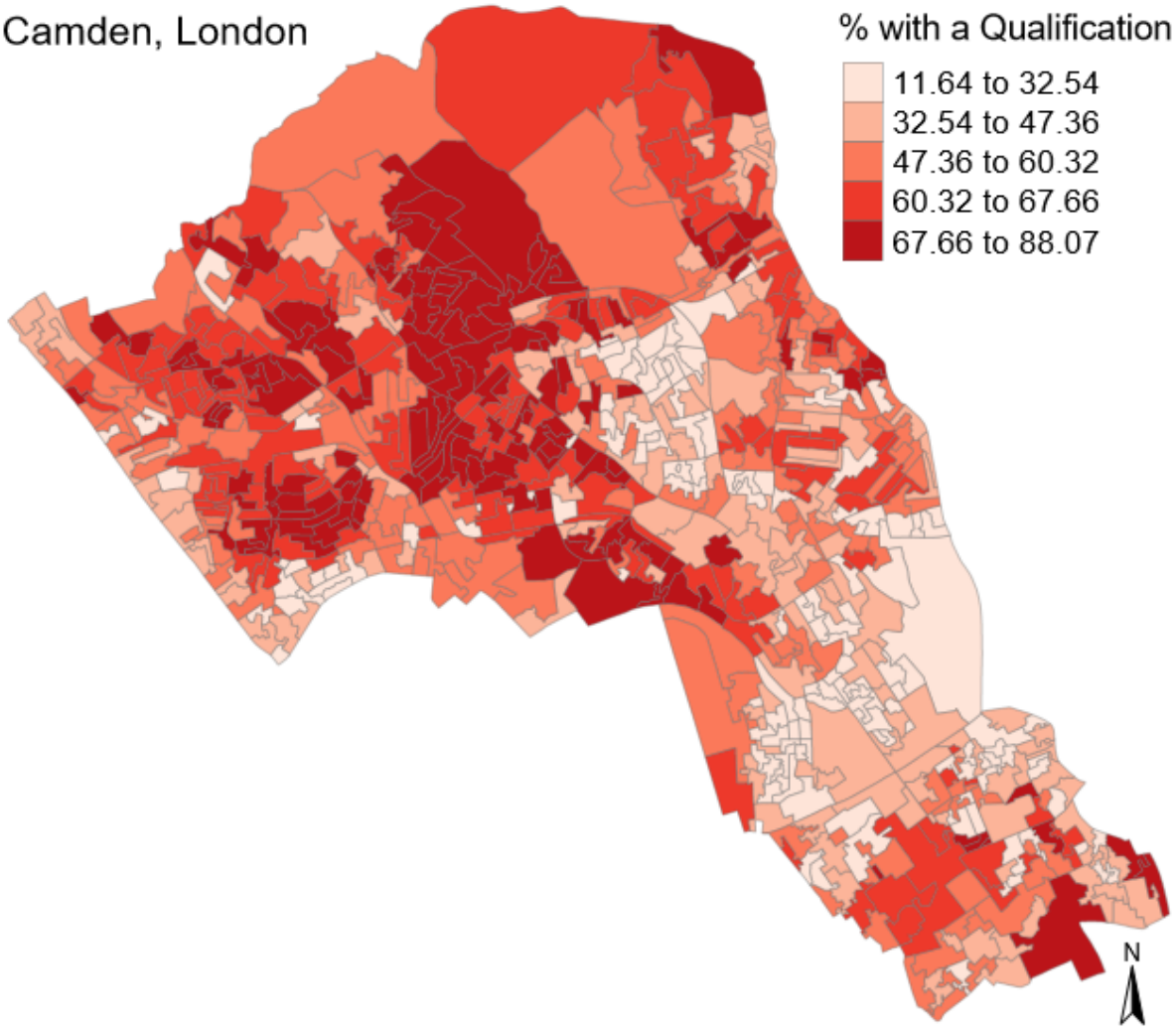


Editing the layout of the map

It is possible to edit the layout using the `tm_layout()` function. In the example below we have also added in a few more commands within the `tm_fill()` function.

```
# adds in layout, gets rid of frame
tm_shape(OA.Census) + tm_fill("Qualification", palette = "Reds",
                              style = "quantile", title = "% with a Qualification") +
tm_borders(alpha=.4) +
tm_compass() +
tm_layout(title = "Camden, London", legend.text.size = 1.1,
          legend.title.size = 1.4, legend.position = c("right", "top"), frame = FALSE)
```

This is an exported copy of the map produced...



Saving the shapefile

Finally, we can save the shapefile with the Census data already attatched to by simply running the following code (remember to change the dsn to your working directory).

```
writeOGR(OA.Census, dsn = "C:/Users/Guy/Documents/Teaching/CDRC/Practicals",
         layer =  "Census_OA_Shapefile", driver="ESRI Shapefile")
```

Practical 6: Mapping Point Data in R

This practical will follow on from the previous exercise by introducing the handling and mapping of spatial point data in R using tmap. Data for the practical can be downloaded from the **Introduction to Spatial Data Analysis and Visualisation in R** homepage.

In this tutorial we will:

- Create a point shapefile from a CSV using coordinates
- Map the points in tmap
- Create a proportional bubble map

First, we must set the working directory and load the practical data.

```
# Set the working directory
setwd("C:/Users/Guy/Documents/Teaching/CDRC/Practicals")

# Load the data. You may need to alter the file directory
Census.Data <-read.csv("practical_data.csv")
```

We will also need the polygon shapefile from our previous exercise and to join our census data to it.

```
# load the spatial libraries
library("rgdal")
library("rgeos")

# Load the output area shapefiles
Output.Areas <- readOGR(".", "Camden_oa11")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "Camden_oa11"
## with 749 features
## It has 1 fields
```

```
# join our census data to the shapefile
OA.Census <- merge(Output.Areas, Census.Data, by.x="OA11CD", by.y="OA")
```

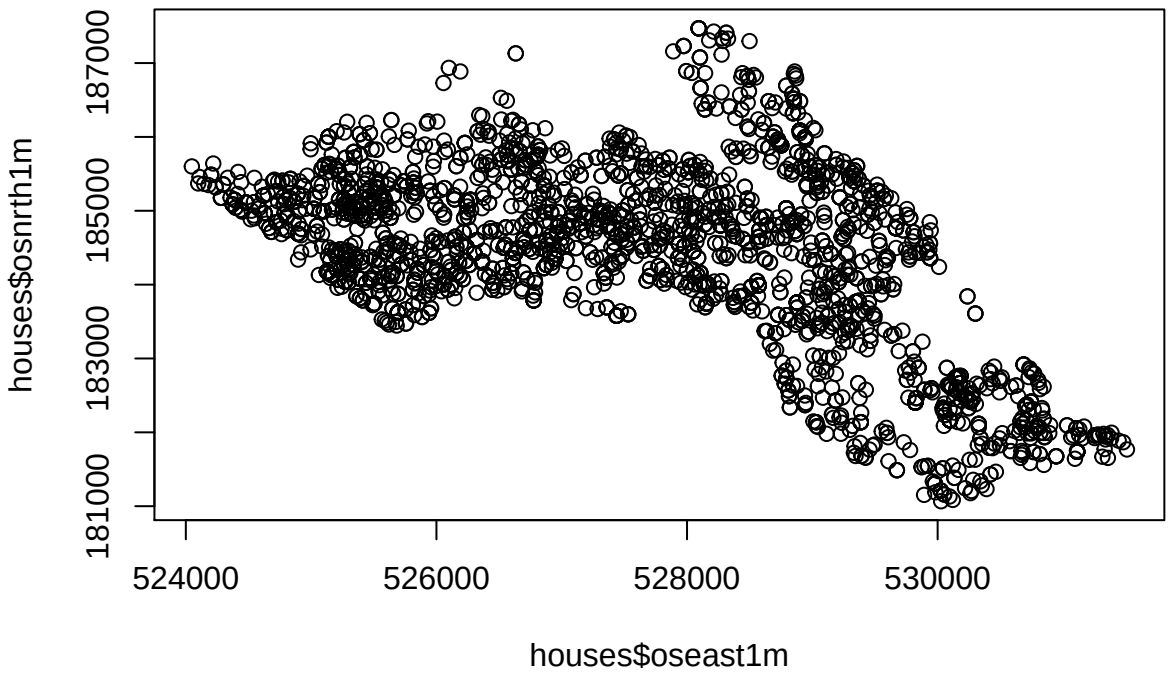
Loading point data into R

In this practical we will be handling house price paid data originally made available for free by the Land Registry. The sample dataset can be downloaded from the CDRC website here. The data is formatted as CSV where each row is a unique house sale, including the price paid in pounds and the postcode. Prior to this practical, the data file was joined to the Office for National Statistics (ONS) postcode lookup table which provides latitude and longitude coordinates for each postcode.

```
# load the house prices csv file
houses <- read.csv("CamdenHouseSales15.csv")

# we only need a few columns for this practical

houses <- houses[,c(1,2,8,9)]
```

Whilst it is possible to plot this data using the standard `plot()` in R (as demonstrated below), it is not being handled as spatial data.

```
# 2D scatter plot
plot(houses$oseast1m, houses$osnrth1m)
```



Therefore, we need to assign spatial attributes to the CSV so it can be mapped properly in R. To do this we will need to load the sp package, this package provides classes and methods for handling spatial data. Remember to install the package first if you have not done so before.

Next we will convert the CSV into a SpatialPointsDataFrame. To do this we will need to set what the data is to be included, what columns contain the x and y coordinates, and what projection system we are using.

```
library("sp")

# create a House.Points SpatialPointsDataFrame
House.Points <-SpatialPointsDataFrame(houses[,3:4], houses,
                                      proj4string = CRS("+init=EPSG:27700"))
```

Mapping point data

Before we map the points, we will create a base map using the output area boundaries.

```
library("tmap")

# This plots a blank base map, we have set the transparency of the borders to 0.4
tm_shape(OA.Census) + tm_borders(alpha=.4)
```

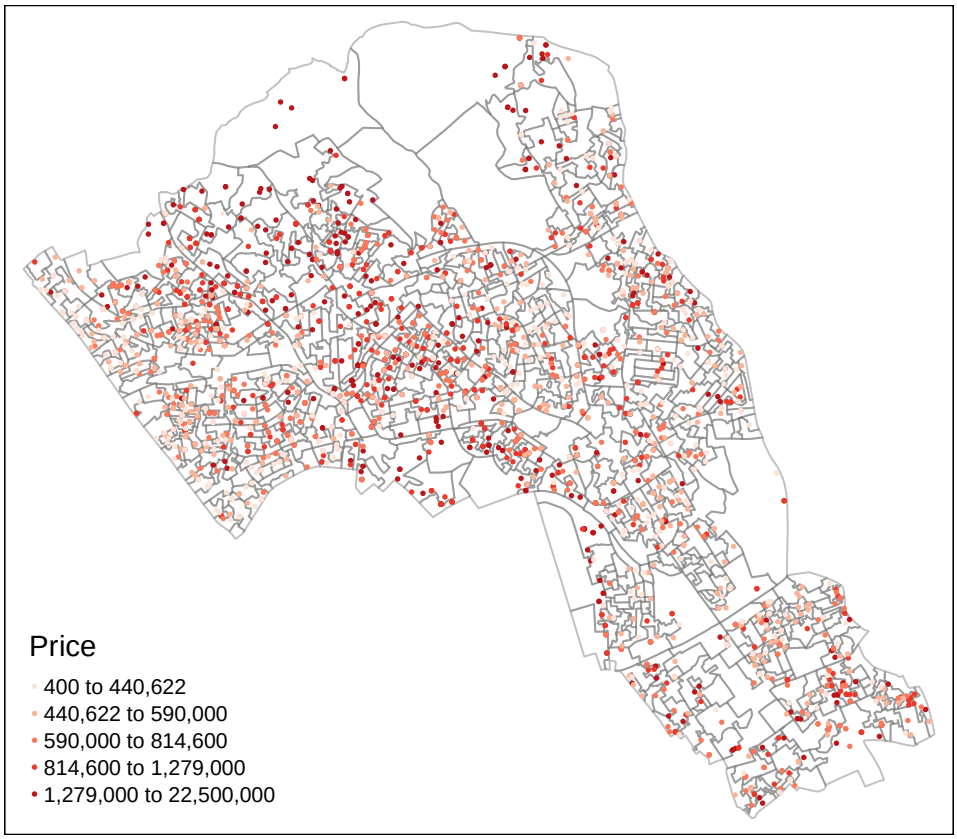
We can now add on the points as an additional *tm_shape* layer in our map.

To do this, we copy in the same code to make the base map, add on a plus symbol, then enter the details for the points data. The additional arguments for the points data can be summarised as:

```
tm_shape(polygon file) + tm_borders(transparency = 40%) +
tm_shape(our spatial points data frame) + tm_dots(what variable is coloured,
                                                  the colour palette and interval style)
```

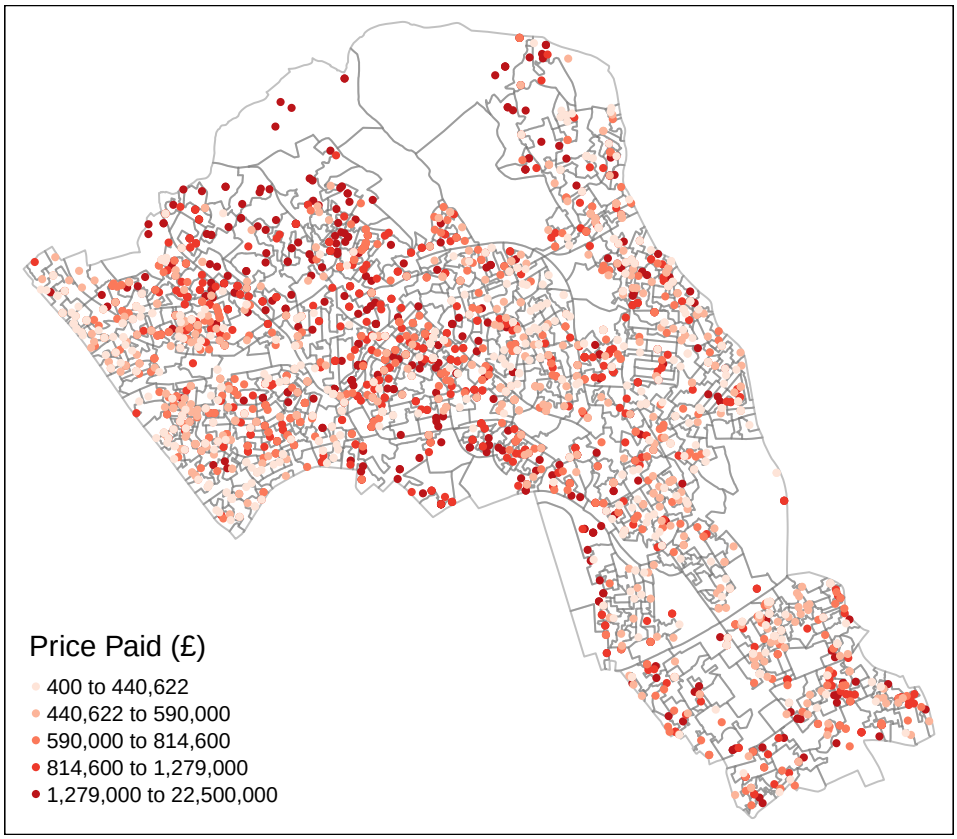
Which is entered into R like this:

```
# creates a coloured dot map
tm_shape(OA.Census) + tm_borders(alpha=.4) +
tm_shape(House.Points) + tm_dots(col = "Price", palette = "Reds", style = "quantile")
```



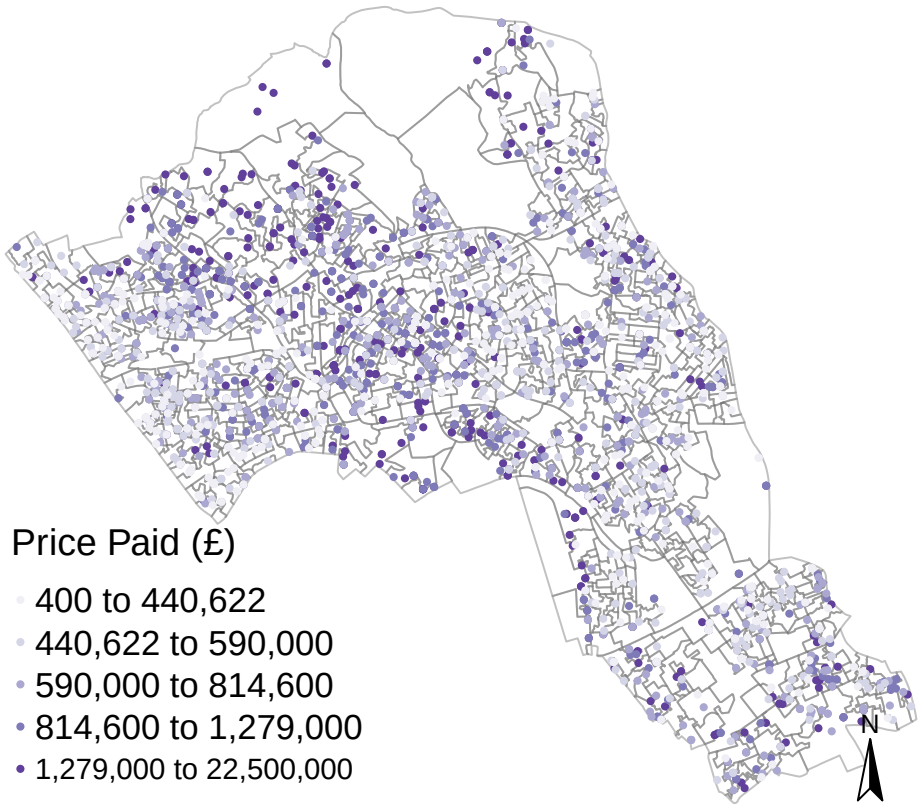
We can also add in more arguments within the `tm_dots()` function for points like we would with `tm_fill()` for polygon data. Some arguments are unique to `tm_dots()`, for example the option `scale`.

```
# creates a coloured dot map
tm_shape(OA.Census) + tm_borders(alpha=.4) +
tm_shape(House.Points) + tm_dots(col = "Price", scale = 1.5, palette = "Reds",
                                 style = "quantile", title = "Price Paid (£)")
```



We can also add `tm_layout()` and `tm_compass()` as we did in the previous practical.

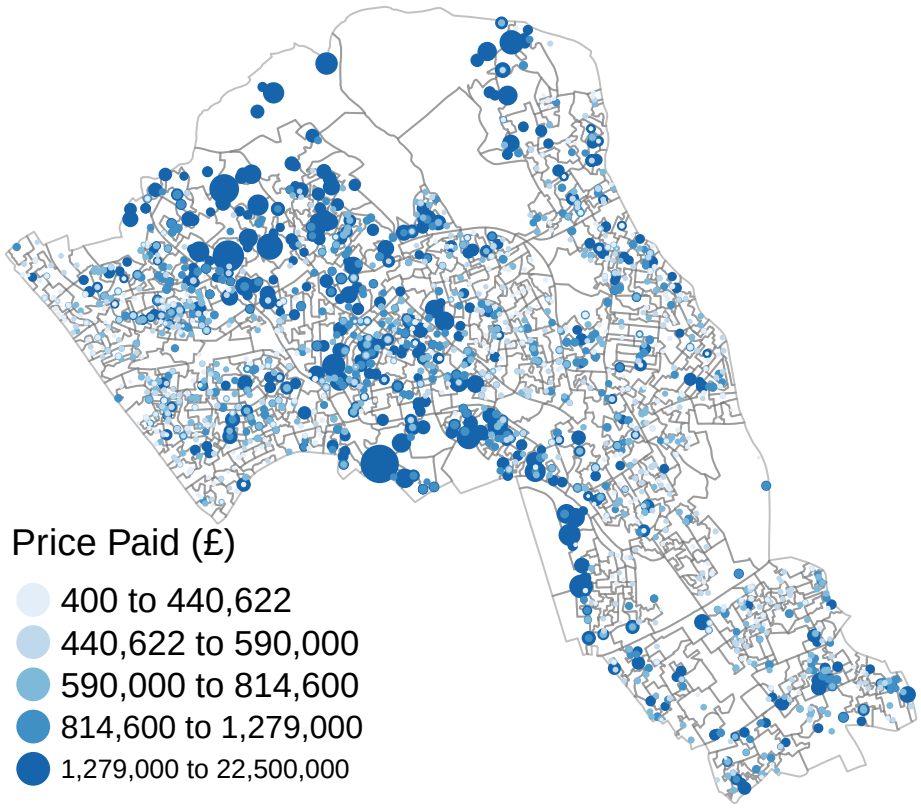
```
# creates a coloured dot map
tm_shape(OA.Census) + tm_borders(alpha=.4) +
tm_shape(House.Points) + tm_dots(col = "Price", scale = 1.5, palette = "Purples",
                                 style = "quantile", title = "Price Paid (£)")  +
tm_compass() +
tm_layout(legend.text.size = 1.1, legend.title.size = 1.4, frame = FALSE)
```



Creating proportional symbol maps

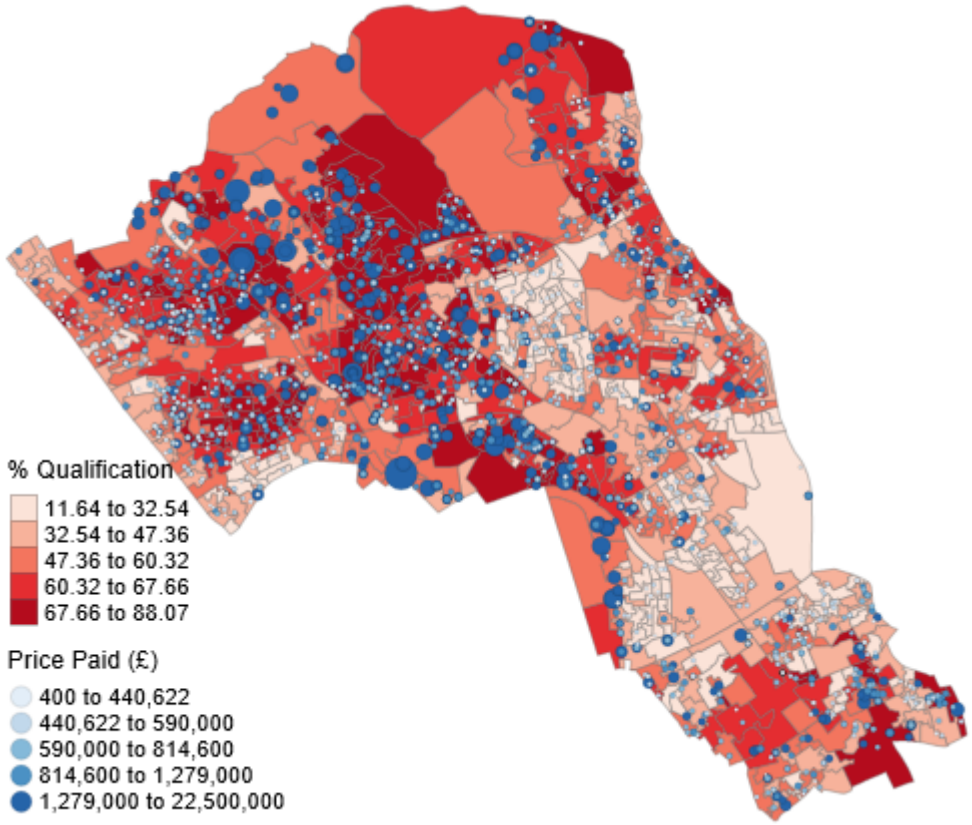
We can also create proportional symbols in tmap. To do it in tmap, we replace the `tm_dots()` function with the `tm_bubbles()` function. In the example below, the size and colours are both set as the price column.

```
# creates a proportional symbol map
tm_shape(OA.Census) + tm_borders(alpha=.4) +
tm_shape(House.Points) + tm_bubbles(size = "Price", col = "Price",
                                    palette = "Blues", style = "quantile",
                                    legend.size.show = FALSE,
                                    title.col = "Price Paid (£)") +
tm_layout(legend.text.size = 1.1, legend.title.size = 1.4, frame = FALSE)
```



We can also make the polygon shapefile display one of our census variables as a choropleth map as shown below. In this example we have also added some more parameters within the tm_bubbles() function to create thin borders around the bubbles.

```
# creates a proportional symbol map
tm_shape(OA.Census) + tm_fill("Qualification", palette = "Reds",
                              style = "quantile", title = "% Qualification") +
tm_borders(alpha=.4) +
tm_shape(House.Points) + tm_bubbles(size = "Price", col = "Price",
                                    palette = "Blues", style = "quantile",
                                    legend.size.show = FALSE,
                                    title.col = "Price Paid (£)",
                                    border.col = "black", border.lwd = 0.1,
                                    border.alpha = 0.1) +
tm_layout(legend.text.size = 0.8, legend.title.size = 1.1, frame = FALSE)
```



Saving the shapefile

Finally, we can write the newly formed House.Points shapefile to our working directory if you wish to save it for later use.

```
# write the shapefiel to your computer (remember to chang the dsn to your workspace)
writeOGR(House.Points, dsn = "C:/Users/Guy/Documents/Teaching/CDRC/Practicals",
         layer =  "Camden_house_sales", driver="ESRI Shapefile")
```

Practical 7: Using R as a GIS

This practical is intended to provide a demonstration of some of the basic spatial functionality of R by taking you through a small number of commonly employed techniques. Data for the practical can be downloaded from the **Introduction to Spatial Data Analysis and Visualisation in R** homepage.

In this tutorial we will:

- Run a point-in-polygon operation
- Create buffers
- Add backing maps from Google
- Create interactive maps with tmap

First we must set the working directory and load the practical data.

```
# Set the working directory
setwd("C:/Users/Guy/Documents/Teaching/CDRC/Practicals")

# Load the data. You may need to alter the file directory
Census.Data <-read.csv("practical_data.csv")
```

We will also need to load the spatial data files from the previous practicals.

```
# load the spatial libraries
library("sp")
library("rgdal")
library("rgeos")

# Load the output area shapefiles
Output.Areas <- readOGR(".", "Camden_oa11")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "Camden_oa11"
## with 749 features
## It has 1 fields
```

```
# join our census data to the shapefile
OA.Census <- merge(Output.Areas, Census.Data, by.x="OA11CD", by.y="OA")

# load the houses point files
House.Points <- readOGR(".", "Camden_house_sales")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "Camden_house_sales"
## with 2547 features
## It has 4 fields
```

We can now commence our operations ##Point-in-polygon

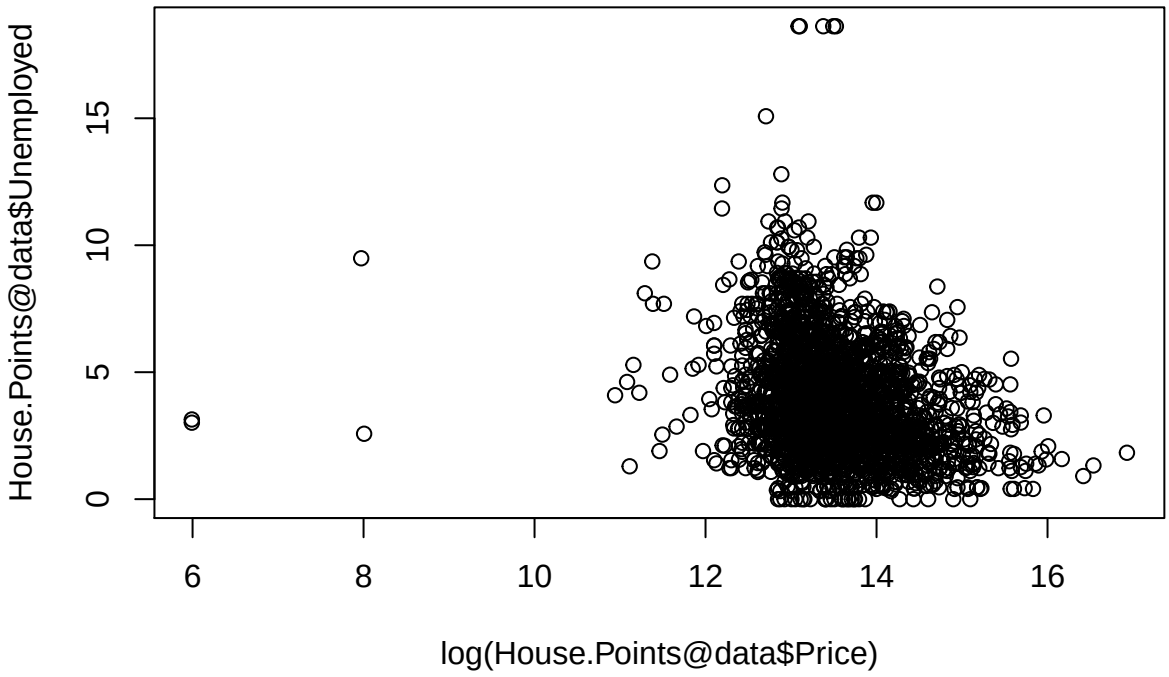
Firstly, we will aim to aggregate our point data into the Output Area polygons using a point in polygon operation.

As we are commencing a spatial operation, both files need to be projected using the same coordinate reference system. In this case, we have ensured that both spatial files have been set to British National Grid (27700).

```
proj4string(OA.Census) <- CRS("+init=EPSG:27700")
proj4string(House.Points) <- CRS("+init=EPSG:27700")
```

With the projections set, it is now possible to assign each house point the characteristics of the output area polygon it falls within.

```
# point in polygon. Gives the points the attributes of the polygons that they are in
pip <- over(House.Points, OA.Census)

# need to bind the census data to our original points
House.Points@data <- cbind(House.Points@data, pip)

View(House.Points@data)

# it is now possible to plot the house prices and local unemployment rates
plot(log(House.Points@data$Price), House.Points@data$Unemployed)
```



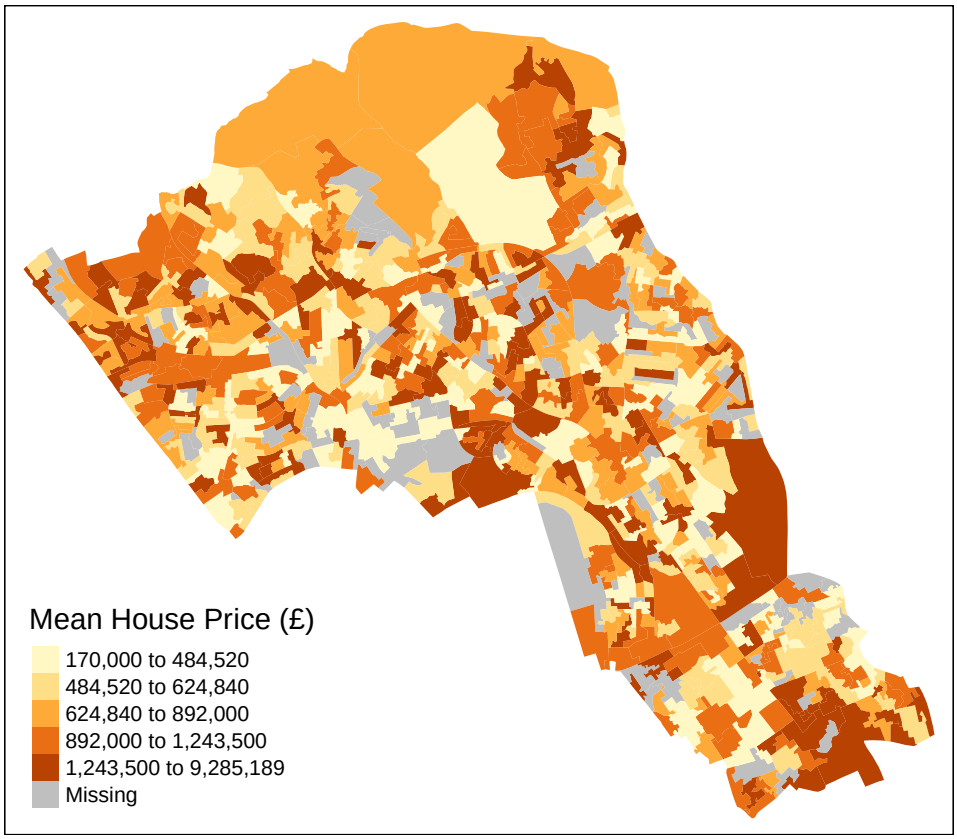
It is also useful to measure the average house prices for each output area. We don't need to run another point in polygon operation, instead we can reaggregate the data by the **OA11CD** column so every output area has one record. Using the `aggregate()` function we can decide what numbers are returned from our house prices (i.e. mean, sum, median).

```
# first we aggregate the house prices by the OA11CD (OA names) column
# we ask for the mean for each OA
OA <- aggregate(House.Points@data$Price, by = list(House.Points@data$OA11CD), mean)

# change the column names of the aggregated data
names(OA) <- c("OA11CD", "Price")

# join the aggregated data back to the OA.Census polygon
OA.Census@data <- merge(OA.Census@data, OA, by = "OA11CD", all.x = TRUE)
```

We can now map the data using tmap. We will have missing data where there are output areas where no

houses were sold in 2015.

```
library(tmap)

tm_shape(OA.Census) + tm_fill(col = "Price", style = "quantile", title = "Mean House Price (£)")
```



It is also possible to now run a linear model between our unemployment variable from the 2011 Census and our new average house price variable.

```
model <- lm(OA.Census@data$Price ~ OA.Census@data$Unemployed)
summary(model)
```

```
##
## Call:
## lm(formula = OA.Census@data$Price ~ OA.Census@data$Unemployed)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -977841 -364618 -138499  145581 8053263
##
## Coefficients:
##                              Estimate Std. Error t value Pr(>|t|)
## (Intercept)                   1434296      58129  24.675   <2e-16 ***
## OA.Census@data$Unemployed  -110798      11754  -9.426   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 748300 on 652 degrees of freedom
```

```
##   (95 observations deleted due to missingness)
## Multiple R-squared:  0.1199, Adjusted R-squared:  0.1186
## F-statistic: 88.85 on 1 and 652 DF,  p-value: < 2.2e-16
```

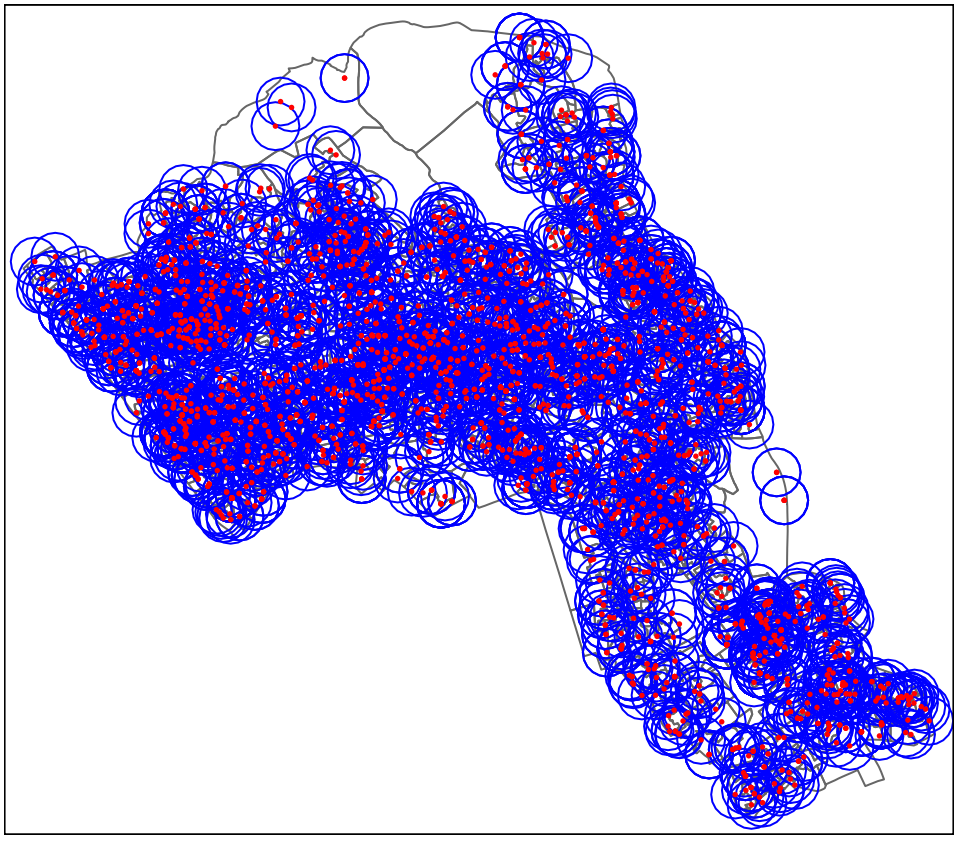
Buffers

A second technique we will demonstrate is buffering. This is a GIS process by which we create linear catchments for each data point based on distance. This simple technique is commonly used to determine which areas are proximal to certain objects. Here we use the `gBuffer()` function from the rgeos package.

```
# create 200m buffers for each house point
house_buffers <- gBuffer(House.Points, width = 200, byid = TRUE)
```

We can plot these in tmap.

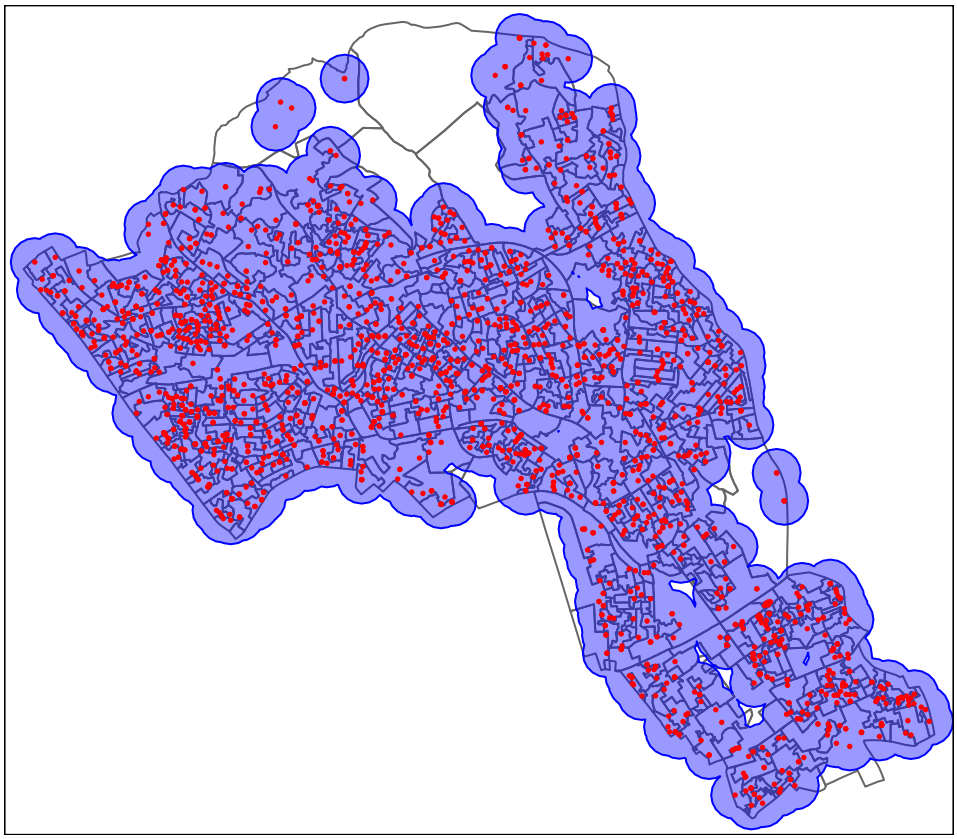
```
# map in tmap
tm_shape(OA.Census) + tm_borders() +
tm_shape(house_buffers) + tm_borders(col = "blue") +
tm_shape(House.Points) + tm_dots(col = "red")
```



Union

We can merge all of the buffers together using a process known as a union. This process will join all intersecting geometries.

```
# merges the buffers
union.buffers <- gUnaryUnion(house_buffers)

# map in tmap
tm_shape(OA.Census) + tm_borders() +
tm_shape(union.buffers) + tm_fill(col = "blue", alpha = .4) + tm_borders(col = "blue") +
tm_shape(House.Points) + tm_dots(col = "red")
```

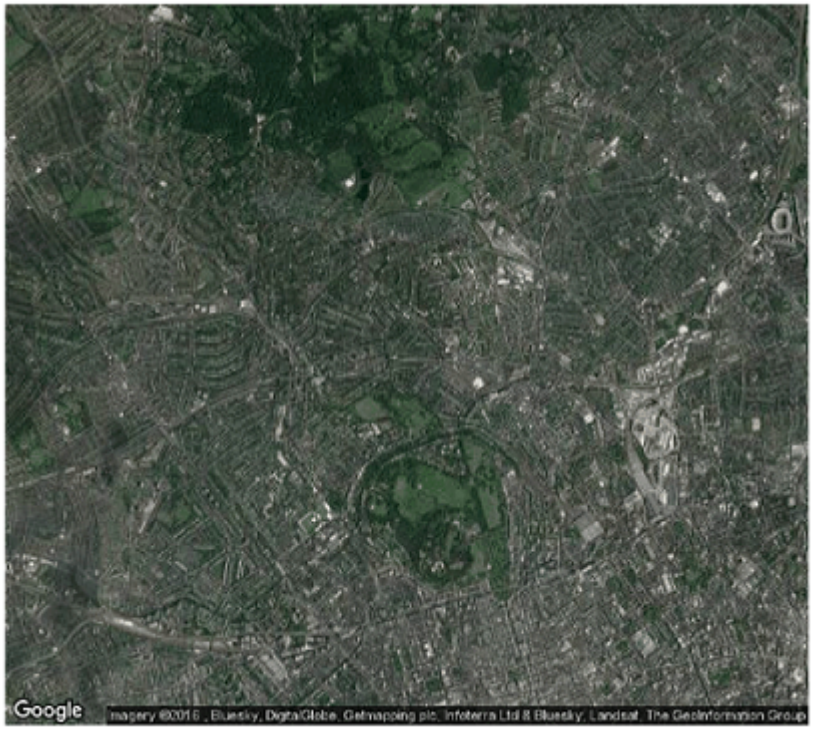


Adding Backing Maps

Adding backing maps is sometimes more complicated then it first seems as each mapping provider may use a different coordinate reference system. In this example, we will download backing maps from Google and project our data on to them.

```
library(raster)
library(dismo)

google.map <- gmap("Camden, London", type = "satellite")
```

You can plot these with `plot(google.map)`



We can also change the type of map we wish to download, and then write it to our file space. For instance.

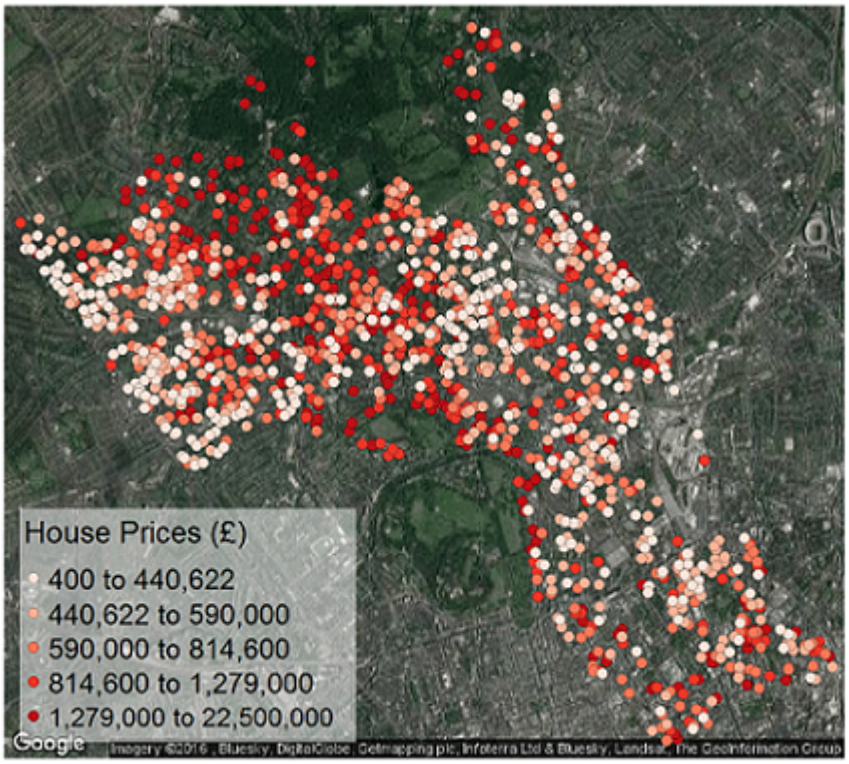
```
#save the map
google.map <- gmap("Camden, London", type = "roadmap", filename = "Camden.gmap")
```

Next we want to plot our house points on to this map. However, our house data is projected via the British National Grid coordinate system, while Google uses the web Mercator projection

```
# convert points first
CRS.new <- CRS("+proj=longlat +ellps=WGS84 +datum=WGS84")
reprojected.houses <- spTransform(House.Points, CRS.new)
```

We can now map both the base map and reprojected house points in tmap. We need to treat the backing map as a raster image.

```
# maps the base and reprojected house points in tmap
tm_shape(google.map) + tm_raster() +
tm_shape(reprojected.houses) + tm_dots(col = "Price", style = "quantile",
                                       scale = 2.5, palette = "Reds",
                                       title = "House Prices (£)",
                                       border.col = "black",
                                       border.lwd = 0.1,
                                       border.alpha = 0.4)  +
tm_layout(legend.position = c("left", "bottom"),  legend.text.size = 1.1,
          legend.title.size = 1.4, frame = FALSE,
          legend.bg.color = "white", legend.bg.alpha = 0.5)
```



Creating Interactive Maps

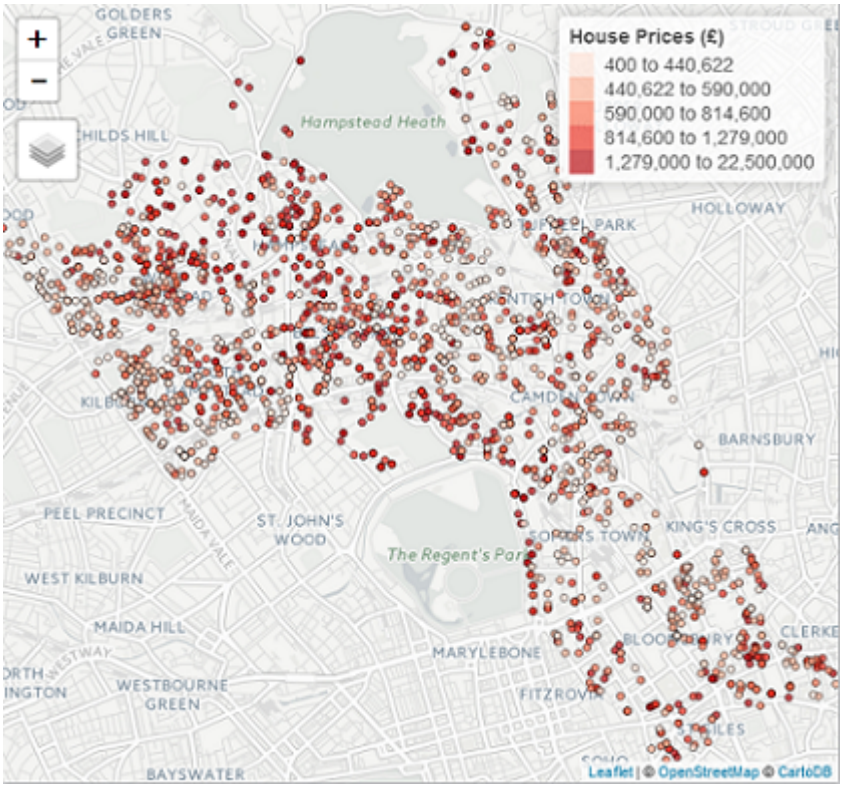
We have now sussed the basics of mapping in R using the tmap package. However, all of the maps produced so far have been static images. It is possible to create interactive 'slippy' maps through which users can zoom in and out, and even turn layers on and off. This can be very easily done in tmap using the same techniques as before. Please note that this technique only works on the most recent editions of RStudio.

The first thing we need to do is to tell R to switch our tmap mode from `plot` to `view`.

```
# interactive maps in tmap
library(leaflet)

# turns view map on
tmap_mode("view")
```

With view mode turned on, every map we now produce with tmap will be produced as an interactive slippy map. First, we will map our house price data (note we do not need to include any attributes on layout anymore).

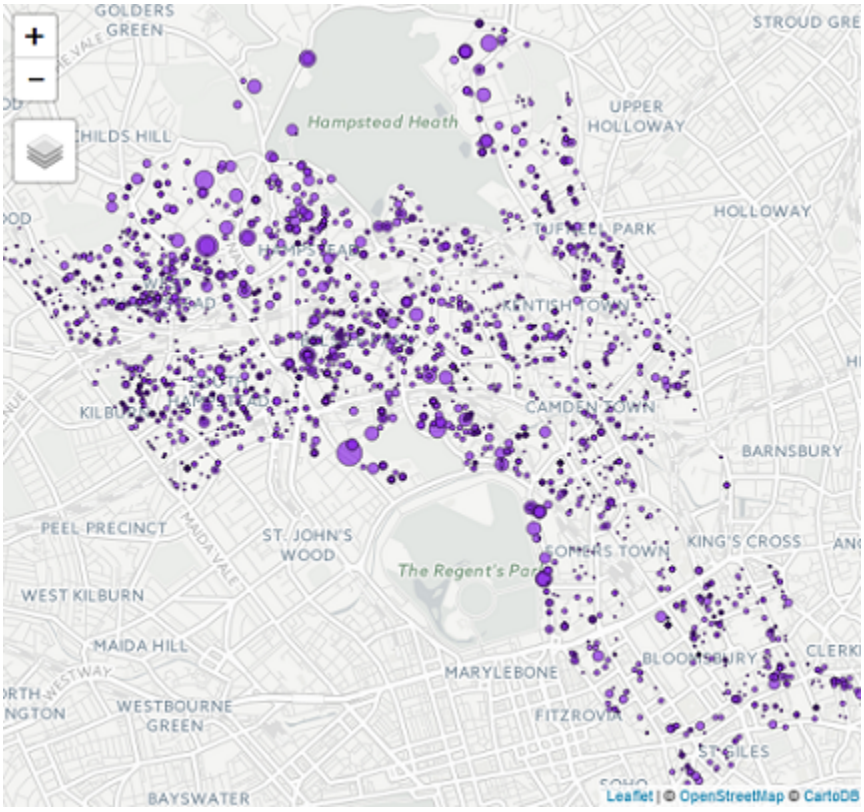
```
tm_shape(House.Points) + tm_dots(title = "House Prices (£)", border.col = "black",
                                 border.lwd = 0.1, border.alpha = 0.2, col = "Price",
                                 style = "quantile", palette = "Reds")
```



The map is interactive. You can zoom in using the controls in the top left corner, you can also change the backing map and turn off layers we have included. In addition, you can click on data in the map to view the attributes for that point. The default base map is a grey map system known as *CartpDB.Positron*. The advantage of the grey shades is that they won't clash with our coloured data. You can also click on the layer option to switch to the other map options which include *OpenStreetMap*.

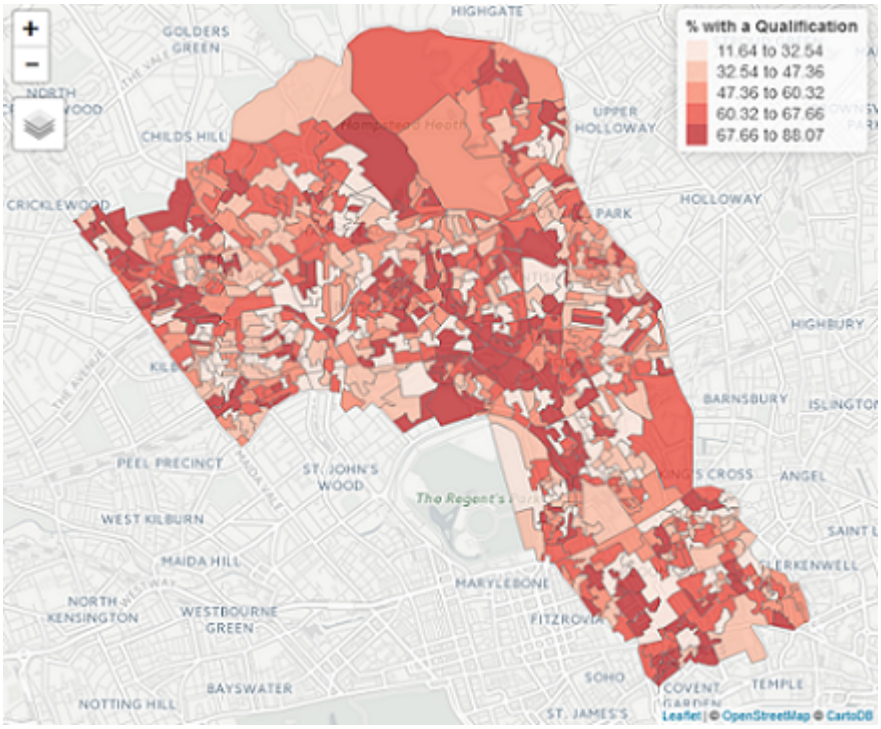
The interactive version of tmap works with all of the tmap functions. For example, we can make bubble plots using `tm_bubbles()` as we did in the previous practical.

```
tm_shape(House.Points) + tm_bubbles(size = "Price", title.size = "House Prices (£)",
                                    border.col = "black", border.lwd = 0.1,
                                    border.alpha = 0.4, legend.size.show = TRUE)
```



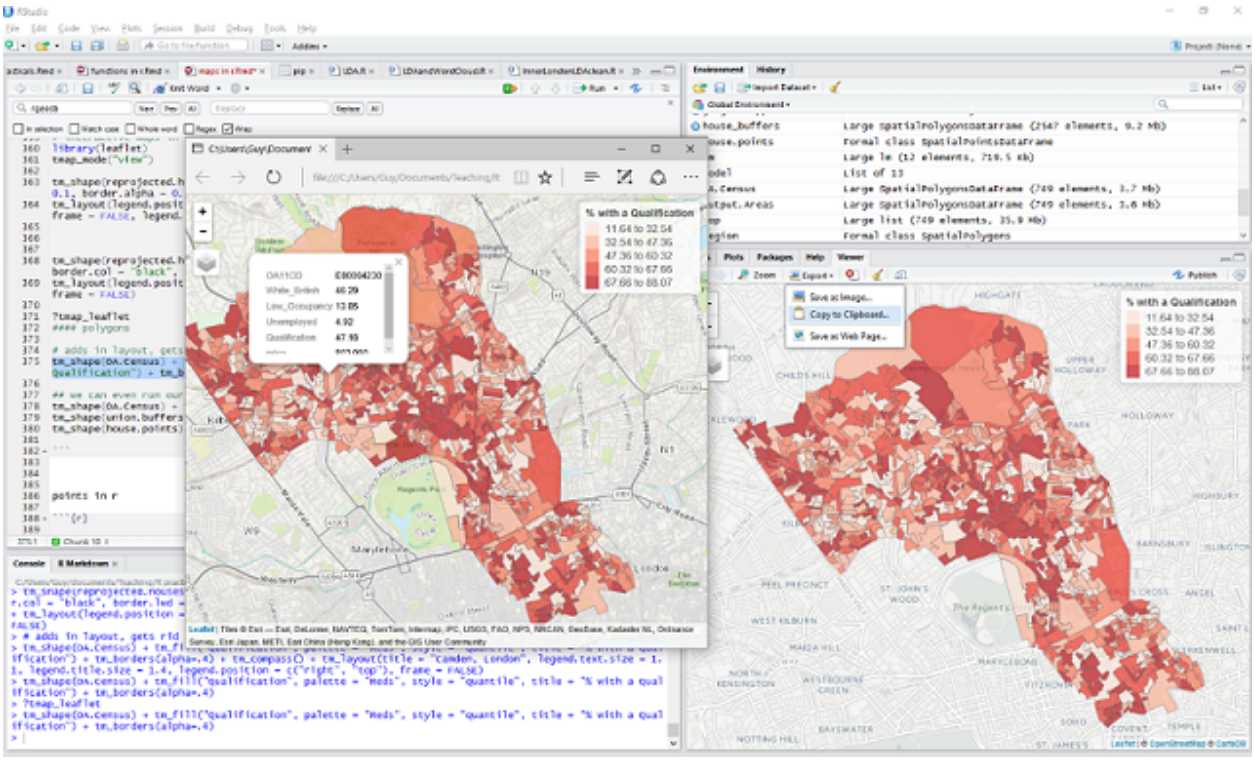
We can also map polygons...

```
tm_shape(OA.Census) + tm_fill("Qualification", palette = "Reds", style = "quantile",
                              title = "% with a Qualification") + tm_borders(alpha=.4)
```



Finally, if you wish to export your map as an interactive webpage click on export in the plots window go

click on **Save as Web Page...**. The html file can either be viewed locally on a web browser, or the code can be inputted into a website so it can displayed on the internet as an interactive widget.



To turn tmap back to the plot view simply write `tmap_mode("plot")`.

If you are interested in more advanced methods for customising interactive maps, please explore the leaflet package in R.

Practical 8: Representing Densities in R

This practical will introduce you to running a kernel density estimation in R. Kernel density estimation is a commonly used means of representing densities of spatial data points. The technique produces a smooth and continuous surface where each pixel represents a density value based on the number of points within a given distance bandwidth. We will also cover some basic techniques in handling and editing raster shapefiles. Data for the practical can be downloaded from the **Introduction to Spatial Data Analysis and Visualisation in R** homepage.

In this tutorial we will:

- Run a kernel density estimation
- Create a raster shapefile
- Map a raster shapefile in tmap
- Use a mask technique to clip a raster

First we must set the working directory and load the practical data.

```
# Set the working directory
setwd("C:/Users/Guy/Documents/Teaching/CDRC/Practicals")
```

We will also need to the load the spatial data files from the previous practicals.

```
# load the spatial libraries
library("sp")
library("rgdal")
library("rgeos")

# Load the output area shapefiles, we won't join it to any data this time
Output.Areas <- readOGR(".", "Camden_oa11")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "Camden_oa11"
## with 749 features
## It has 1 fields
```

```
# load the houses point files
House.Points <- readOGR(".", "Camden_house_sales")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "Camden_house_sales"
## with 2547 features
## It has 4 fields
```

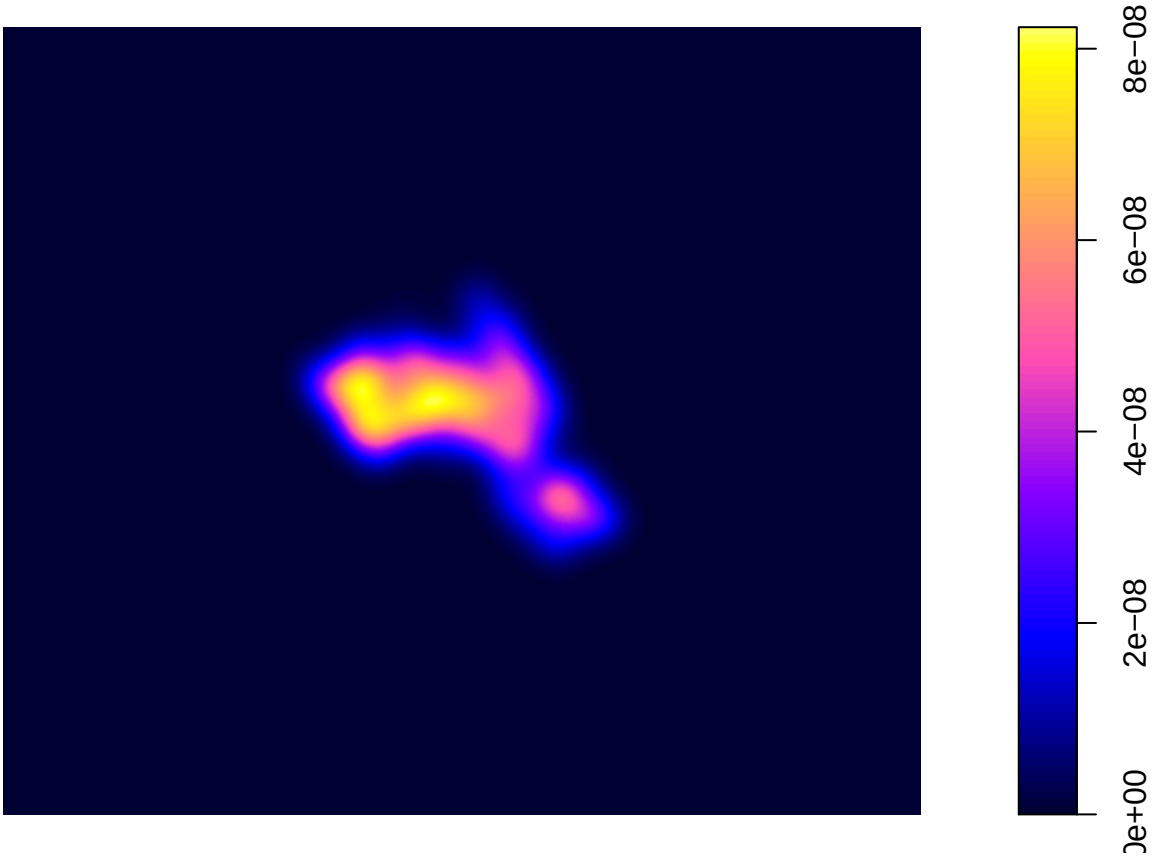
Running a kernel density estimation

Whilst it is straight-forward to determine the frequency of a phenomena across space with polygon data, it is more complicated to measure densities of points coherently. A relatively simple approach would be to use polygon zones to spatially aggregate the data (as we did in practical 7), and to then count the number of occurances in each area.

However, techniques which have been devised to represent the densities of data points across two-dimensions also exist. One such approach, known as kernel density estimation, uses a moving quadrant to calculate the density for each area within a given threshold. For more information on the statistics behind kernel density measures please read the Geospatial Ananalysis (de Smith *et al*, 2015) webpage on *density, kernels and occupancy*

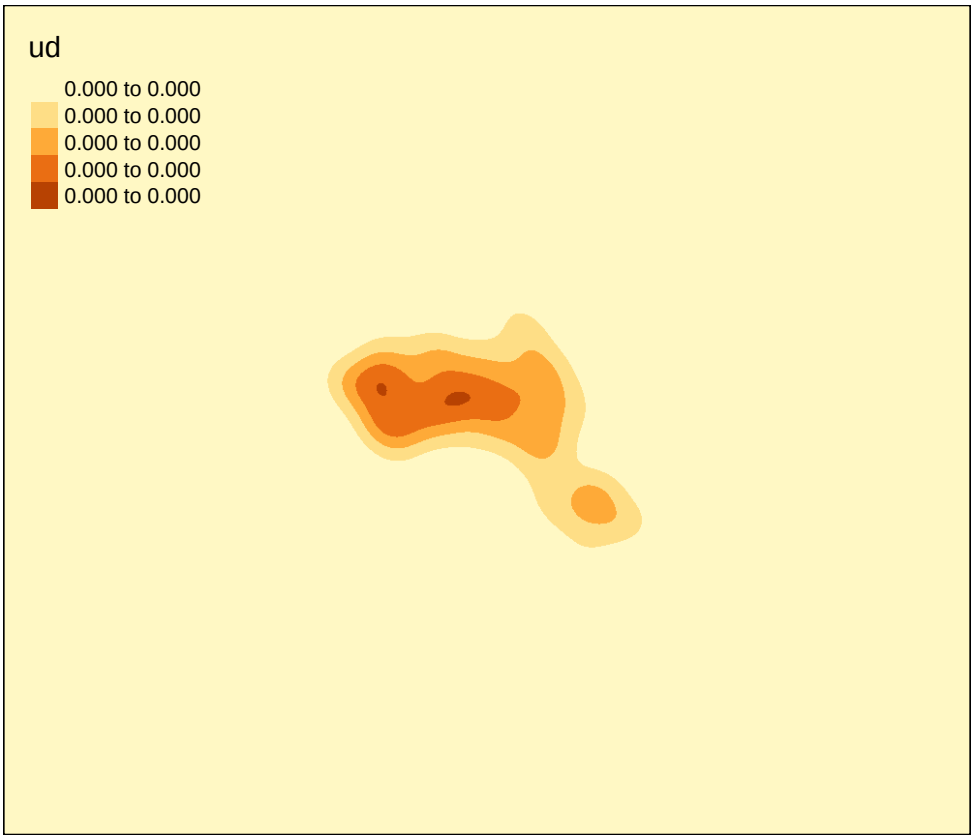
The following code will run a kernel density estimation for our Land Registry house price data. There are several different ways to run this through R, we will use the functions available from the *adehabitatHR* package.

```
# load the spatial libraries
library(raster)
library(adehabitatHR)

# runs the kernel density estimation,look up the function parameters for more options
kde.output <- kernelUD(House.Points, h="href", grid = 1000)

plot(kde.output)
```



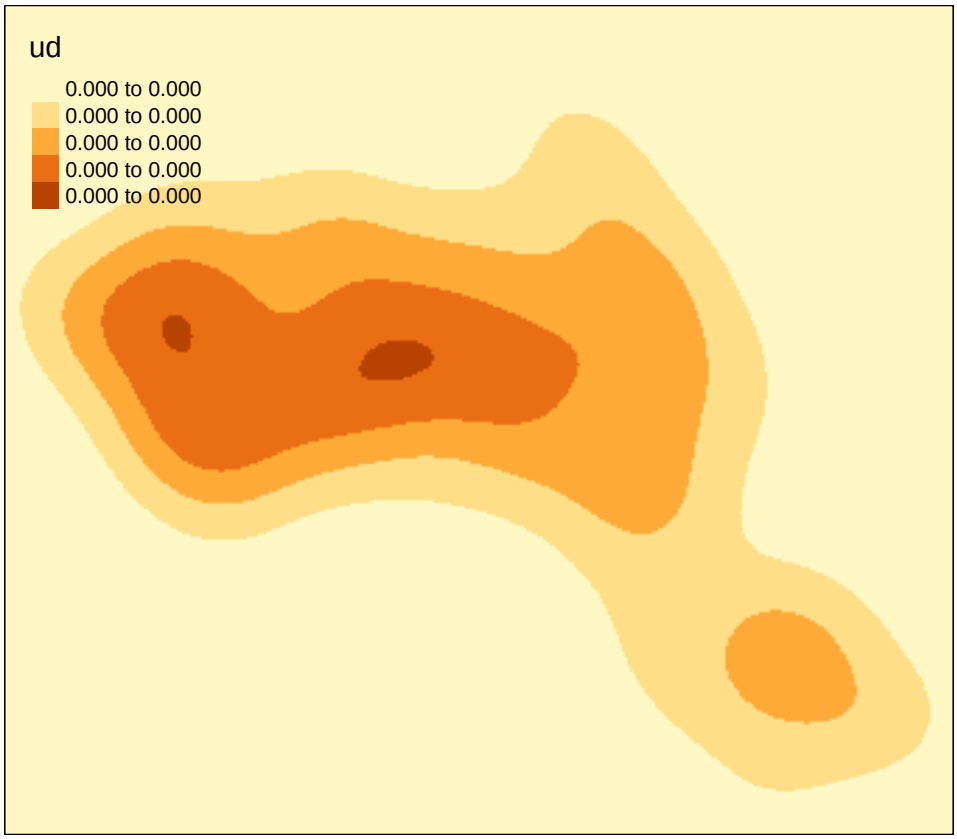
We can also create contour plots in R. Simply enter `contour(kde)` into R. To map the raster in tmap, we first need to ensure it has been projected correctly.

```
# converts to raster
kde <- raster(kde.output)
# sets projection to British National Grid
projection(kde) <- CRS("+init=EPSG:27700")

library(tmap)

# maps the raster in tmap, "ud" is the density variable
tm_shape(kde) + tm_raster("ud")
```



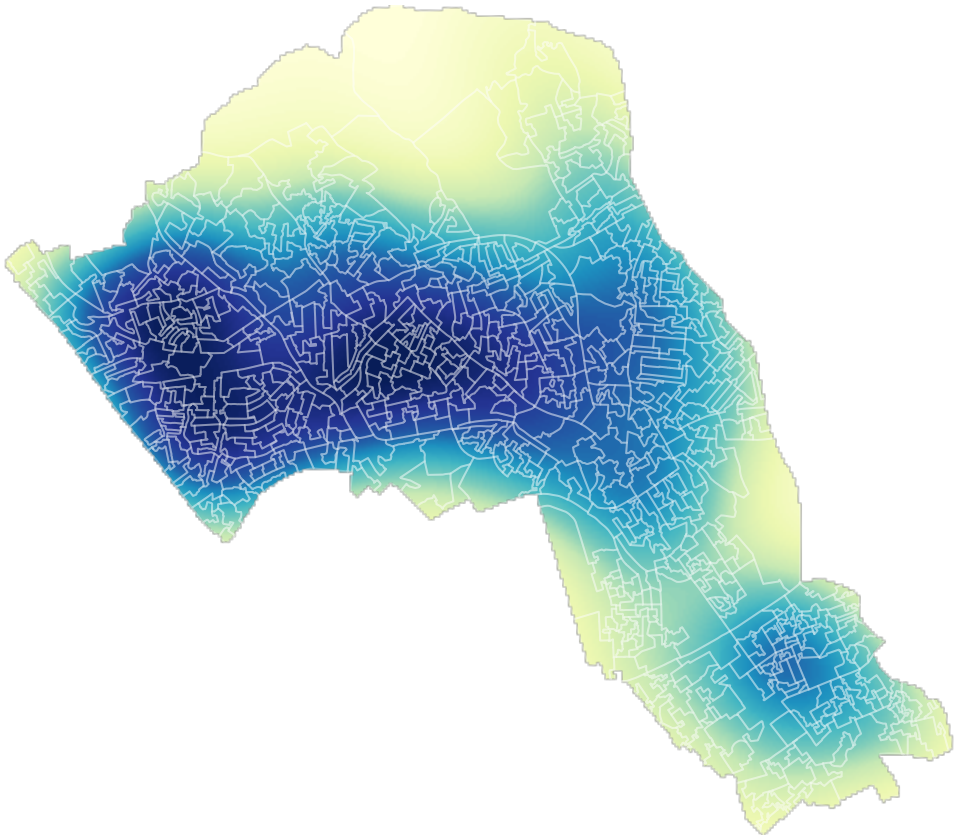
We can just about make out the shape of Camden. However, in this case the raster includes a lot of empty space, we can zoom in on Camden by setting the map to the extents of a bounding box.

```
# creates a bounding box based on the extents of the Output.Areas polygon
bounding_box <- bb(Output.Areas)

# maps the raster within the bounding box
tm_shape(kde, bbox = bounding_box) + tm_raster("ud")
```



We can also mask (or clip) the raster by the output areas polygon and tidy up the graphic. This operation only preserves the parts of the raster which are within the spatial extent of the masking polygon.

```
# mask the raster by the output area polygon
masked_kde <- mask(kde, Output.Areas)

# maps the masked raster, also maps white output area boundaries
tm_shape(masked_kde, bbox = bounding_box) + tm_raster("ud", style = "quantile",
                                                      n = 100,
                                                      legend.show = FALSE,
                                                      palette = "YlGnBu") +
tm_shape(Output.Areas) + tm_borders(alpha=.3, col = "white") +
tm_layout(frame = FALSE)
```



Identifying homeranges

We can also create catchment boundaries from the kernel density estimations.

```
# compute homeranges for 75%, 50%, 25% of points,
# objects are returned as spatial polygon data frames
range75 <- getverticeshr(kde.output, percent = 75)
range50 <- getverticeshr(kde.output, percent = 50)
range25 <- getverticeshr(kde.output, percent = 25)
```

With the ranges calculated we can then map them together in tmap. Notice that this tmap combines multiple layers. Remember the layer at the bottom of the list is the last one to be printed, and therefore will appear at the front of the graphic.

Upon first glance the code looks quite complicated. So to summarise, each line does the following...

```
.   Create a grey background using the Output.Areas polygon with white borders
.   Plot the locations of houses using House.Points
.   Plot the 75% range, set attributes for border and fill (i.e. colour, transparency, line width)
.   Plot the 50% range, set attributes for border and fill (i.e. colour, transparency, line width)
.   Plot the 25% range, set attributes for border and fill (i.e. colour, transparency, line width)
.   The outside frame is removed
```

```
# the code below creates a map of several layers using tmap
tm_shape(Output.Areas) + tm_fill(col = "#f0f0f0") + tm_borders(alpha=.8, col = "white") +
tm_shape(House.Points) + tm_dots(col = "blue") +
tm_shape(range75) + tm_borders(alpha=.7, col = "#fb6a4a", lwd = 2) +
```

```
  tm_fill(alpha=.1, col = "#fb6a4a") +
tm_shape(range50) + tm_borders(alpha=.7, col = "#de2d26", lwd = 2) +
  tm_fill(alpha=.1, col = "#de2d26") +
tm_shape(range25) + tm_borders(alpha=.7, col = "#a50f15", lwd = 2) +
  tm_fill(alpha=.1, col = "#a50f15") +
tm_layout(frame = FALSE)
```

Now try adding on an addition range for the densest 10% of our data.

Whilst mapping the densities of house sales is reasonably interesting, this technique can be applied to all sorts of point data. You could, for example, get two sets of data and create two separate ranges to compare their distributions - i.e. two species of animals.

If you wish to save your raster files to your computer, you can use the WriteRaster() function.

```
writeRaster(masked_kde, filename = "kernel_density.grd")
```

Practical 9: Measuring Spatial Autocorrelation in R

An Introduction to Spatial Data Analysis and Visualisation in R - Guy Lansley & James Cheshire (2016)

This practical will cover how to run various measures of spatial autocorrelation in R. We will consider both statistics of global spatial autocorrelation and how to identify spatial clustering across our study area. Data for the practical can be downloaded from the **Introduction to Spatial Data Analysis and Visualisation in R** homepage.

In this practical we will:

- Run a global Spatial autocorrelation for a shapefile
- Identify local indicators of spatial autocorrelation
- Run a Getis-Ord

First, we must set the working directory and load the practical data.

```
# Set the working directory
setwd("C:/Users/Guy/Documents/Teaching/CDRC/Practicals")

# Load the data. You may need to alter the file directory
Census.Data <-read.csv("practical_data.csv")
```

We will also need to load the spatial data files from the previous practicals.

```
# load the spatial libraries
library("sp")
library("rgdal")
library("rgeos")

# Load the output area shapefiles
Output.Areas <- readOGR(".", "Camden_oa11")
```

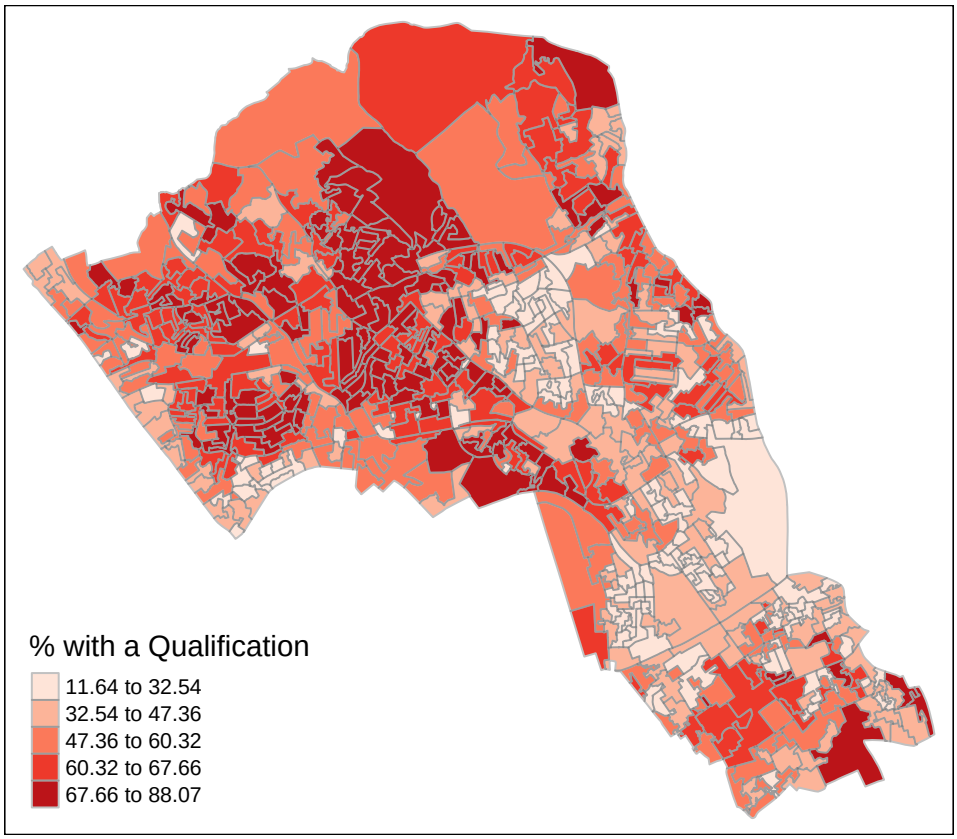
```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "Camden_oa11"
## with 749 features
## It has 1 fields
```

```
# join our census data to the shapefile
OA.Census <- merge(Output.Areas, Census.Data, by.x="OA11CD", by.y="OA")

# load the houses point files
House.Points <- readOGR(".", "Camden_house_sales")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "Camden_house_sales"
## with 2547 features
## It has 4 fields
```

Remember the distribution of our qualification variable? We will be working on that today. We have first mapped it to remind us of its spatial distribution across our study area.

```
library("tmap")

tm_shape(OA.Census) + tm_fill("Qualification", palette = "Reds", style = "quantile",
                              title = "% with a Qualification") + tm_borders(alpha=.4)
```



Running a spatial autocorrelation

A spatial autocorrelation measures how distance influences a particular variable. In other words, it quantifies the degree of which objects are similar to nearby objects. Variables are said to have a positive spaital autocorrelation when similar values tend to be nearer together than dissimilar values.

Waldo Tober's first law of geography is that *"Everything is related to everything else, but near things are more related than distant things."* so we would expect most geographic phenomena to exert a spatial autocorrelation of some kind. In population data this is often the case as persons of similar characteristics tend to reside in similar neighbourhoods due to a range of reasons including house prices, proximity to work places and cultural factors.

We will be using the spatial autocorrelation functions available from the spdep package.

```
library(spdep)
```

Finding neighbours

In order for the subsequent model to work, we need to work out what polygons neighbour each other. The following code will calculate neighbours for our *OA.Census* polygon and print out the results below.

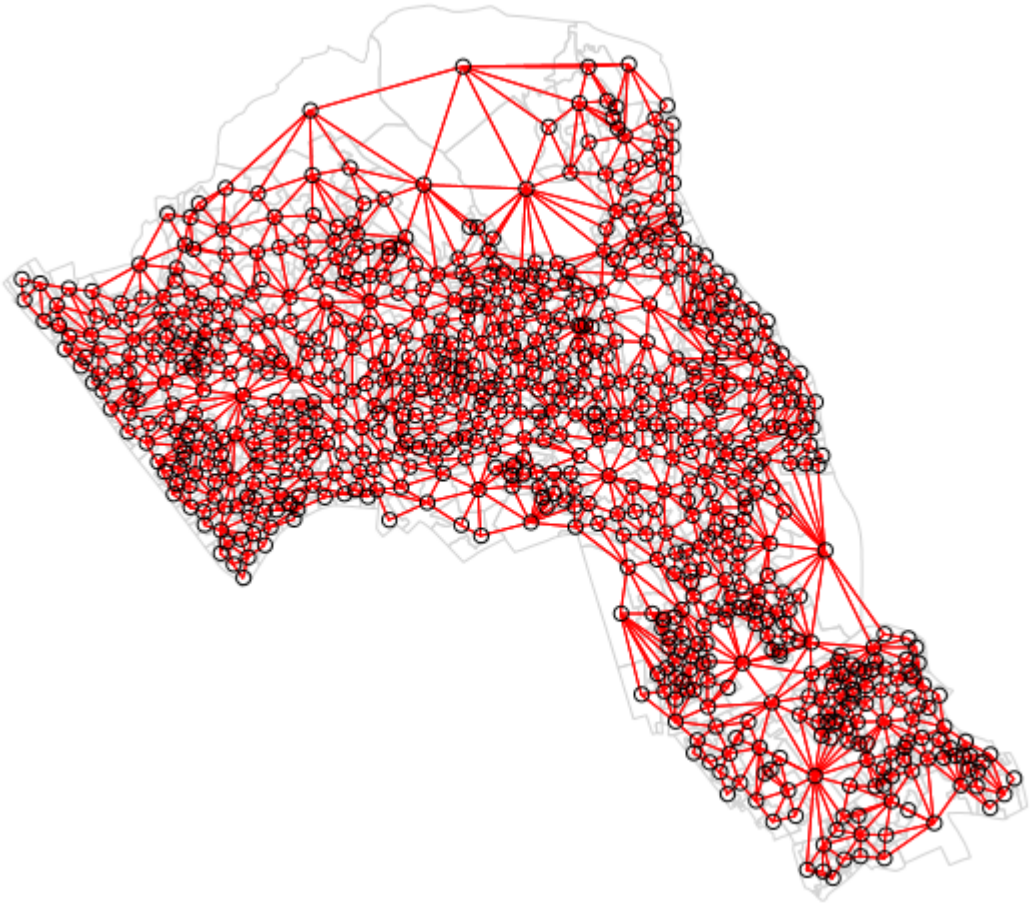
```
# Calculate neighbours
neighbours <- poly2nb(OA.Census)
neighbours
```

```
## Neighbour list object:
```

```
## Number of regions: 749
## Number of nonzero links: 4342
## Percentage nonzero weights: 0.7739737
## Average number of links: 5.797063
```

We can plot the links between neighbours to visualise their distribution across space.

```
plot(OA.Census, border = 'lightgrey')
plot(neighbours, coordinates(OA.Census), add=TRUE, col='red')
```



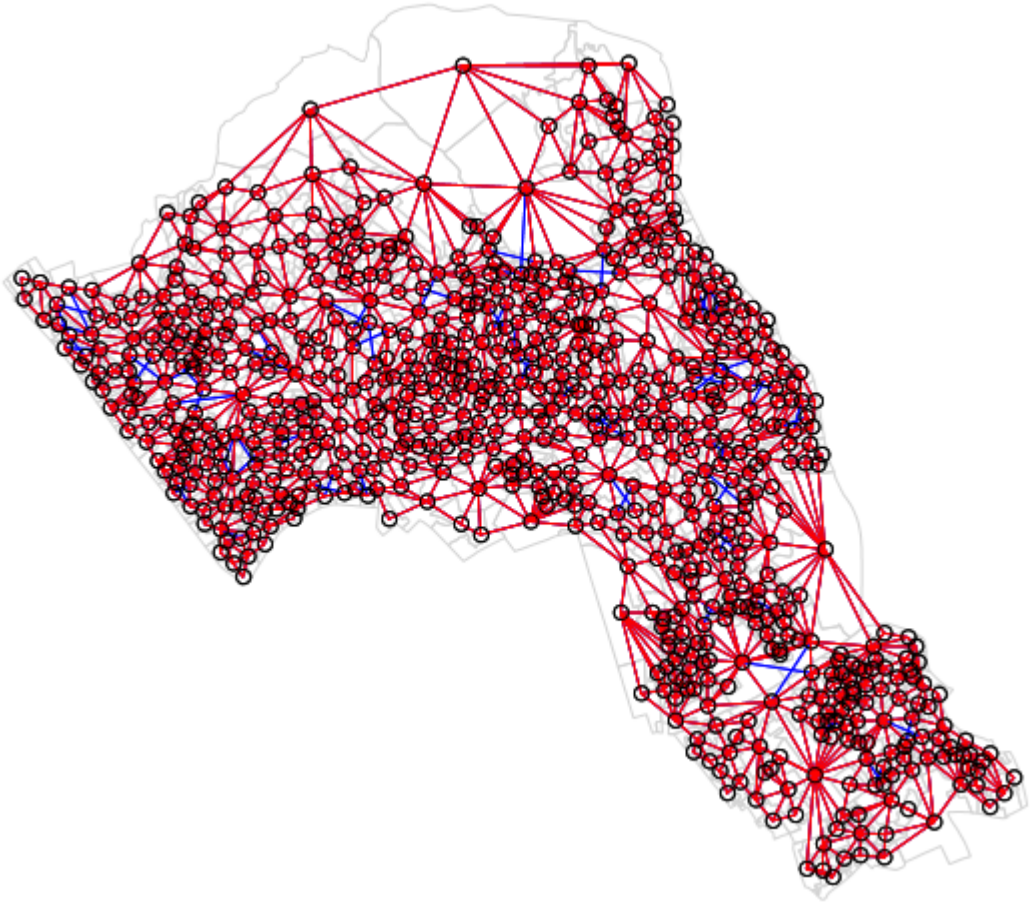
```
# Calculate the Rook's case neighbours
neighbours2 <- poly2nb(OA.Census, queen = FALSE)
neighbours2
```

```
## Neighbour list object:
## Number of regions: 749
## Number of nonzero links: 4176
## Percentage nonzero weights: 0.7443837
## Average number of links: 5.575434
```

We can already see that this approach has identified fewer links between neighbours. By plotting both neighbour outputs we can interpret their differences.

```
# compares different types of neighbours
plot(OA.Census, border = 'lightgrey')
```

```
plot(neighbours, coordinates(OA.Census), add=TRUE, col='blue')
plot(neighbours2, coordinates(OA.Census), add=TRUE, col='red')
```



We can represent spatial autocorrelation in two ways; globally or locally. Global models will create a single measure which represent the entire data whilst local models let us explore spatial clustering across space.

Running a global spatial autocorrelation

With the neighbours defined. We can now run a model. First we need to convert the data types of the neighbours object. This file will be used to determine how the neighbours are weighted

```
# Convert the neighbour data to a listw object
listw <- nb2listw(neighbours2)
listw
```

```
## Characteristics of weights list object:
## Neighbour list object:
## Number of regions: 749
## Number of nonzero links: 4176
## Percentage nonzero weights: 0.7443837
## Average number of links: 5.575434
##
```

```
## Weights style: W
## Weights constants summary:
##     n     nn  S0       S1       S2
## W 749 561001 749 285.3793 3113.982
```

We can now run the model. This type of model is known as a Moran's test. This will create a correlation score between -1 and 1. Much like a correlation coefficient, 1 determines perfect postive spatial autocorrelation (so our data is clustered), 0 identifies the data is randomly distributed and -1 represents negative spatial autocorrelation (so dissimilar values are next to each other).

```
# global spatial autocorrelation
moran.test(OA.Census$Qualification, listw)
```

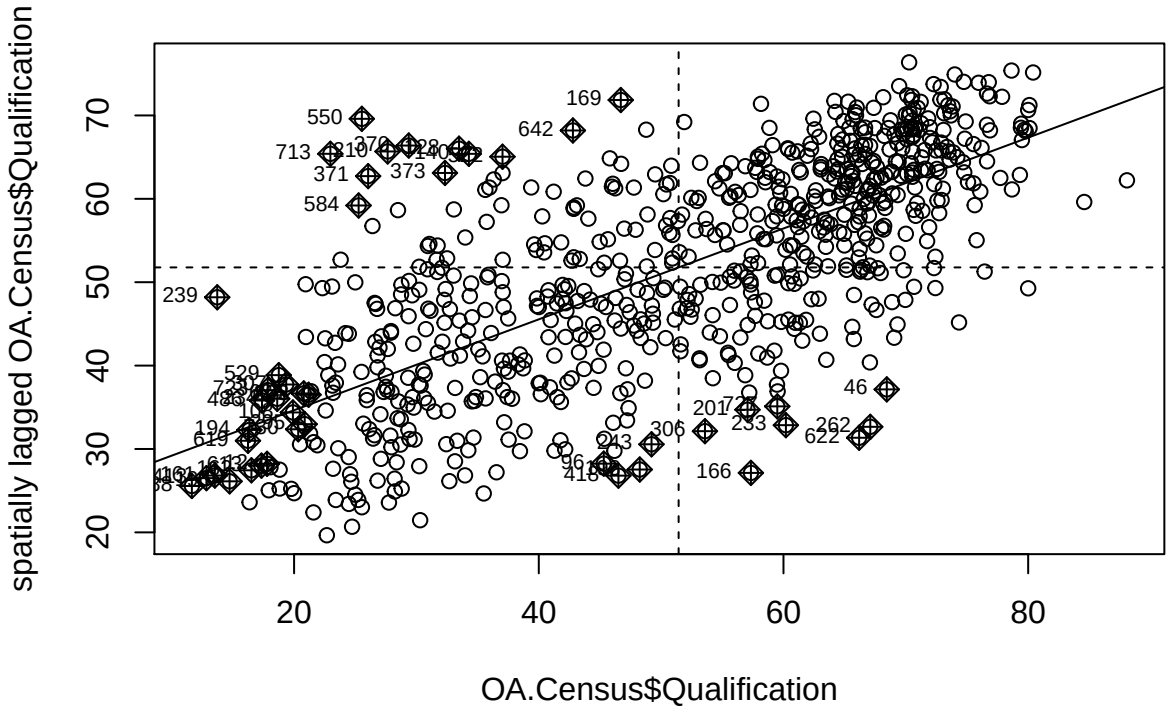
```
##
##  Moran I test under randomisation
##
## data:  OA.Census$Qualification
## weights: listw
##
## Moran I statistic standard deviate = 24.292, p-value < 2.2e-16
## alternative hypothesis: greater
## sample estimates:
## Moran I statistic       Expectation          Variance
##      0.5448699398     -0.0013368984      0.0005055733
```

The Moran I statistic is 0.54, we can therefore determine that there our qualification variable is positively autocorrelated in Camden. In other words, the data does spatially cluster. We can also consider the p-value as a measure of the statistical significance of the model.

Running a local spatial autocorrelation

We will first create a moran plot which looks at each of the values plotted against their spatially lagged values. It basically explores the relationship between the data and their neighbours as a scatter plot. The style refers to how the weights are coded. "W" weights are row standardised (sums over all links to n).

```
# creates a moran plot
moran <- moran.plot(OA.Census$Qualification, listw = nb2listw(neighbours2, style = "W"))
```



Is it possible to determine a positive relationship from observing the scatter plot?

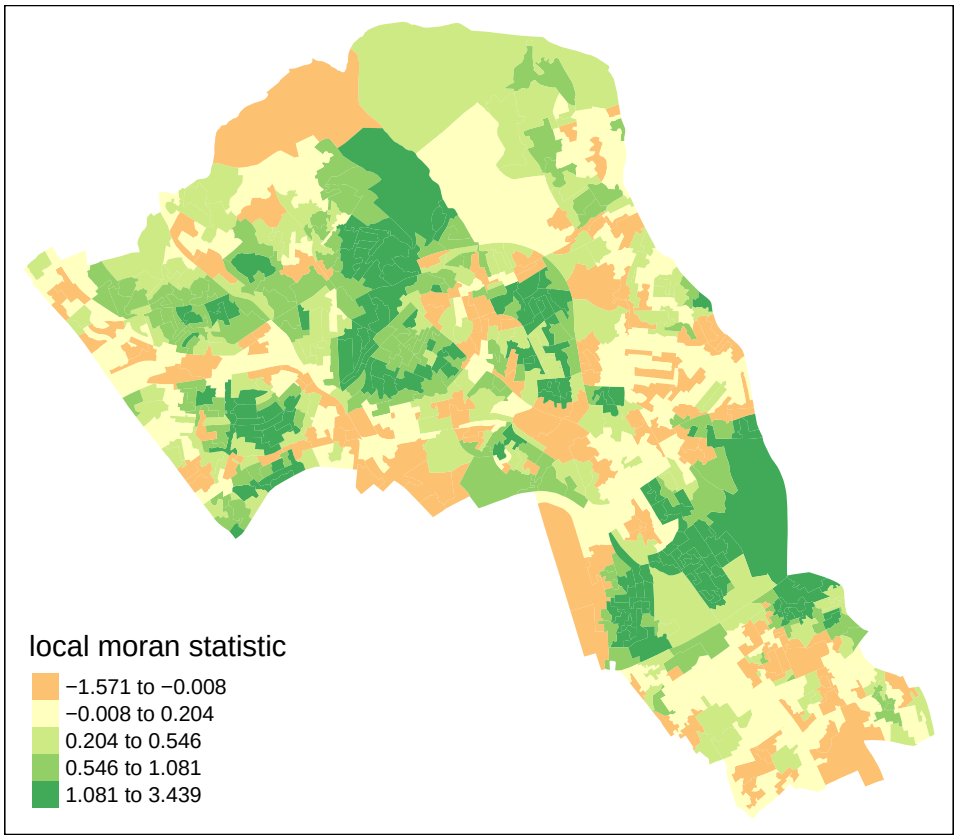
```
# creates a local moran output
local <- localmoran(x = OA.Census$Qualification,
                    listw = nb2listw(neighbours2, style = "W"))
```

By considering the help page for the localmoran function (run `?localmoran` in R) we can observe the arguments and outputs. We get a number of useful statistics from the model which are as defined:

Name	Description
Ii	local moran statistic
E.Ii	expectation of local moran statistic
Var.Ii	variance of local moran statistic
Z.Ii	standard deviate of local moran statistic
Pr()	p-value of local moran statistic

First we will map the local moran statistic (Ii). A positive value for Ii indicates that the unit is surrounded by units with similar values.

```
# binds results to our polygon shapefile
moran.map <- cbind(OA.Census, local)


# maps the results
tm_shape(moran.map) + tm_fill(col = "Ii", style = "quantile",
                              title = "local moran statistic")
```



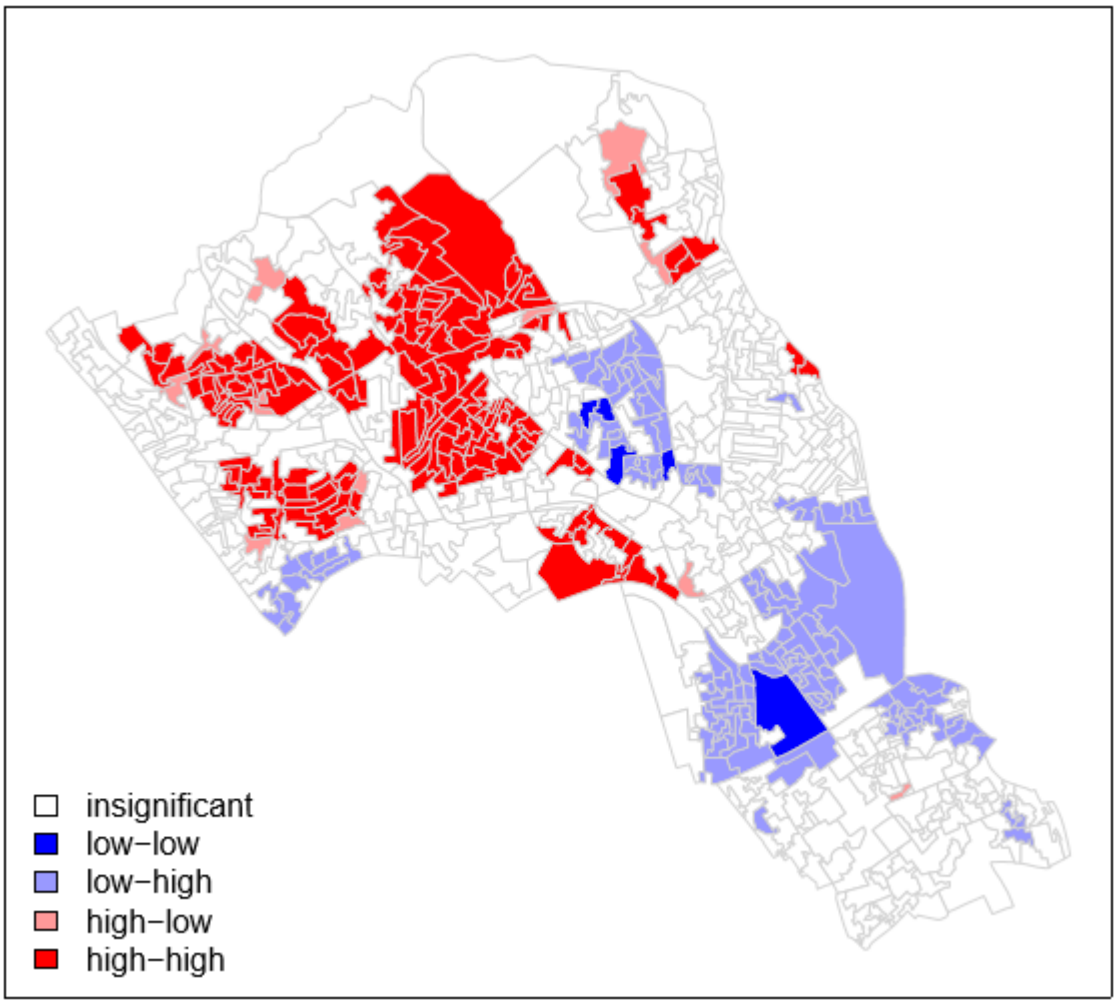
From the map it is possible to observe the variations in autocorrelation across space. We can interpret that there seems to be a geographic pattern to the autocorrelation. However, it is not possible to understand if these are clusters of high or low values.

Why not try to make a map of the P-value to observe variances in significance across Camden? Use `names(moran.map@data)` to find the column headers.

One thing we could try to do is to create a map which labels the features based on the types of relationships they share with their neighbours (i.e. high and high, low and low, insigificant, etc. . .). The following code will run this for you. *Source: Brunsdon and Comber (2015)*

```
### to create LISA cluster map ###
quadrant <- vector(mode="numeric",length=nrow(local))

# centers the variable of interest around its mean
m.qualification <- OA.Census$Qualification - mean(OA.Census$Qualification)

# centers the local Moran's around the mean
m.local <- local[,1] - mean(local[,1])

# significance threshold
signif <- 0.1

# builds a data quadrant
quadrant[m.qualification >0 & m.local>0] <- 4
quadrant[m.qualification <0 & m.local<0] <- 1
quadrant[m.qualification <0 & m.local>0] <- 2
quadrant[m.qualification >0 & m.local<0] <- 3
```

```
quadrant[local[,5]>signif] <- 0

# plot in r
brks <- c(0,1,2,3,4)
colors <- c("white","blue",rgb(0,0,1,alpha=0.4),rgb(1,0,0,alpha=0.4),"red")
plot(OA.Census,border="lightgray",col=colors[findInterval(quadrant,brks,all.inside=FALSE)])
box()
legend("bottomleft",legend=c("insignificant","low-low","low-high","high-low","high-high"),
       fill=colors,bty="n")
```



It is apparent that there is a statistically significant geographic pattern to the clustering of our qualification variable in Camden.

Getis-Ord

Another approach we can take is hot-spot analysis. The Getis-Ord Gi Statistic looks at neighbours within a defined proximity to identify where either high or low values cluster spatially. Here statistically significant hot-spots are recognised as areas of high values where other areas within a neighbourhood range also share high values too.

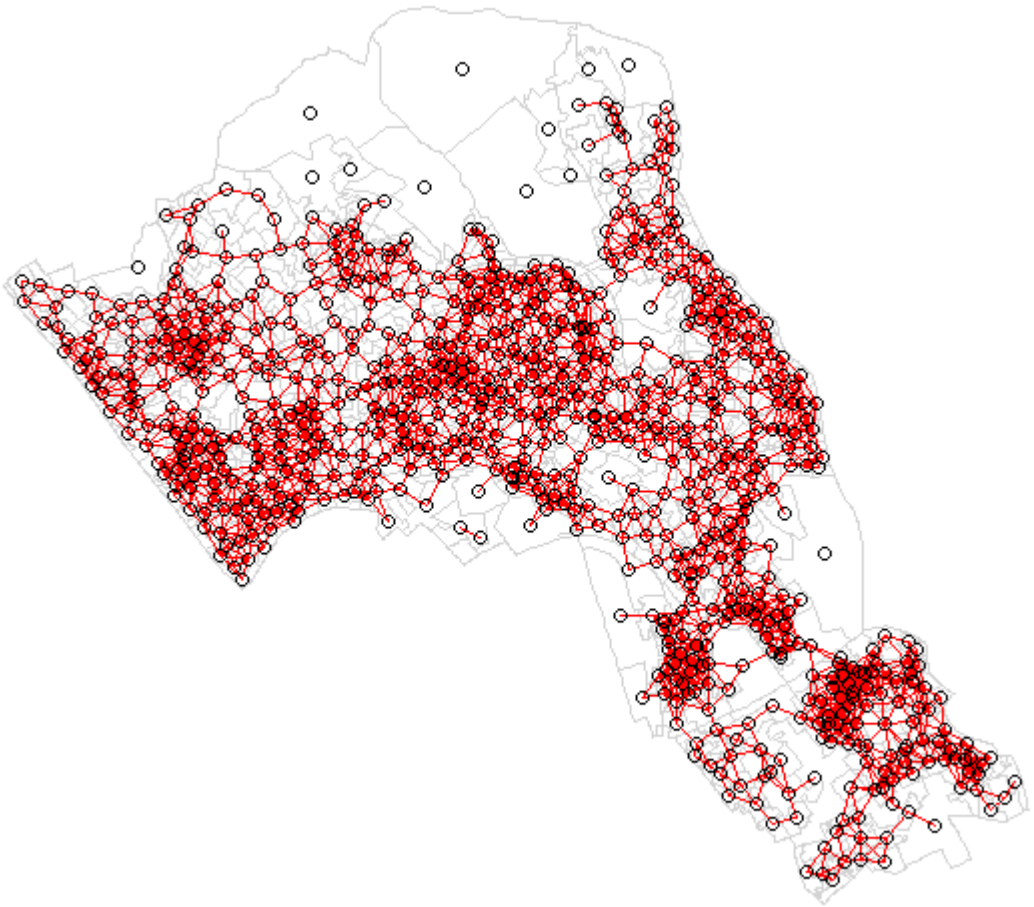
First, we need to define a new set of neighbours. Whilst the spatial autocorrection considered units which

shared borders, for Getis-Ord we are defining neighbours based on proximity.

```
# creates centroid and joins neighbours within 0 and x units
nb <- dnearneigh(coordinates(OA.Census),0,800)
# creates listw
nb_lw <- nb2listw(nb, style = 'B')
```

```
# plot the data and neighbours
plot(OA.Census, border = 'lightgrey')
plot(nb, coordinates(OA.Census), add=TRUE, col = 'red')
```

Note in this example map below we only had a search radius of 250 metres to demonstrate the function. However, it means that some areas do not have any defined nearest neighbours so we will need to search 800 meters or more for our model in Camden.

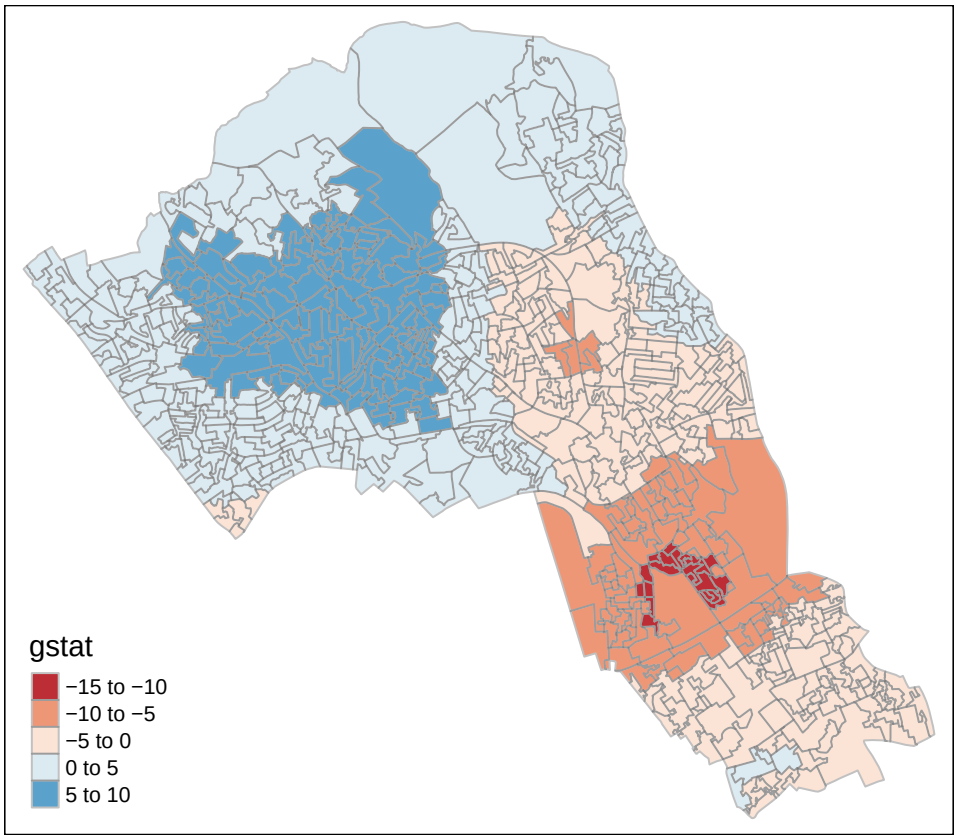


With a set of neighbourhoods estabilished we can now run the test and bind the results to our polygon file.

On some machines the `cbind` may not work with a spatial data file, in this case you will need to change *OA.Census* to *OA.Census@data* so that R knows which part of the spatial data file to join. If you take this approach the subsequent column ordering may be different to what is shown in the example below.

```
# compute Getis-Ord Gi statistic
local_g <- localG(OA.Census$Qualification, nb_lw)
local_g <- cbind(OA.Census, as.matrix(local_g))
```

```
names(local_g)[6] <- "gstat"

# map the results
tm_shape(local_g) + tm_fill("gstat", palette = "RdBu", style = "pretty") +
 tm_borders(alpha=.4)
```



The Gi Statistic is represented as a Z-score. Greater values represent a greater intensity of clustering and the direction (positive or negative) indicates high or low clusters. The final map should indicate the location of hot-spots across Camden. Repeat this for another variable.

Practical 10: Geographically Weighted Regression in R

An Introduction to Spatial Data Analysis and Visualisation in R - Guy Lansley & James Cheshire (2016)

This practical will teach you how to run a Geographically Weighted Regression (GWR). GWR is a multivariate model which can indicate where non-stationarity may take place across space; it can be used to identify how locally weighted regression coefficients may vary across the study area. We will first explore the residuals of a linear model to understand its limitations. Next, we will run a GWR and observe its parameters across space. Data for the practical can be downloaded from the **Introduction to Spatial Data Analysis and Visualisation in R** homepage.

In this tutorial we will:

- Run a linear model to predict the occurrence of a variable across small areas
- Run a geographically weighted regression to understand how models may vary across space
- Print multiple maps in one graphic using gridExtra() with tmap

First, we must set the working directory and load the practical data.

```
# Set the working directory
setwd("C:/Users/Guy/Documents/Teaching/CDRC/Practicals")

# Load the data. You may need to alter the file directory
Census.Data <-read.csv("practical_data.csv")
```

We will also need to load the spatial data files from the previous practicals.

```
# load the spatial libraries
library("sp")
library("rgdal")
library("rgeos")
library("tmap")

# Load the output area shapefiles
Output.Areas <- readOGR(".", "Camden_oa11")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "Camden_oa11"
## with 749 features
## It has 1 fields
```

```
# join our census data to the shapefile
OA.Census <- merge(Output.Areas, Census.Data, by.x="OA11CD", by.y="OA")
```

Run a linear model

First, we will run a linear model to understand the global relationship between our variables in our study area. In this case, the percentage of people with qualifications is our dependent variable, and the percentages of unemployed economically active adults and White British ethnicity are our two independent (or predictor) variables.

```
# runs a linear model
model <- lm(OA.Census$Qualification ~ OA.Census$Unemployed+OA.Census$White_British)

summary(model)
```

```
##
## Call:
## lm(formula = OA.Census$Qualification ~ OA.Census$Unemployed +
##     OA.Census$White_British)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -50.311  -8.014   1.006   8.958  38.046
##
## Coefficients:
##                          Estimate Std. Error t value Pr(>|t|)
## (Intercept)              47.86697    2.33574   20.49   <2e-16 ***
## OA.Census$Unemployed     -3.29459    0.19027  -17.32   <2e-16 ***
## OA.Census$White_British   0.41092    0.04032   10.19   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 12.69 on 746 degrees of freedom
## Multiple R-squared:  0.4645, Adjusted R-squared:  0.463
## F-statistic: 323.5 on 2 and 746 DF,  p-value: < 2.2e-16
```

As we saw in Practical 4, there is lots of interesting information we can derive from a linear regression model output. This model has an adjusted R-squared value of 0.463. So we can assume 46% of the variance can be explained by the model. We can also observe the influences of each of the variables. However, the overall fit of the model and each of the coefficients may vary across space if we consider different parts of our study area. It is therefore worth considering the standardised residuals from the model to help us understand and improve our future models.

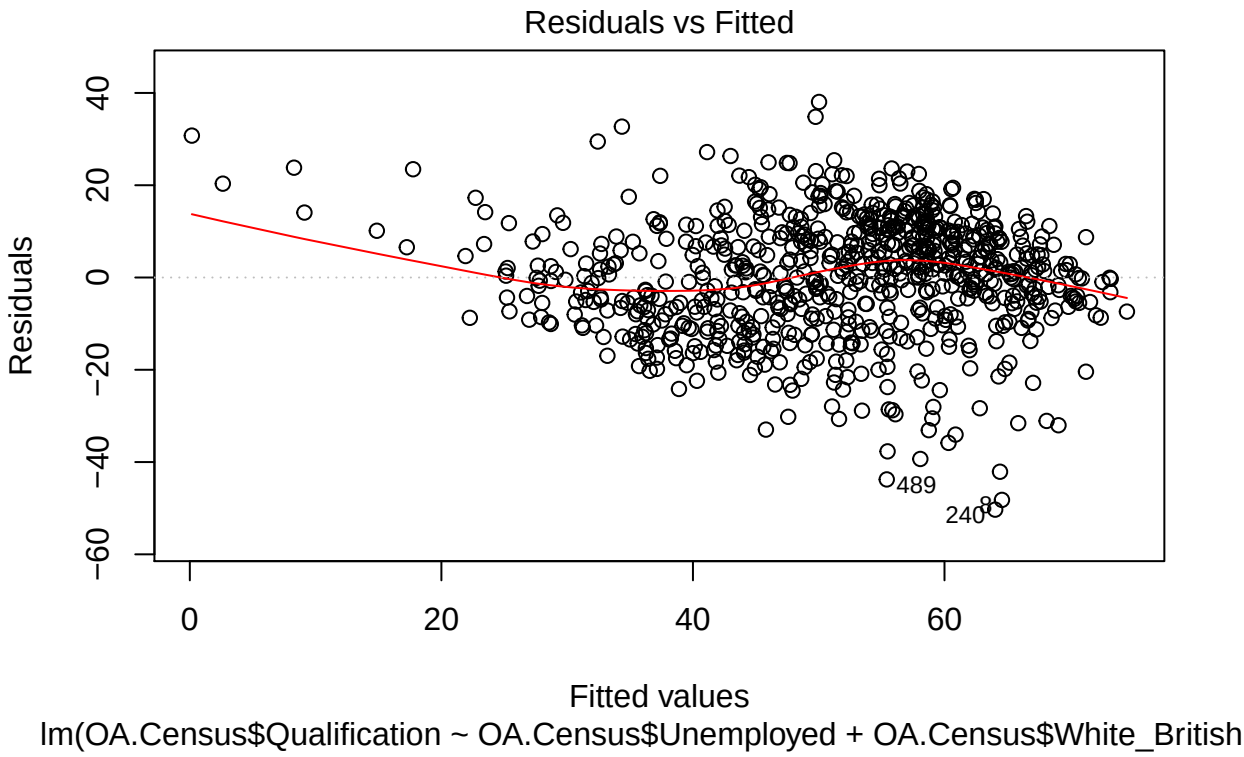
A residual is the difference between the predicted and observed values for an observation in the model. Models with lower r-squared values would have greater residuals on average as the data would not fit the modelled regression line as well. Standardised residuals are represented as Z-scores where 0 represent the predicted values.

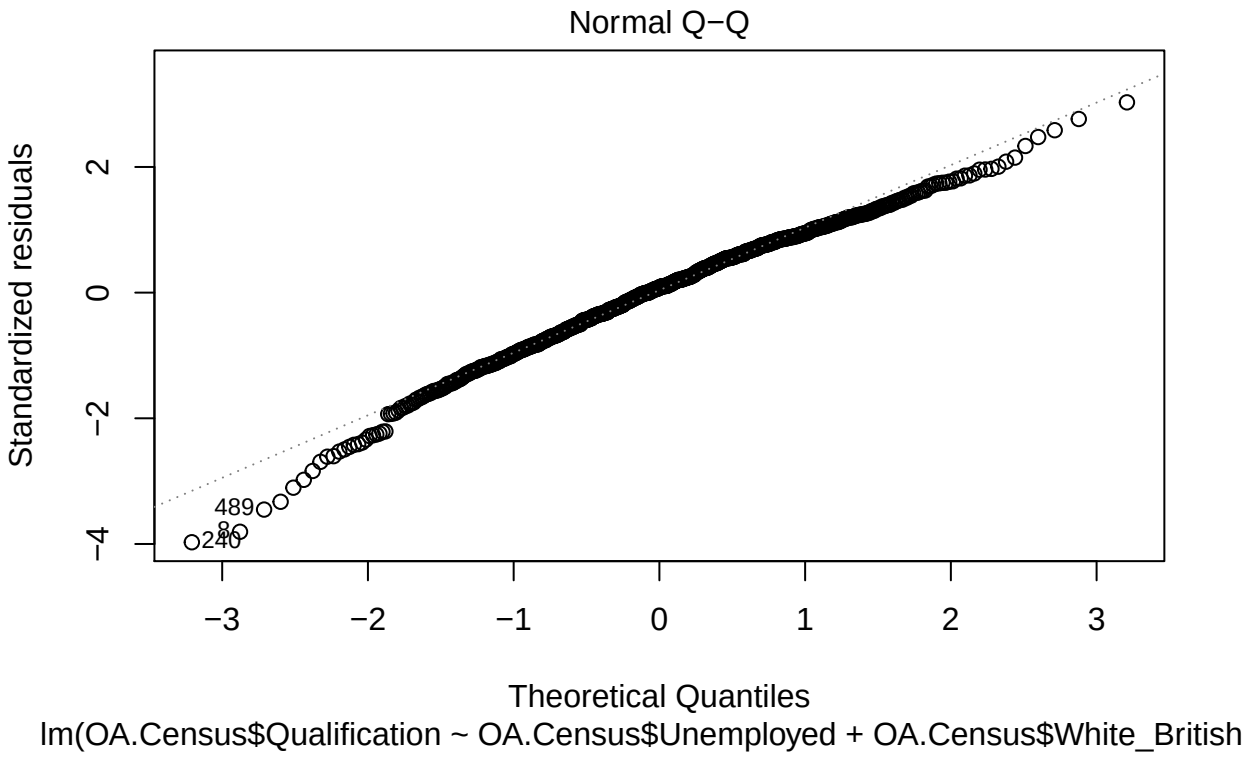
If you plot a linear model (i.e. our *model* object), R will print out four different plots of which are useful for evaluating the model fit. These are very briefly summarised as:

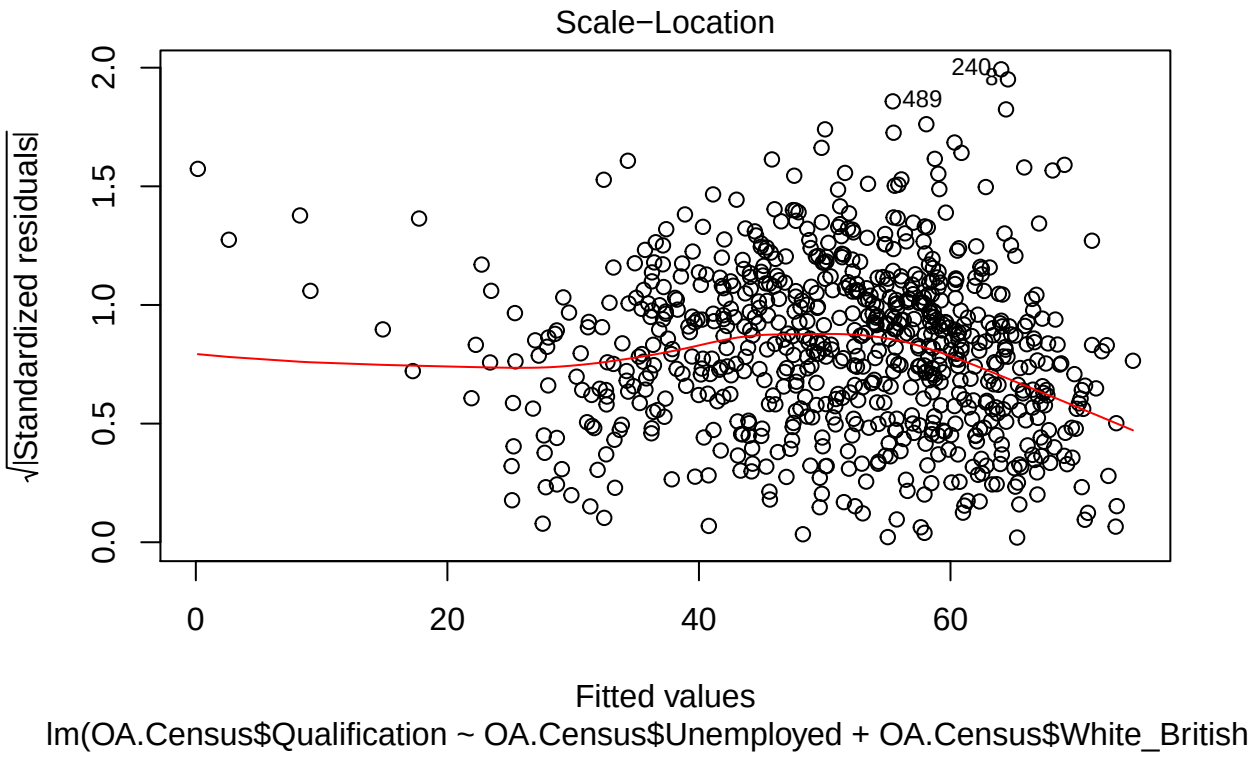
- **Residuals vs Fitted** - considers the relationship between the actual and the predicted data. The more dispersed the residuals are, the weaker the R2 should be. This can be useful to identify outliers too.The fit also tells us if the residuals are non-linearly distributed.
- **Normal Q-Q** - Demonstrates the extent to which the residuals are normally distributed. Normal residuals should fit the straight line well.
- **Scale-Location** - Shows if the residuals are spread equally across the full range of the predictors. If the values in this chart display a linear positive relationship, it suggests that the residuals spread wider and wider for greater values (this is known as heteroscedasticity).
- **Residuals vs Leverage** - this graph identifies outliers, high-leverage points and influential observations. This plot is pretty difficult to interpret and there are other means of identifying these values.

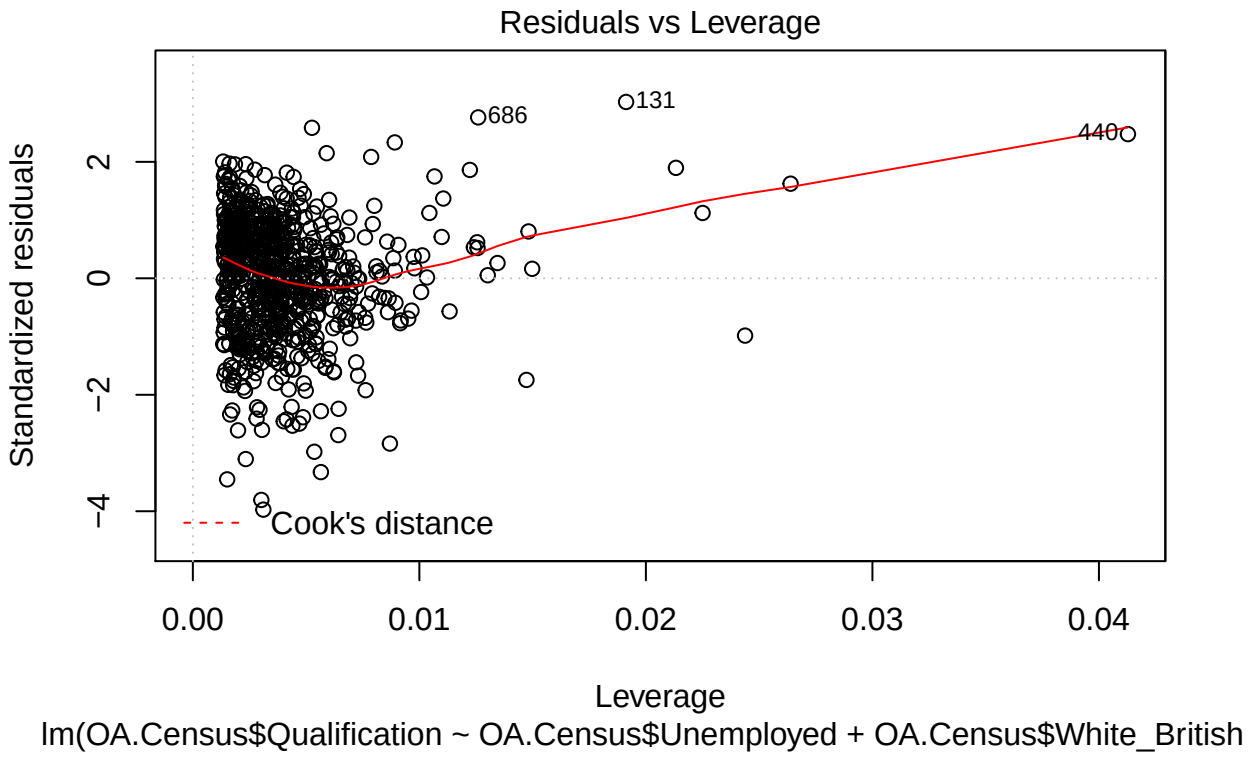
A good description of these plots and how to interpret them can be found here

```
# this will plot 4 scatter plots from the linear model
plot(model)
```

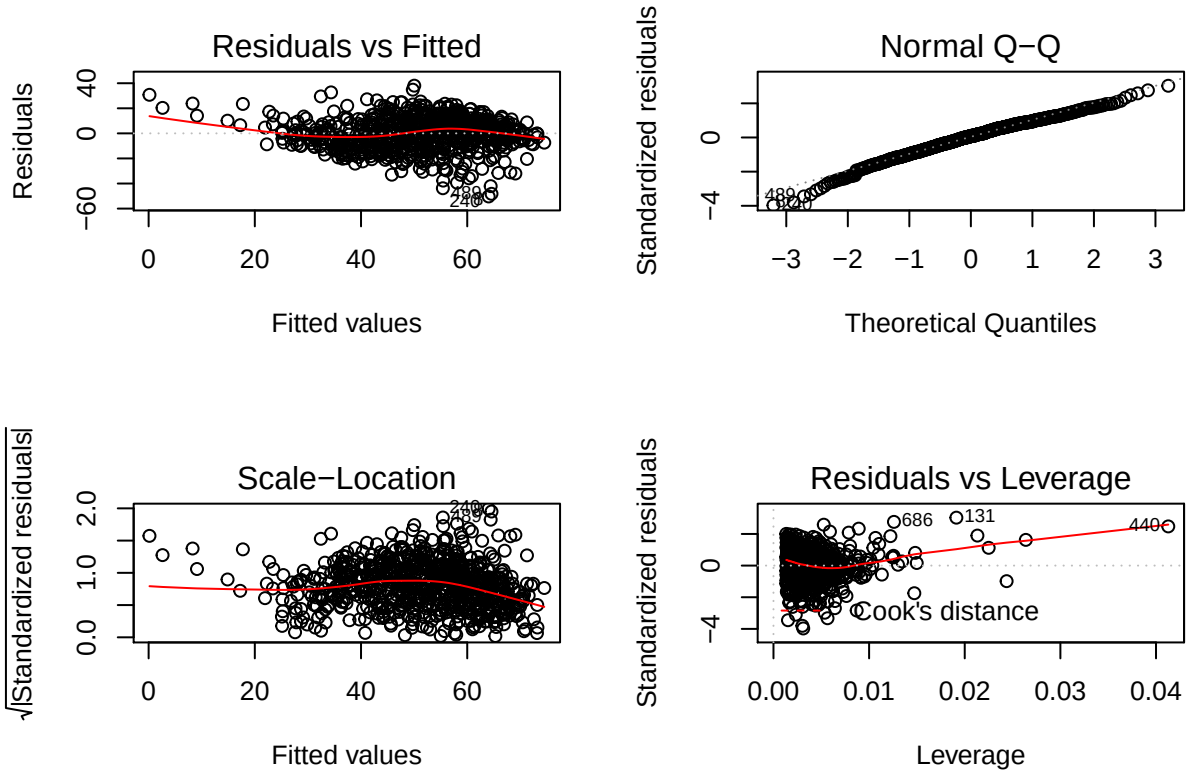

Residuals vs Fitted
Residuals
40
20
0
−20
−40
−60
0
20
40
60
489
240
Fitted values
lm(OA.Census$Qualification ~ OA.Census$Unemployed + OA.Census$White_British


Normal Q−Q
Standardized residuals
Theoretical Quantiles
lm(OA.Census$Qualification ~ OA.Census$Unemployed + OA.Census$White_British
489
86
240
−4
−2
0
2
−3
−2
−1
0
1
2
3


Scale−Location
√|Standardized residuals|
0.0
0.5
1.0
1.5
2.0
240
489
0
20
40
60
Fitted values
lm(OA.Census$Qualification ~ OA.Census$Unemployed + OA.Census$White_British




```
# we can use the par function if we want to plot them in a 2x2 frame
par(mfrow=c(2,2))
plot(model)
```

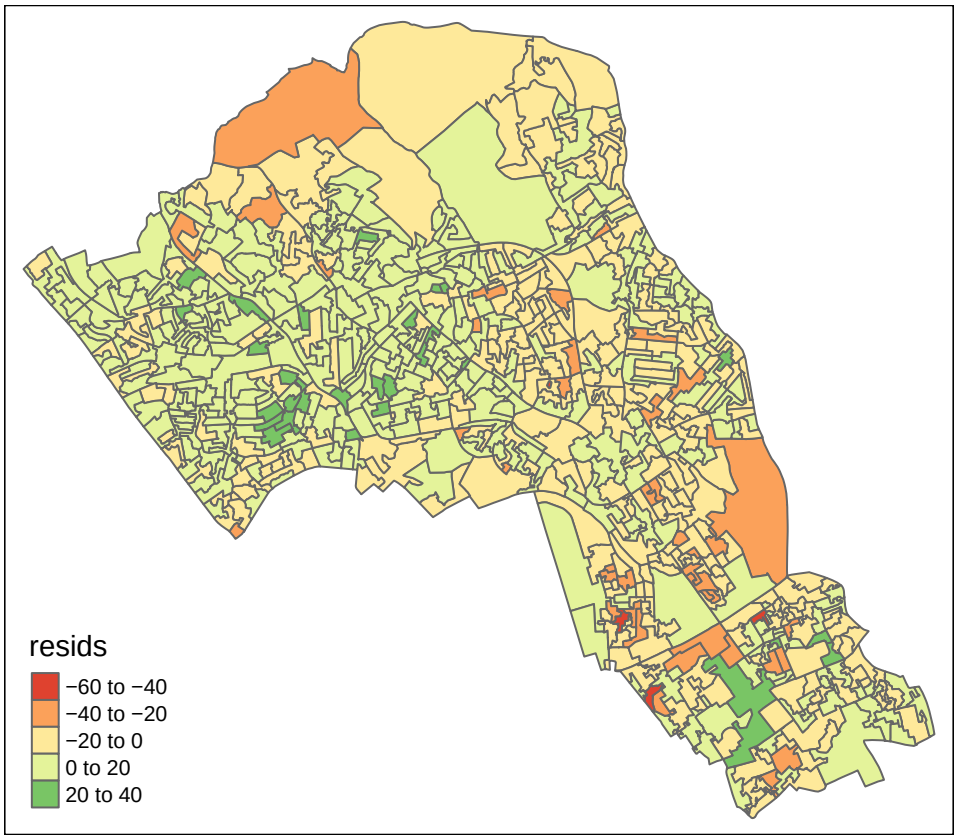


If you want to print just one of the plots you can enter *which* = *n* within the `plot()` function. i.e. `plot(model, which = 3)`

Mapping the resididuals

We can also map the residuals to see if there is a spatial distribution of them across Camden.

```
resids<-residuals(model)

map.resids <- cbind(OA.Census, resids)
# we need to rename the column header from the resids file
# in this case its the 6th column of map.resids
names(map.resids)[6] <- "resids"

# maps the residuals using the quickmap function from tmap
qtm(map.resids, fill = "resids")
```



If you notice a geographic pattern to the residiuals, it is possible that an unobserved variable may also be influencing our dependent variable in the model (% with a qualification).

Run a GWR

> GWR is the term introduced by Fotheringham, Charlton and Brunsdon (1997, 2002) to describe a family of regression models in which the coefficients are allowed to vary spatially. GWR uses the coordinates of each sample point or zone centroid, ti, as a target point for a form of spatially weighted least squares regression (for some models the target points can be separately defined, e.g. as grid intersection points, rather than observed data points). (de Smith *et al*, 2015)

Prior to running the GWR model we need to calculate a kernel bandwidth. This will determine now the GWR subsets the data when its test multiple models across space.

```
library("spgwr")

#calculate kernel bandwidth
GWRbandwidth <- gwr.sel(OA.Census$Qualification ~ OA.Census$Unemployed +
                          OA.Census$White_British, data=OA.Census, adapt =TRUE)
```

```
## Adaptive q: 0.381966 CV score: 101420.8
## Adaptive q: 0.618034 CV score: 109723.2
## Adaptive q: 0.236068 CV score: 96876.06
## Adaptive q: 0.145898 CV score: 94192.41
## Adaptive q: 0.09016994 CV score: 91099.75
## Adaptive q: 0.05572809 CV score: 88242.89
## Adaptive q: 0.03444185 CV score: 85633.41
```

```
## Adaptive q: 0.02128624 CV score: 83790.04
## Adaptive q: 0.01315562 CV score: 83096.03
## Adaptive q: 0.008130619 CV score: 84177.45
## Adaptive q: 0.01535288 CV score: 83014.34
## Adaptive q: 0.01515437 CV score: 82957.49
## Adaptive q: 0.01436908 CV score: 82857.74
## Adaptive q: 0.01440977 CV score: 82852.4
## Adaptive q: 0.01457859 CV score: 82833.25
## Adaptive q: 0.01479852 CV score: 82855.45
## Adaptive q: 0.01461928 CV score: 82829.32
## Adaptive q: 0.01468774 CV score: 82823.82
## Adaptive q: 0.01473006 CV score: 82835.89
## Adaptive q: 0.01468774 CV score: 82823.82
```

Next we can run the model and view the results.

```
#run the gwr model
gwr.model = gwr(OA.Census$Qualification ~ OA.Census$Unemployed+OA.Census$White_British,
                data = OA.Census, adapt=GWRbandwidth, hatmatrix=TRUE, se.fit=TRUE)

#print the results of the model
gwr.model
```

```
## Call:
## gwr(formula = OA.Census$Qualification ~ OA.Census$Unemployed +
##     OA.Census$White_British, data = OA.Census, adapt = GWRbandwidth,
##     hatmatrix = TRUE, se.fit = TRUE)
## Kernel function: gwr.Gauss
## Adaptive quantile: 0.01468774 (about 11 of 749 data points)
## Summary of GWR coefficient estimates at data points:
##                             Min. 1st Qu.  Median 3rd Qu.    Max.  Global
## X.Intercept.            11.0800 34.4300 45.7700 59.7500 85.0200 47.8670
## OA.Census.Unemployed    -5.4530 -3.2830 -2.5540 -1.7940  0.7702 -3.2946
## OA.Census.White_British -0.2805  0.1995  0.3779  0.5322  0.9468  0.4109
## Number of data points: 749
## Effective number of parameters (residual: 2traceS - traceS'S): 132.6449
## Effective degrees of freedom (residual: 2traceS - traceS'S): 616.3551
## Sigma (residual: 2traceS - traceS'S): 9.903539
## Effective number of parameters (model: traceS): 94.44661
## Effective degrees of freedom (model: traceS): 654.5534
## Sigma (model: traceS): 9.610221
## Sigma (ML): 8.983902
## AICc (GWR p. 61, eq 2.33; p. 96, eq. 4.21): 5633.438
## AIC (GWR p. 96, eq. 4.22): 5508.777
## Residual sum of squares: 60452.16
## Quasi-global R2: 0.7303206
```

Upon first glance, much of the outputs of this model are identical to the outputs of the linear model. However we can explore the coefficients of this model across each area unit.

We create a results output from the model which contains a number of attributes which correspond with each unique output area from our OA.Census file. In the example below we have printed the names of each of the new variables.

They include a local R2 value, the predicted values (for % qualifications) and local coefficients for each variable. We will then bind the outputs to our OA.Census polygon so we can map them.

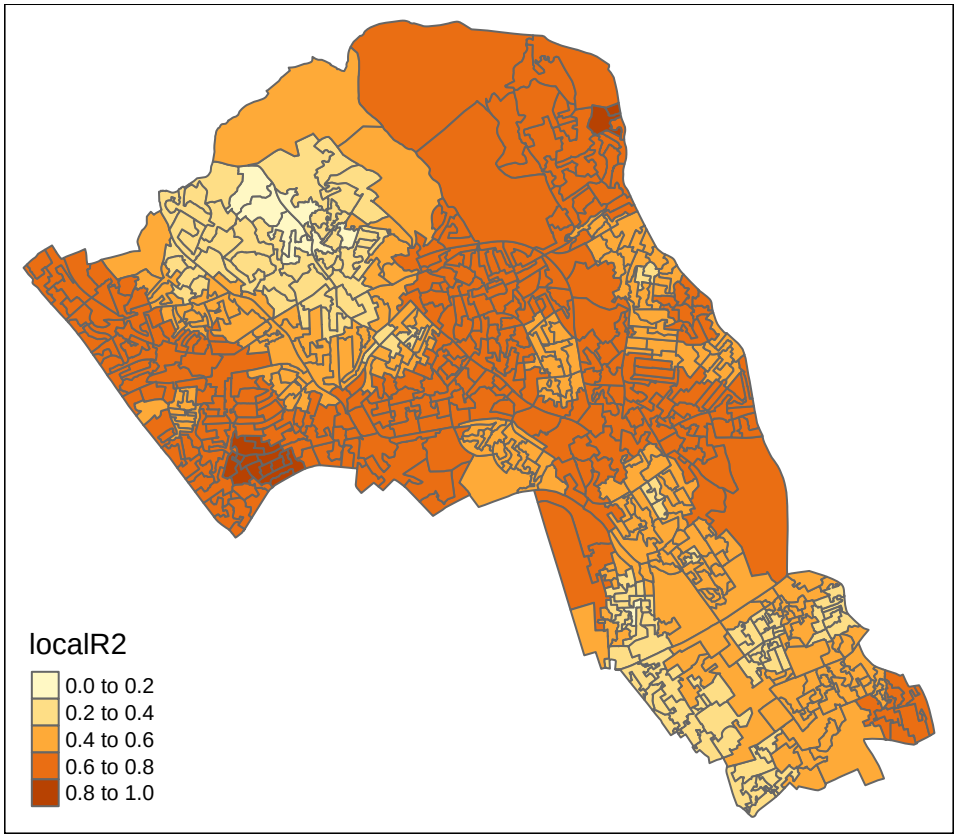
```
results <-as.data.frame(gwr.model$SDF)

names(results)
```

```
##  [1] "sum.w"                          "X.Intercept."
##  [3] "OA.Census.Unemployed"           "OA.Census.White_British"
##  [5] "X.Intercept._se"                "OA.Census.Unemployed_se"
##  [7] "OA.Census.White_British_se"     "gwr.e"
##  [9] "pred"                           "pred.se"
## [11] "localR2"                        "X.Intercept._se_EDF"
## [13] "OA.Census.Unemployed_se_EDF"    "OA.Census.White_British_se_EDF"
## [15] "pred.se_EDF"
```

```
gwr.map <- cbind(OA.Census, as.matrix(results))
```

The variable names followed by the name of our original dataframe (i.e. OA.Census.Unemployed) are the coefficients of the model.

```
qtm(gwr.map, fill = "localR2")
```



Using gridExtra

We will now consider some of the other outputs. We will create four maps in one image to show the original distributions of our unemployed and White British variables, and their coefficients in the GWR model.

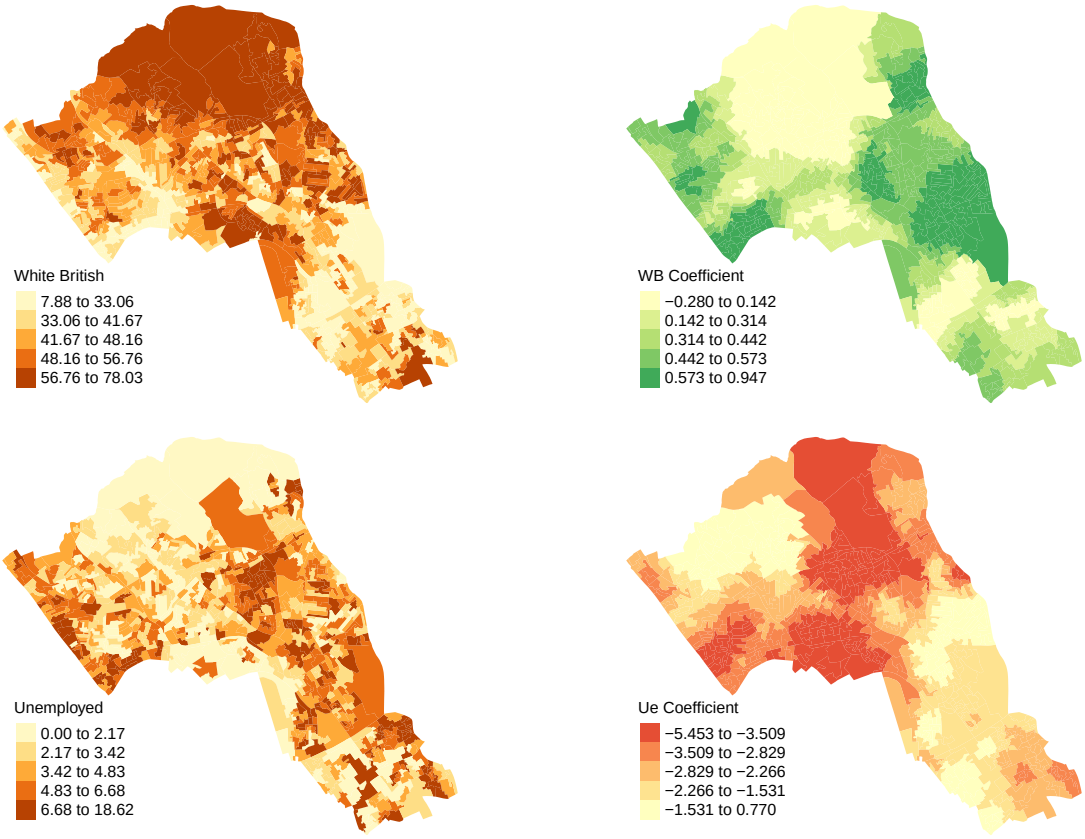
To facet four maps in tmap we can use functions from the grid and gridExtra packages which allow us to split the output window into segments. We will divide the output into four and print a map in each window.

Firstly, we will create four map objects using tmap. Instead of printing them directly as we have done usually, we all assign each map an object ID so it can be called later.

```
# create tmap objects
map1 <- tm_shape(gwr.map) + tm_fill("White_British", n = 5, style = "quantile",
                                    title = "White British") +
  tm_layout(frame = FALSE, legend.text.size = 0.5, legend.title.size = 0.6)

map2 <- tm_shape(gwr.map) + tm_fill("OA.Census.White_British", n = 5, style = "quantile",
                                    title = "WB Coefficient") +
  tm_layout(frame = FALSE, legend.text.size = 0.5, legend.title.size = 0.6)

map3 <- tm_shape(gwr.map) + tm_fill("Unemployed", n = 5, style = "quantile",
                                    title = "Unemployed") +
  tm_layout(frame = FALSE, legend.text.size = 0.5, legend.title.size = 0.6)

map4 <- tm_shape(gwr.map) + tm_fill("OA.Census.Unemployed", n = 5, style = "quantile",
                                    title = "Ue Coefficient") +
  tm_layout(frame = FALSE, legend.text.size = 0.5, legend.title.size = 0.6)
```

With the four maps ready to be printed, we will now create a grid to print them into. From now on everytime we wish to recreate the maps we will need to run the grid.newpage() function to clear the existing grid window.

```
library(grid)
library(gridExtra)
# creates a clear grid
grid.newpage()
# assigns the cell size of the grid, in this case 2 by 2
pushViewport(viewport(layout=grid.layout(2,2)))

# prints a map object into a defined cell
print(map1, vp=viewport(layout.pos.col = 1, layout.pos.row =1))
print(map2, vp=viewport(layout.pos.col = 2, layout.pos.row =1))
print(map3, vp=viewport(layout.pos.col = 1, layout.pos.row =2))
print(map4, vp=viewport(layout.pos.col = 2, layout.pos.row =2))
```


White British
7.88 to 33.06
33.06 to 41.67
41.67 to 48.16
48.16 to 56.76
56.76 to 78.03
WB Coefficient
−0.280 to 0.142
0.142 to 0.314
0.314 to 0.442
0.442 to 0.573
0.573 to 0.947
Unemployed
0.00 to 2.17
2.17 to 3.42
3.42 to 4.83
4.83 to 6.68
6.68 to 18.62
Ue Coefficient
−5.453 to −3.509
−3.509 to −2.829
−2.829 to −2.266
−2.266 to −1.531
−1.531 to 0.770

Practical 11: Interpolating Point Data in R

This practical provides an introduction to some techniques which are useful for interpolating point data across space in R. Interpolation describes a means of estimating a value for a particular setting based on a known sequence of data. In a spatial context, it refers to using an existing distribution of data points to predict values across space. Data for the practical can be downloaded from the **Introduction to Spatial Data Analysis and Visualisation in R** homepage.

In this tutorial we will:

- Create thiessen polygons
- Run an inverse distance weighting to interpolate point data
- Clip spatial data using the crop (for polygons) and mask (for rasters) functions

First, we must set the working directory and load the practical data.

```
# Set the working directory
setwd("C:/Users/Guy/Documents/Teaching/CDRC/Practicals")
```

We will also need to load the spatial data files from the previous practicals.

```
# load the spatial libraries
library("sp")
library("rgdal")
library("rgeos")
library("tmap")

# Load the output area shapefiles, we won't join it to any data this time
Output.Areas <- readOGR(".", "Camden_oa11")
```

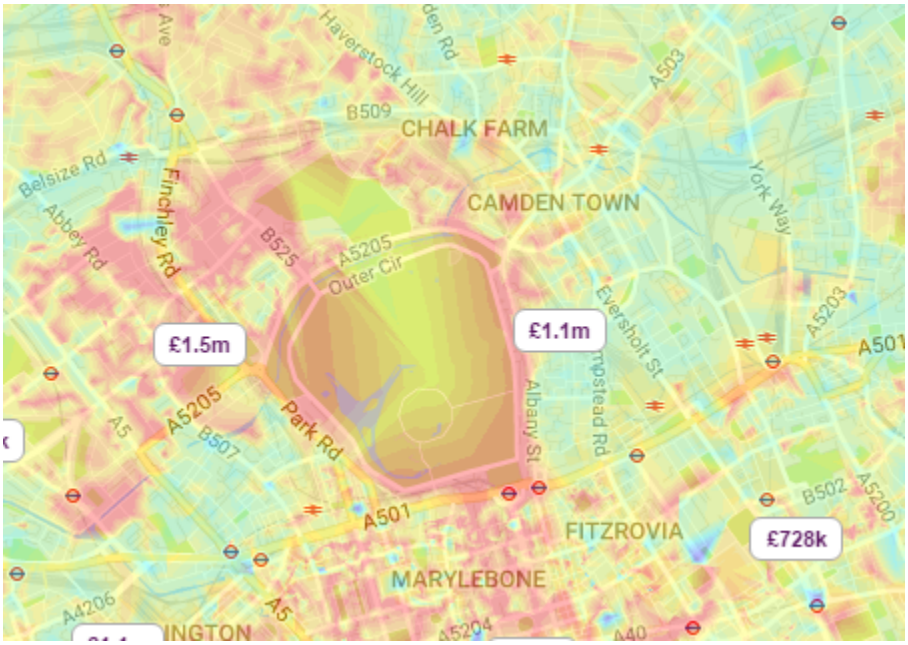
```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "Camden_oa11"
## with 749 features
## It has 1 fields
```

```
# load the houses point files
House.Points <- readOGR(".", "Camden_house_sales")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "Camden_house_sales"
## with 2547 features
## It has 4 fields
```

Why interpolate data

There are many reasons why we may wish to interpolate point data across a map. It could be because we are trying to predict a variable across space, including in areas where there are little to no data. As exemplified by house price heat maps on major property websites (see below).

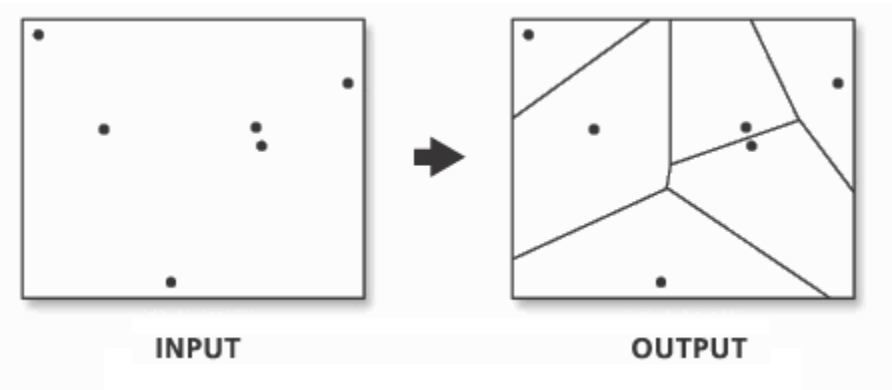


Estimating the spatial distribution of house prices across Camden and the surrounding area. Source: Zoopla.co.uk

We might also want to smooth the data across space so that we cannot interpret the results of individuals, but still identify the general trends from the data. This is particularly useful when the data corresponds to individual persons and disclosing their locations is unethical.

Thiessen polygons

The first step we can take to interpolate the data across space is to create Thiessen polygons. Thiessen polygons are formed to assign boundaries of the areas closest to each unique point. Therefore, for every point in a dataset, it has a corresponding Thiessen polygon. This is demonstrated in the diagram below.

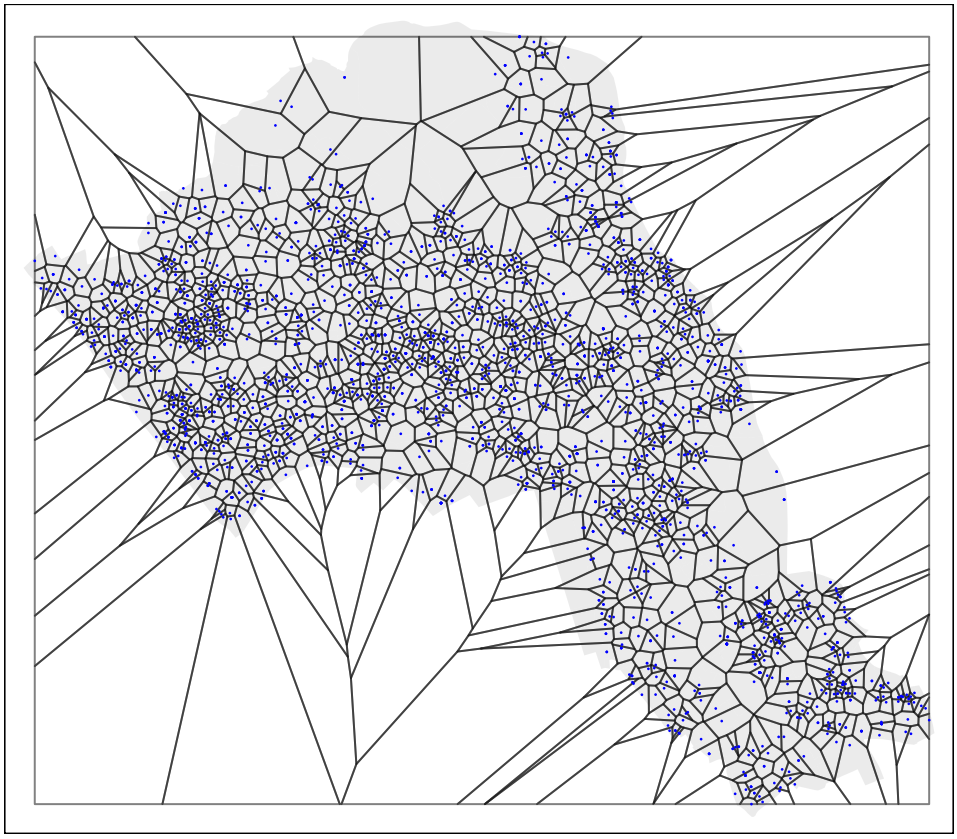


A demonstration of Thiessen polygon generation from point data. Source: resources.esri.com

For more information on Thiessen (or Voronoi) polygons please visit the Geospatial Ananalysis (de Smith *et al*, 2015) webpage on *tessellations and triangulations*

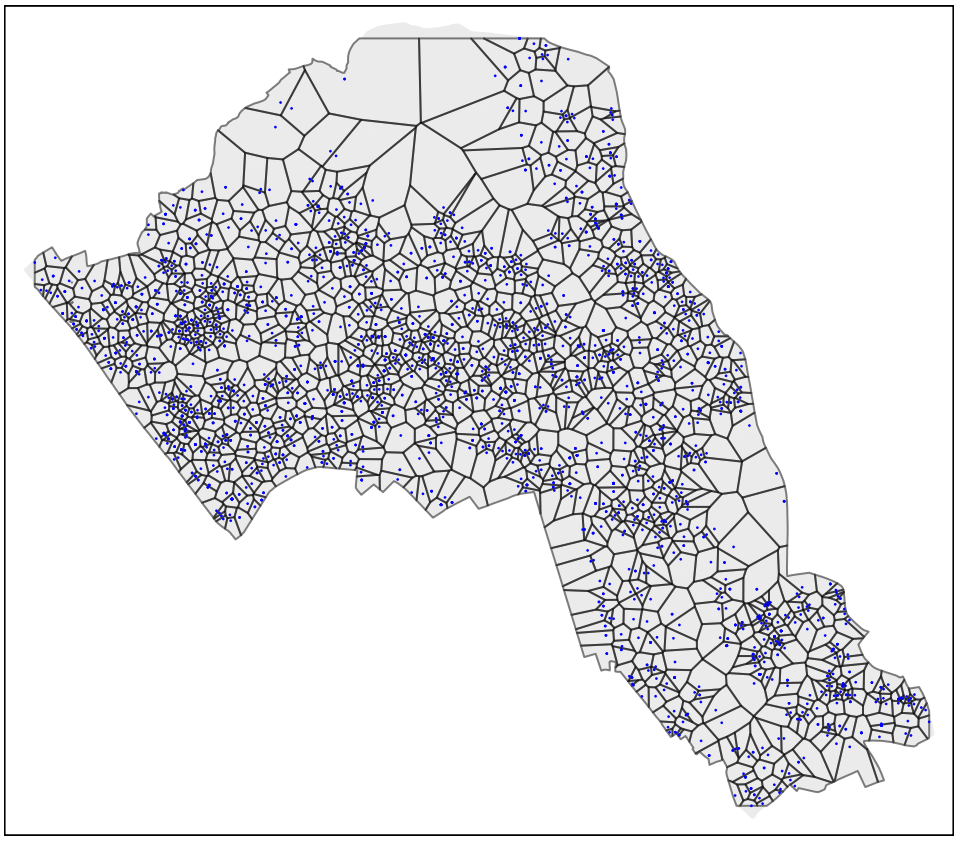
So in this exercise, we will create Thiessen polygons for our house data, then use the polygons to map the house prices of their corresponding house point. The spatstat package provides the functionality to produce Thiessen polygons via its dirichelet tesselllation of spatial point patterns function (`dirichlet()`). We also need to first convert the data to a ppp (point pattern) object class. The maptools package will enable the `as.ppp()` function.

```
library(spatstat)
library(maptools)  # Needed for conversion from SPDF to ppp

# Create a tessellated surface
dat.pp <- as(dirichlet(as.ppp(House.Points)), "SpatialPolygons")
dat.pp <- as(dat.pp,"SpatialPolygons")

# Sets the projection to British National Grid
proj4string(dat.pp) <- CRS("+init=EPSG:27700")
proj4string(House.Points) <- CRS("+init=EPSG:27700")

# Assign to each polygon the data from House.Points
int.Z <- over(dat.pp,House.Points, fn=mean)

# Create a SpatialPolygonsDataFrame
thiessen <- SpatialPolygonsDataFrame(dat.pp, int.Z)


# maps the thiessen polygons and House.Points
tm_shape(Output.Areas) + tm_fill(alpha=.3, col = "grey") +
tm_shape(thiessen) +  tm_borders(alpha=.5, col = "black") +
tm_shape(House.Points) + tm_dots(col = "blue", scale = 0.5)
```



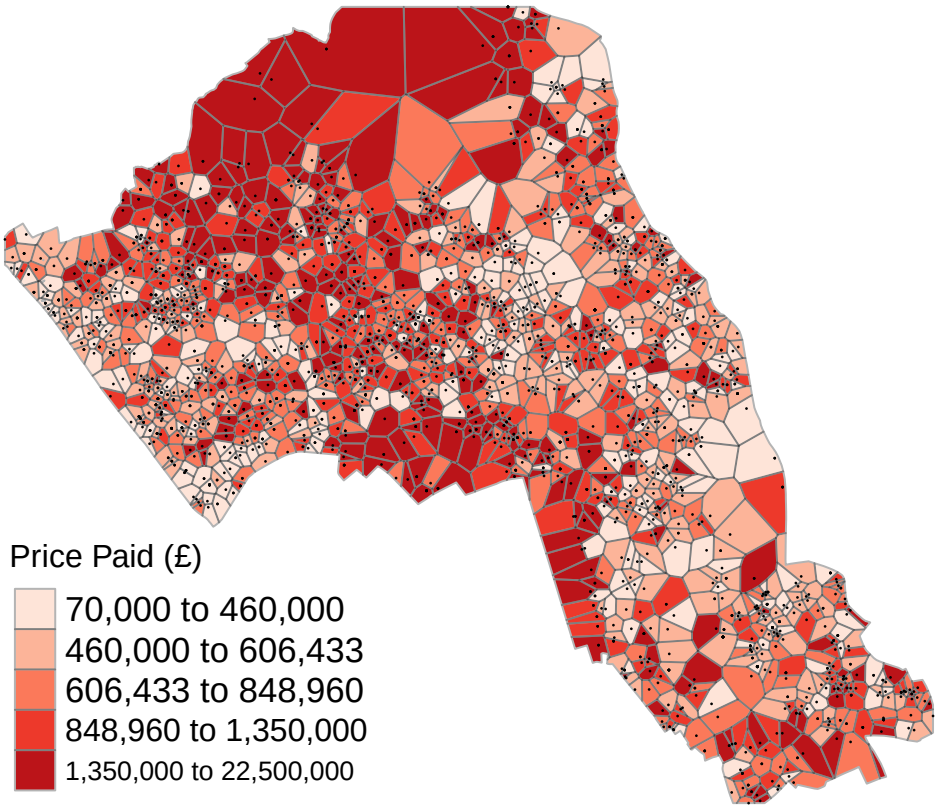
From the map you can interpret the nearest neighbourhoods for each point. We can also clip the Thiessen polygon by the Output.Areas shapefile so it only represents the borough of Camden. We will do this using the `crop()` function from the raster package.

```
library(raster)
# crops the polygon by our output area shapefile
thiessen.crop <-crop(thiessen, Output.Areas)

# maps cropped thiessen polygons and House.Points
tm_shape(Output.Areas) + tm_fill(alpha=.3, col = "grey") +
tm_shape(thiessen.crop) +  tm_borders(alpha=.5, col = "black") +
tm_shape(House.Points) + tm_dots(col = "blue", scale = 0.5)
```



We can now map our point data using the newly formed polygons.

```
# maps house prices across thiessen polygons
tm_shape(thiessen.crop) + tm_fill(col = "Price", style = "quantile", palette = "Reds",
                                  title = "Price Paid (£)") +
  tm_borders(alpha=.3, col = "black") +
tm_shape(House.Points) + tm_dots(col = "black", scale = 0.5) +
tm_layout(legend.position = c("left", "bottom"),  legend.text.size = 1.05,
          legend.title.size = 1.2, frame = FALSE)
```



Inverse Distance Weighting (IDW)

There are a range of deterministic interpolation methods which are useful for interpolating point data across two dimensions. One of the most commonly used is Inverse Distance Weighting.

An IDW is a means of converting point data of numerical values into a continuous surface to visualise how the data may be distributed across space. The technique interpolates point data by using a weighted average of a variable from nearby points to predict the value of that variable for each location (de Smith *et al*, 2015). The weighting of the points is determined by their inverse distances drawing on Tobler's first law of geography:

> "everything is related to everything else, but near things are more related than distant things"

To run an IDW we will need to run the code below which will interpolate the price variable of our *House.Points* object.

```
library(gstat)
library(xts)

# define sample grid based on the extent of the House.Points file
grid <-spsample(House.Points, type = 'regular', n = 10000)

# runs the idw for the Price variable of House.Points
idw <- idw(House.Points$Price ~ 1, House.Points, newdata= grid)
```

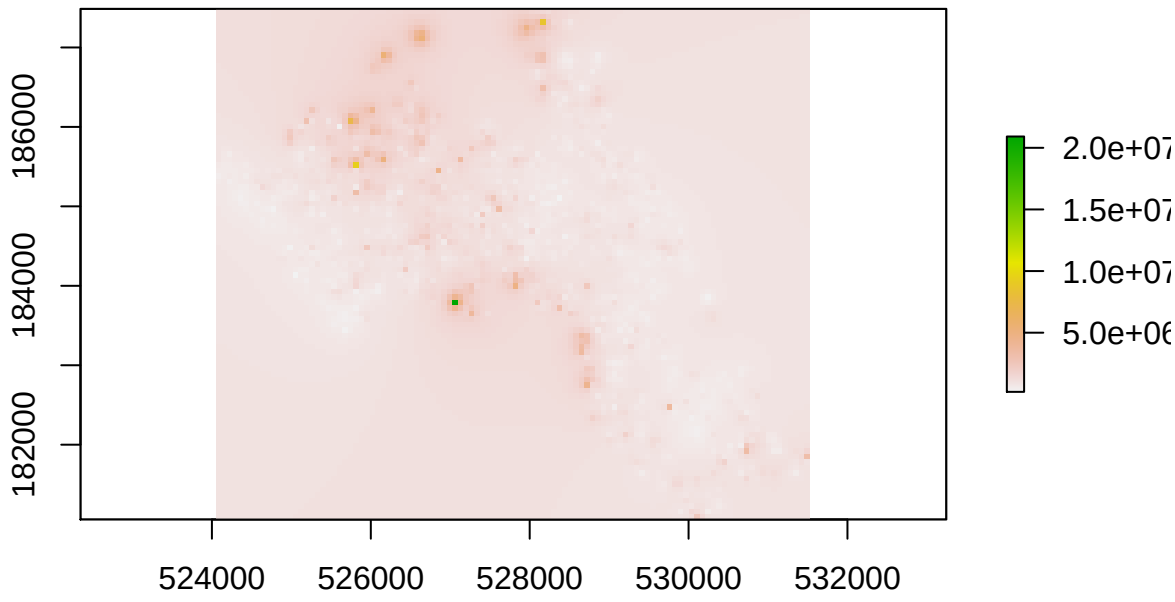
```
## [inverse distance weighted interpolation]
```

The IDW is outputted to a data type, but more needs to be done before it can be visualised. We will first transform the data into a data frame object. We can then rename the column headers.

```
idw.output = as.data.frame(idw)
names(idw.output)[1:3] <- c("long", "lat", "prediction")
```

Next we need to convert this data into a raster, the raster can then be plotted on a map along with our other spatial data.

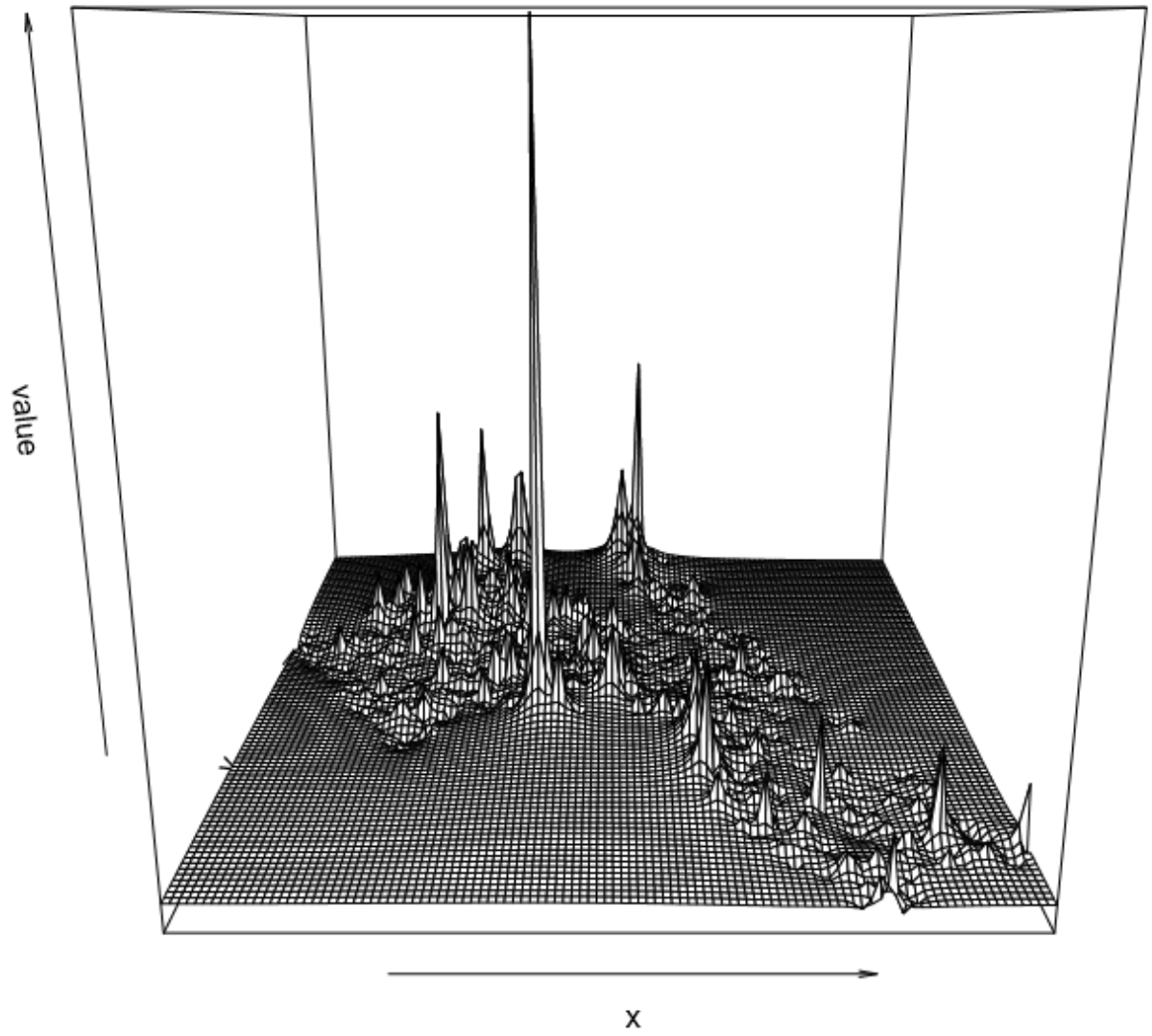
```
library(raster) # you will need to load this package if you have not done so already

# create spatial points data frame
spg <- idw.output
coordinates(spg) <- ~ long + lat

# coerce to SpatialPixelsDataFrame
gridded(spg) <- TRUE
# coerce to raster
raster_idw <- raster(spg)

# sets projection to British National Grid
projection(raster_idw) <- CRS("+init=EPSG:27700")

# we can quickly plot the raster to check its okay
plot(raster_idw)
```



3D surfaces

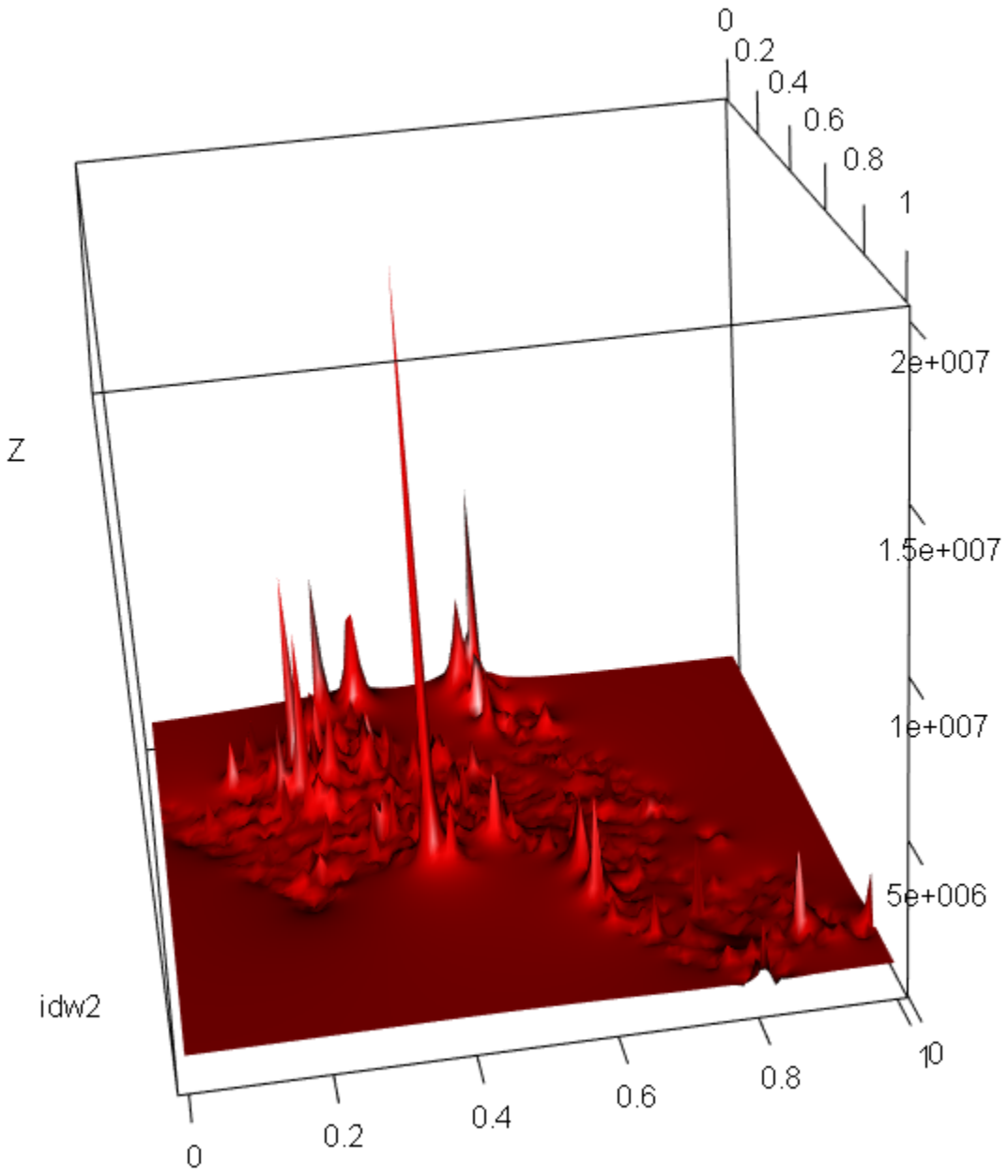
It is also possible to make a 3D plot, by very simply using the `persp()` function. Simply enter `persp(raster_idw)` into R.

```
persp(raster_idw)
```



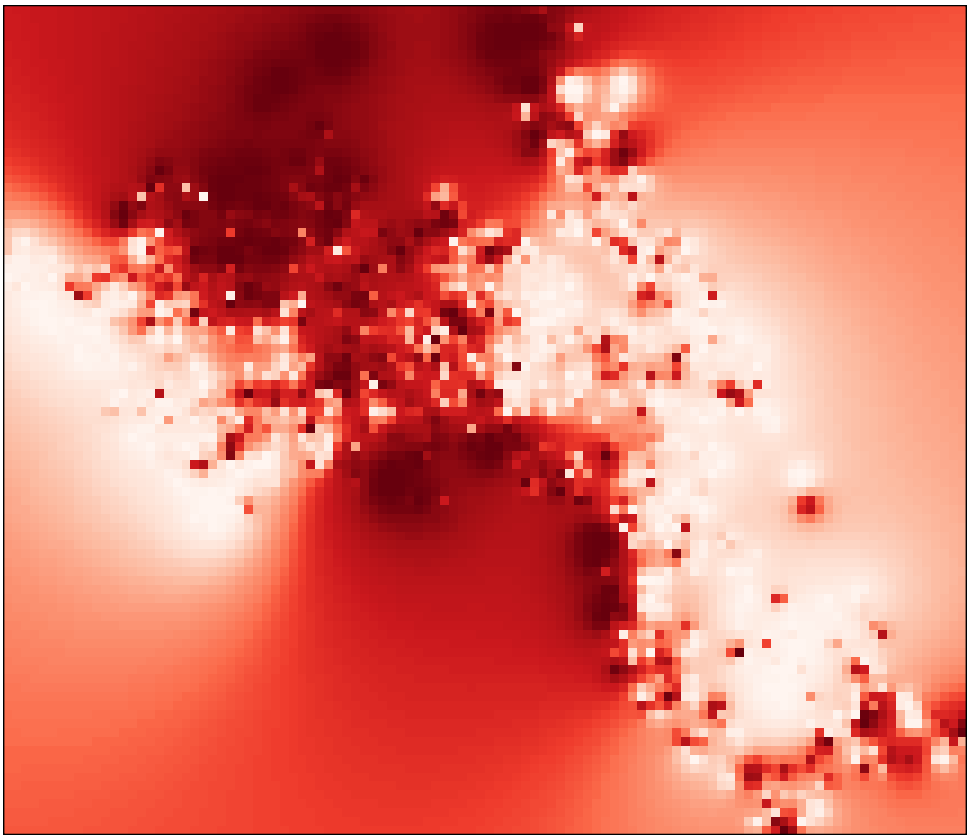
We can also create a 3D interactive chart using the rgl package and its `persp3d()` function.

```
# this package lets us create interactive 3d visualisations
library(rgl)

# we need to first convert the raster to a matrix object type
idw2 <- as.matrix(raster_idw)

persp3d(idw2, col = "red")
```



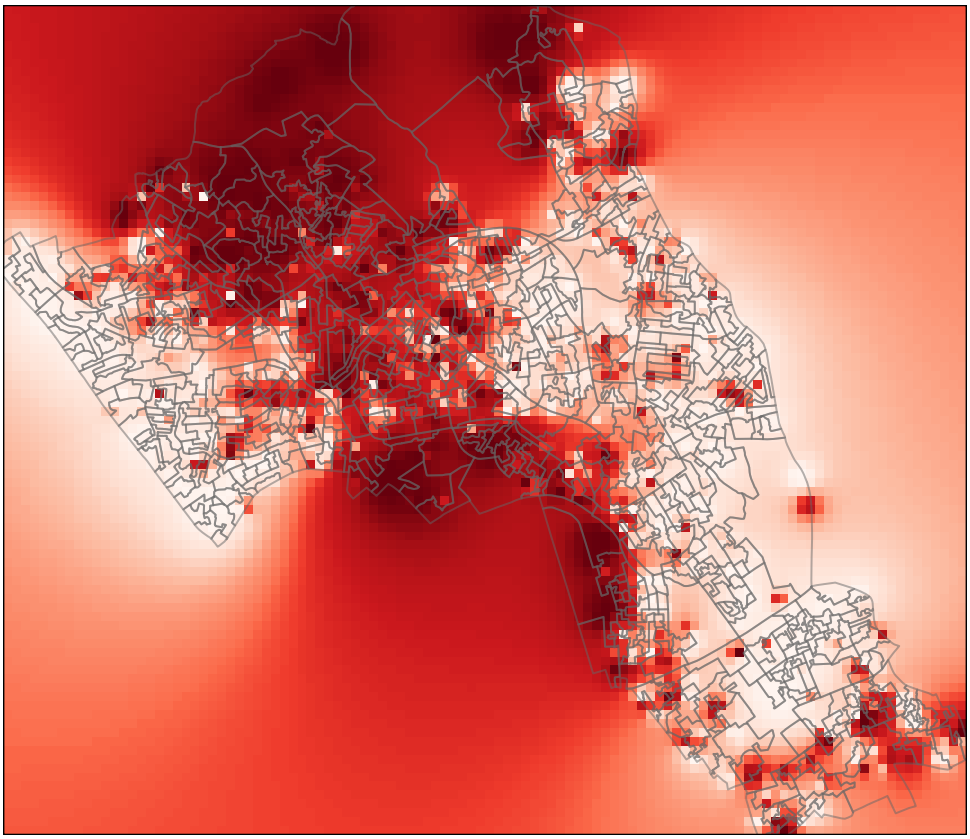
Now we are ready to plot the raster. As with our previous practicals, we will use the functionality of the tmap package, using the `tm_raster()` function. The instructions below will create a smoothed surface. In the example we have created 100 colour breaks and turned the legend off, this will create a smoothed colour effect. You can experiment by changing the breaks (n) to 7 and turning the legend back on (`legend.show`) to make the predicted values easier to interpret.

```
library(tmap)
tm_shape(raster_idw) + tm_raster("prediction", style = "quantile", n = 100,
                                 palette = "Reds", legend.show = FALSE)
```



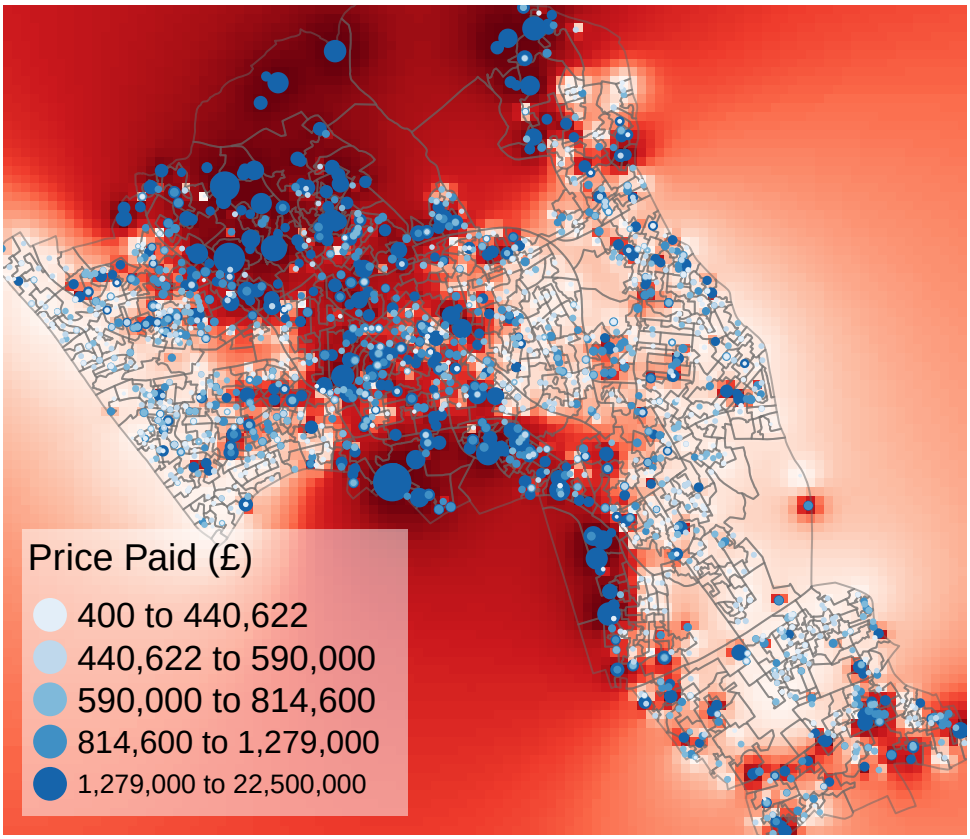
It is also possible to overlay the output area boundaries to provide some orientation. This requires us to load the Output.Areas shapefie with `tm_shape()`.

```
tm_shape(raster_idw) + tm_raster("prediction", style = "quantile", n = 100,
                                 palette = "Reds", legend.show = FALSE) +
tm_shape(Output.Areas) + tm_borders(alpha=.5)
```



We can also overlay the original house price data as this provides a good demonstration of how the IDW is distributed. In the example we have also edited the layout. Notice that we have made the legend background white (`legend.bg.color`) and also made it semi-transparent (`legend.bg.alpha`).

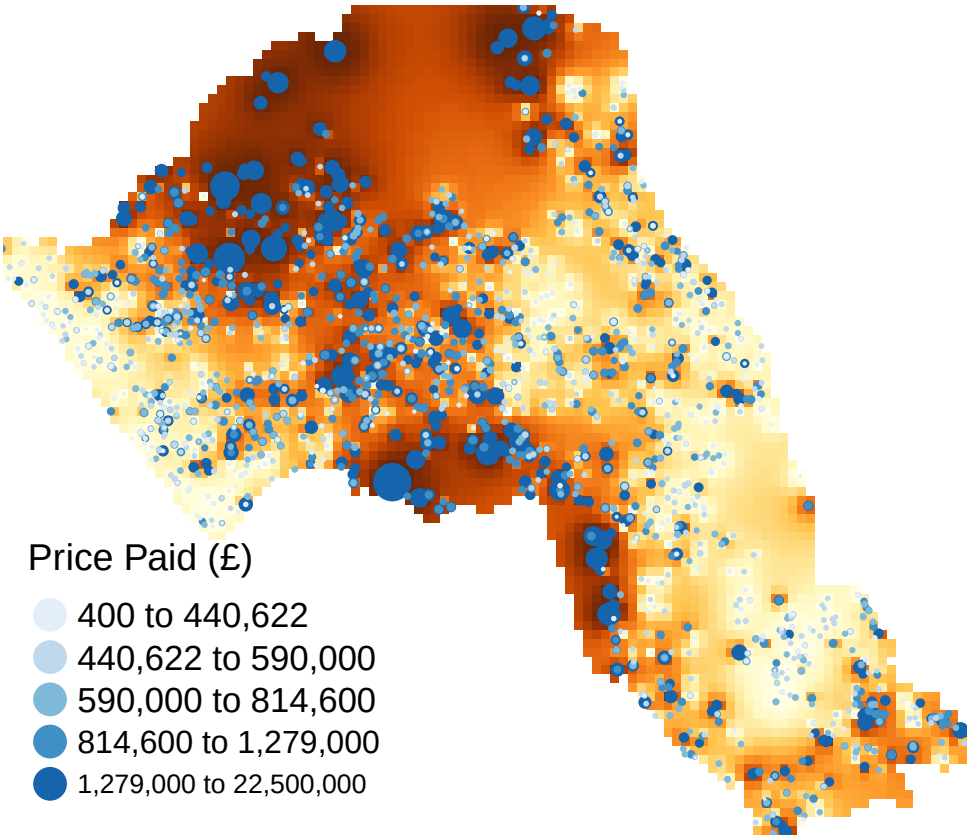
```
tm_shape(raster_idw) + tm_raster("prediction", style = "quantile", n = 100,
                                 palette = "Reds",legend.show = FALSE) +
tm_shape(Output.Areas) + tm_borders(alpha=.5,) +
tm_shape(House.Points) + tm_bubbles(size = "Price", col = "Price",
                                    palette = "Blues", style = "quantile",
                                    legend.size.show = FALSE,
                                    title.col = "Price Paid (£)") +
tm_layout(legend.position = c("left", "bottom"),  legend.text.size = 1.1,
          legend.title.size = 1.4, frame = FALSE, legend.bg.color = "white",
          legend.bg.alpha = 0.5)
```



Masking a raster

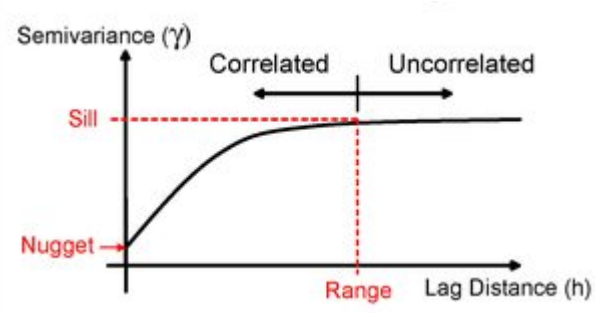
Masking is a GIS process of when a raster is clipped using the outline of another shapefile. In this case we will clip the raster so it only shows areas within the borough of Camden. The mask function is enabled by the raster package which we have already loaded.

```
# masks our raster by our output areas polygon file
masked_idw <- mask(raster_idw, Output.Areas)

# plots the masked raster
tm_shape(masked_idw) + tm_raster("prediction", style = "quantile", n = 100,
                                 legend.show = FALSE) +
tm_shape(House.Points) + tm_bubbles(size = "Price", col = "Price",
                                    palette = "Blues", style = "quantile",
                                    legend.size.show = FALSE,
                                    title.col = "Price Paid (£)") +
tm_layout(legend.position = c("left", "bottom"),  legend.text.size = 1.1,
          legend.title.size = 1.4, frame = FALSE)
```



Geostatistical interpolation

An IDW is not the only means of interpolating point data across space. A range of geostatistical techniques have also been devised. One of the most commonly used is kriging. Whilst an IDW is created by looking at just known values and their linear distances, a kriging also considers spatial autocorrelation. The approach is therefore more appropriate if there is a known spatial or directional bias in the data. Kriging is a statistical model which uses variograms to calculate the autocorrelation between points and distance. Like an IDW the the values across space are estimated from weighted averages formed by the nearest points and considers the influence of distance. However, the approach is also geostatistical given that the weights are also determined by a semivariogram.



A Semivariogram

There are various different ways of running a kriging, some are quite complex given that the method requires an experimental variogram of the data in order to run.

However, the automap package provides the simplest means of doing this. For a more comprehensive approach please see this practical - the practical data has been provided.

```
library(automap)
## Warning: package 'automap' was built under R version 3.3.1
# this is the same grid as before
grid <-spsample(House.Points, type = 'regular', n = 10000)

# runs the kriging
kriging_result = autoKrige(log(Price)~1, House.Points, grid)

plot(kriging_result)
```

Practical 12: Functions and Loops in R

This extra practical provides a very basic introduction to some programming techniques which can make tasks in R less time consuming if they involve repetition or iterating operations for multiple data. Data for the practical can be downloaded from the **Introduction to Spatial Data Analysis and Visualisation in R** homepage.

In this tutorial we will:

- Learn how to create user defined functions in R
- Run a simple for loop
- Learn how to implement if…else statements

First, we must set the working directory and load the practical data.

```
# Set the working directory
setwd("C:/Users/Guy/Documents/Teaching/CDRC/Practicals")

# Load the data. You may need to alter the file directory
Census.Data <-read.csv("practical_data.csv")
```

We will also need to load the spatial data files from the previous practicals.

```
# load the spatial libraries
library("sp")
library("rgdal")
library("rgeos")
library("tmap")

# Load the output area shapefiles
Output.Areas <- readOGR(".", "Camden_oa11")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "Camden_oa11"
## with 749 features
## It has 1 fields
```

```
# join our census data to the shapefile
OA.Census <- merge(Output.Areas, Census.Data, by.x="OA11CD", by.y="OA")
```

One of the core advantages of R and other command line packages over traditional software is the ability to automate several commands. You can tell R to repeat your methodological steps with additional variables without the need to retype the entire code again.

There are two ways of automating codes. One means is by implementing user-written functions, another is by running for loops.

User-written functions

User written functions allow you to create a function which runs a series of commands, it can include and input and output variable. You must also name the function yourself. Once the function has been run, it is saved in your work environment by R and you can run data through it by calling it as you would do with any other function in R (I.e. myfunction(data)).

The general structure is provided below:

```
# to create a new function
myfunction <- function(object1, object2, ... ){
```

```
statements
return(newobject)
}

# to run the function
Output <- myfunction(data)
```

When you create a function you do not need the name of the variables you use. You simply create a temporary object name (say 'x' for example), then when you run a data object through your function, it will take the place of x.

So for purposes of example, we will work on rounding our data first.

To round a variable we can simply use the `round()` function. It only requires us to input a data object and to specify how many decimal places we want our data rounded to.

So for one variable it would be:

```
# rounds data to one decimal place
round(Census.Data$Qualification, 1)
```

```
##   [1] 73.6 69.9 67.6 60.8 66.0 74.2 62.4 60.4 70.1 66.7 66.7 64.5 73.5 65.4
##  [15] 72.9 74.8 73.7 69.1 58.2 23.0 72.0 64.9 70.5 70.5 76.1 65.9 70.3 72.4
##  [29] 66.2 65.3 80.0 79.3 70.3 65.6 68.9 65.7 72.8 70.8 36.3 25.5 32.3 72.4
##  [43] 66.2 57.9 29.4 26.1 26.4 65.2 76.2 55.7 58.5 35.2 26.6 30.9 34.5 32.5
##  [57] 56.9 31.7 49.3 32.1 33.5 44.5 51.6 57.1 47.2 64.2 50.9 67.1 62.6 68.2
##  [71] 62.7 46.7 60.7 57.2 72.9 27.6 65.6 68.8 62.6 31.7 25.9 26.1 38.5 20.9
##  [85] 44.9 58.0 37.0 40.3 66.2 70.7 62.0 42.9 55.6 34.3 76.4 58.2 40.0 71.8
##  [99] 46.3 69.7 66.8 76.9 67.9 47.2 72.6 57.3 60.4 78.7 50.7 48.7 43.4 61.9
## [113] 46.7 44.8 70.6 74.7 67.1 26.6 43.1 31.1 61.5 49.3 66.3 62.9 66.5 56.5
## [127] 35.9 29.4 47.0 42.1 57.1 59.5 27.1 27.3 37.1 24.5 30.6 58.7 48.0 39.9
## [141] 39.8 43.3 39.6 59.4 63.3 66.2 60.4 57.3 47.0 56.1 63.4 41.9 59.5 64.5
## [155] 56.3 51.5 48.8 65.0 50.7 67.9 50.2 65.0 50.2 30.9 25.0 51.1 61.3 59.9
## [169] 37.0 58.4 58.6 65.7 43.3 46.2 46.6 69.4 63.1 64.8 65.5 64.1 70.4 70.6
## [183] 67.5 67.6 62.2 70.5 56.0 69.6 55.1 65.9 65.7 66.1 70.6 57.9 64.8 28.5
## [197] 66.7 61.4 55.7 71.5 60.7 59.1 76.6 66.4 62.9 72.3 63.7 70.5 70.1 58.9
## [211] 79.4 61.1 59.6 70.4 70.0 68.8 68.3 55.1 37.0 58.9 60.5 68.0 65.0 59.4
## [225] 72.7 53.6 61.2 55.2 71.2 71.2 48.9 67.6 68.3 46.2 27.1 21.2 32.4 72.3
## [239] 24.7 35.6 71.9 27.0 23.4 19.8 40.4 23.6 28.2 69.3 37.5 18.1 55.5 65.8
## [253] 29.3 60.6 67.4 64.5 57.9 66.9 65.2 67.2 72.9 75.8 58.4 70.0 59.8 42.8
## [267] 47.8 61.5 34.4 27.7 43.7 23.2 28.3 73.8 76.9 50.8 70.6 66.2 71.6 55.6
## [281] 68.0 51.9 63.1 76.8 74.9 69.2 68.2 73.0 73.1 71.1 72.7 63.6 76.6 70.5
## [295] 66.0 69.8 69.5 75.5 70.1 72.9 75.5 70.5 37.0 71.2 73.2 73.9 46.0 68.2
## [309] 70.1 67.3 80.1 73.1 68.4 29.6 29.3 47.3 38.7 42.9 37.0 46.1 46.1 52.3
## [323] 28.5 29.7 30.3 23.3 70.7 45.4 27.8 29.7 27.5 60.2 18.6 16.4 38.9 73.6
## [337] 28.5 23.6 70.7 45.7 77.7 40.9 66.7 58.7 30.9 19.4 63.6 55.5 54.8 54.0
## [351] 33.1 66.3 72.5 64.6 50.4 60.2 57.7 20.9 55.0 41.0 32.3 63.7 53.2 70.0
## [365] 65.2 59.7 34.0 40.7 66.6 64.4 61.2 35.4 45.8 66.4 41.5 41.9 66.8 57.7
## [379] 51.7 65.6 66.9 52.9 71.3 67.4 59.9 68.9 39.4 39.8 69.0 37.4 67.2 56.5
## [393] 42.7 68.9 42.4 23.1 35.8 53.6 50.4 50.9 44.4 67.7 26.6 61.5 74.3 53.1
## [407] 31.7 43.7 31.6 37.6 42.2 63.5 26.8 30.2 43.8 45.1 59.5 26.8 40.1 68.3
## [421] 32.4 37.8 56.0 31.1 43.0 35.1 51.6 54.9 37.5 37.1 52.1 36.1 58.4 27.9
## [435] 44.2 25.3 49.4 52.5 65.2 62.3 60.6 48.1 59.8 67.4 45.5 42.3 58.5 60.9
## [449] 60.6 63.6 56.5 50.3 56.0 59.5 64.7 52.5 66.3 59.1 68.1 49.2 61.3 59.3
## [463] 56.9 45.0 68.2 80.1 57.4 69.8 28.7 38.3 70.6 30.8 32.7 32.3 46.5 22.5
## [477] 32.6 42.7 41.3 21.4 37.2 35.6 40.8 64.9 62.3 67.4 57.9 48.8 47.8 42.2
## [491] 71.4 51.4 66.9 52.4 41.3 33.6 23.8 63.3 61.8 44.1 26.7 22.6 67.6 28.0
```

```
## [505] 29.3 25.5 57.4 41.8 42.9 30.3 20.3 51.6 27.5 48.3 32.7 41.5 56.5 54.8
## [519] 54.2 47.1 65.8 57.7 67.1 69.3 27.9 35.5 40.0 29.8 37.5 18.8 47.2 20.8
## [533] 34.2 66.2 31.5 21.3 28.2 32.8 57.3 28.7 27.9 28.7 22.3 36.4 33.8 30.7
## [547] 67.1 65.6 21.6 23.2 30.6 25.0 36.5 59.3 28.8 23.4 37.1 11.6 17.4 34.5
## [561] 45.3 41.8 22.7 38.5 23.1 18.8 40.2 28.8 47.1 45.3 65.1 49.2 54.2 57.0
## [575] 52.2 43.9 52.2 53.1 24.5 59.5 37.2 28.4 34.8 22.8 24.5 33.2 21.6 16.5
## [589] 29.4 33.3 13.5 27.1 34.7 22.5 17.9 14.7 46.2 21.0 53.1 16.3 17.8 44.9
## [603] 60.9 59.5 24.2 34.9 40.8 25.7 20.5 18.8 27.6 29.9 28.0 19.9 38.8 30.2
## [617] 34.2 26.5 34.0 21.9 25.2 17.3 16.2 25.5 12.8 20.0 24.6 33.2 26.8 25.9
## [631] 72.0 71.8 74.7 70.5 63.8 50.4 71.7 63.5 68.7 70.8 63.1 34.5 50.9 57.3
## [645] 51.5 48.6 49.5 47.9 46.7 36.0 57.3 61.3 72.3 41.2 69.6 54.4 66.8 79.8
## [659] 47.4 65.9 69.3 35.7 74.0 73.5 71.0 72.2 61.2 80.4 30.2 78.6 52.8 65.8
## [673] 60.0 77.9 75.9 48.8 75.1 79.5 54.4 64.3 65.9 64.8 65.6 62.3 69.3 62.5
## [687] 41.8 62.6 76.6 74.8 49.1 40.5 67.0 57.8 32.1 49.5 53.7 48.3 62.9 42.3
## [701] 61.4 33.0 48.3 64.4 43.2 61.6 64.4 67.1 66.7 73.4 70.6 72.5 70.3 79.7
## [715] 79.4 64.7 64.2 29.4 26.8 13.7 55.4 30.2 17.8 63.0 55.6 57.9 61.6 61.9
## [729] 35.7 31.0 75.6 70.9 80.0 54.2 61.0 46.7 33.5 42.2 68.6 62.4 84.6 88.1
## [743] 60.4 52.8 37.1 50.7 53.2 45.3 24.7
```

This is very straight forward. We will now put this in a new function called "myfunction" which will comprise this code.

```
# function is called "myfunction", accepts one object (to be called x)
myfunction <- function(x){

# x is rounded to one decimal place, creates object z
z <- round(x,1)

# object z is returned
return(z)

# close function
}
```

Now run the function, it should appear in your environment panel. To run it we simply run data through it like so:

```
# lets create a new data frame so we don't overwrite our original data object
newdata <- Census.Data

# runs the function, returns the output to new Qualification_2 variable in newdata
newdata$Qualificaton_2 <- myfunction(Census.Data$Qualification)
```

This example is very simple. However, we can add further functions into our function to make it undertake more tasks. In the demonstration below, we also run logarithmic transformation prior to rounding the data.

```
# creates new function which logs and then rounds data
myfunction <- function(x){

z <- log(x)
y <- round(z,4)

return(y)

}

# resets the newdata object so we can run it again
```

```
newdata <- Census.Data

# runs the function
newdata$Qualification_2 <- myfunction(Census.Data$Qualification)
newdata$Unemployed_2 <- myfunction(Census.Data$Unemployed)
```

So here we create the z variable first, then run it through a second function to produce a y variable. However, as the code is run in order, we could give both z and y the same name to overwrite them as we run the function. I.e.

```
myfunction <- function(x){

z <- log(x)
z <- round(z,4)

return(z)

}
```

Now we will try something more advanced. We will enter our mapping code using tmap into a function so we are able to create maps more easily in the future. Here we have called the function 'map'. The code has three arguments this time (of course we could include more if we wanted to add more customisations) they are described in the table below.

Object label	Argument
x	A polygon shapefile
y	A variable of the shapefile
z	A Colour palette

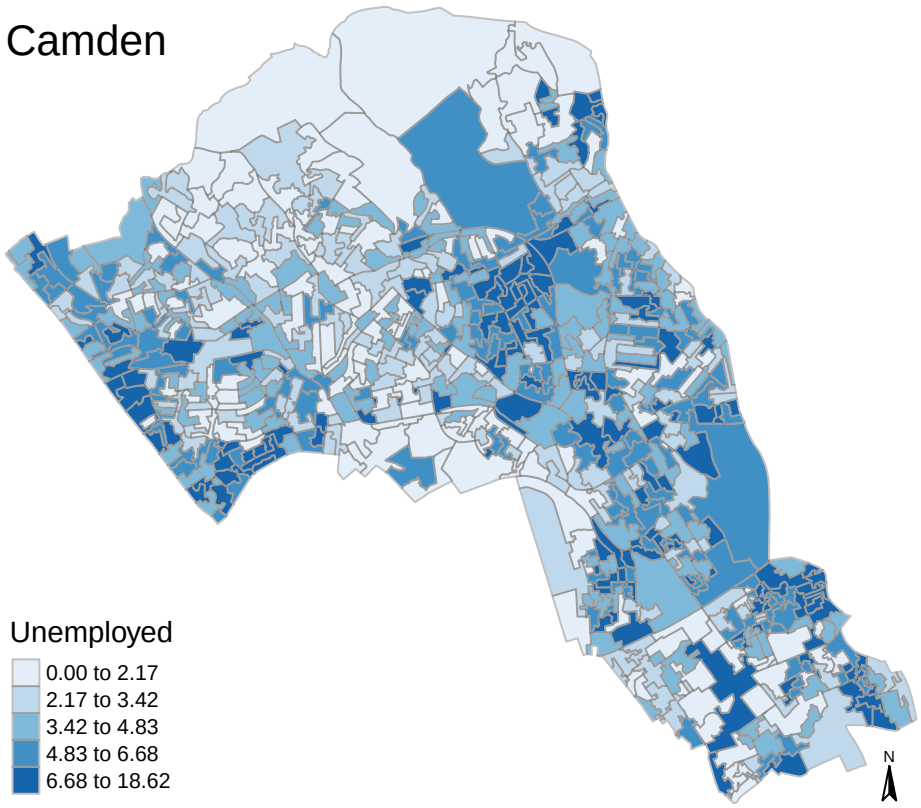
So the code is as follows...

```
# map function with 3 arguments
map <- function(x,y,z){

tm_shape(x) + tm_fill(y, palette = z, style = "quantile") + tm_borders(alpha=.4) +
tm_compass(size = 1.8, fontsize = 0.5) +
tm_layout(title = "Camden", legend.title.size = 1.1, frame = FALSE)

}
```

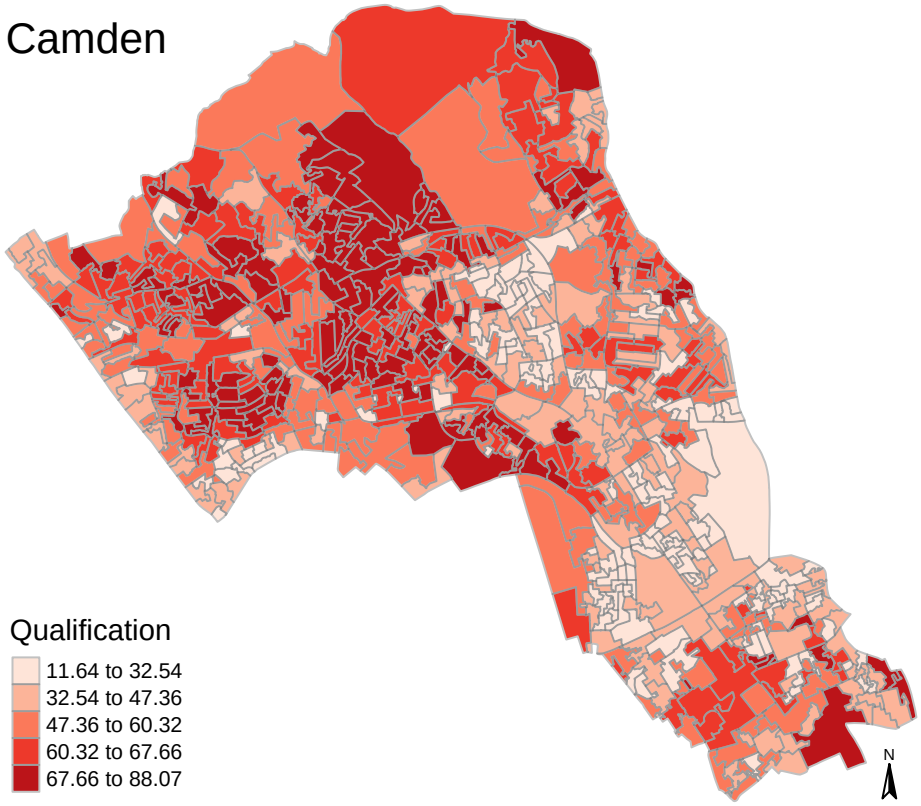
Now to run the *map* function, we can very simply call the function and enter the three input parameters (x,y,z). Running our function is an easier way to create consistent maps in the future as it requires only one small line of code to call it.

We can now use the function to create a map for any spatial polygon data frame which contains numerical data. All we have to do is call the function and enter the three arguments. There are two examples below.

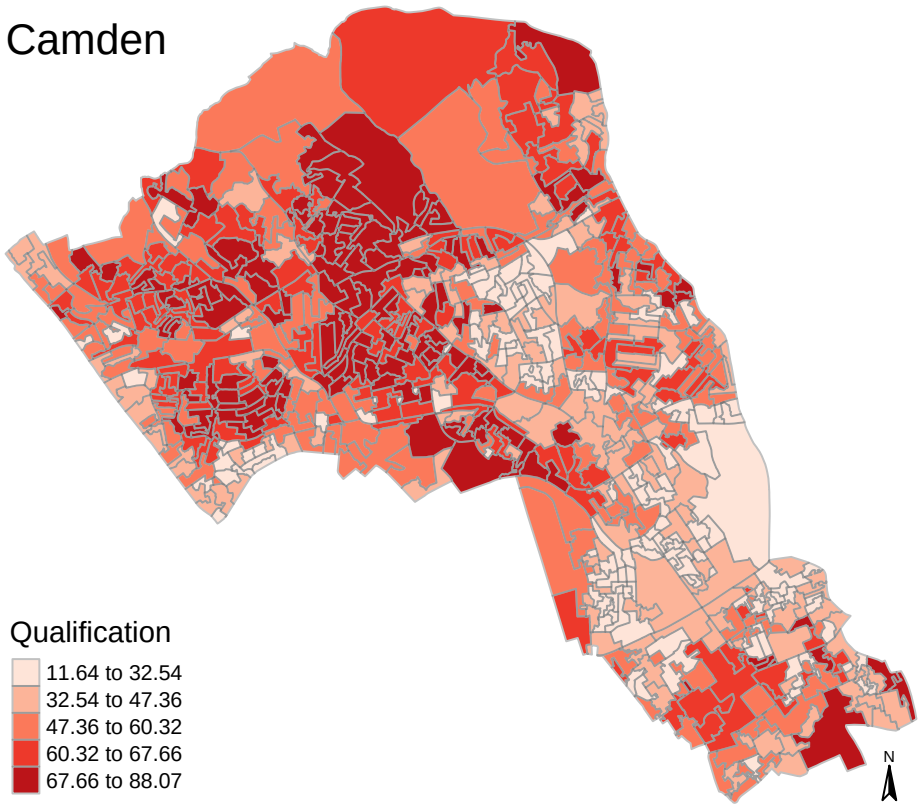
```
# runs map function, remember we need to include all 3 arguments of the function
map(OA.Census, "Unemployed", "Blues")
```




```
map(OA.Census, "Qualification", "Reds")
```


Camden
Qualification
11.64 to 32.54
32.54 to 47.36
47.36 to 60.32
60.32 to 67.66
67.66 to 88.07
N

```
# we can easily change the key parameters. i.e...
map(OA.Census, "Qualification", "Reds")
```



Loops

A loop is a means to repeat code under defined conditions. This is a great way for reducing the amount of code you may need to write. For example, you could use the loop to iterate a series of commands through a number of different positions (i.e. rows) in a data frame.

For loop

The most basic means of looping commands in R is known as the for loop. To implement it,you run the `for()` function and within the parameters you define the sequence through which the loop will input data. Usually an argument is assigned (i) and given a range of numerical positions (i.e. `in 2:5` which runs from position 2 to position 5). You next enter the commands which the loop will run within curly brackets following the parameters. The defined argument (i) is used to determine the input data for each iteration. This has been demonstrated in the example below which loops the code for four columns of our data frame.

Note in this example we are running the loop from columns 2 onwards as column 1 does not contain our numeric data

```
#create a new data frame of the same properties as our data file
newdata <- Census.Data

# a for loop where i iterates from 2 to 5
```

```
for(i in 2:5){
 # i is used to identify the column number
 newdata[, i] <- round(Census.Data[,i], 1)

}

# open the new data
View(newdata)
```

We can also use the `ncol()` function if you do not know the total number of columns or accept that the number may change under certain circumstances.

```
# use ncol to determine the number of columns
for(i in 2: ncol (Census.Data)){

 newdata[, i] <- Census.Data[,i]/100

}
```

We will be using the ratio data from newdata in the upcoming section

If. else statements

There are various commands we can run within for loops to edit their functionality. If-else statements allow us to change the outcome of the loop depending on a particular statement. For instance, if a value is below a certain threshold, you can tell the loop to round it up, otherwise (else) do nothing.

This time we are going to run the loop for every row in one variable to recode them to "low" and "high" labels. In this example we are asking if the value in our data is below 0.5, if it is then we are giving it a "low" label. The else argument determines what R will do if the value does not meet the criterion, in this case it is given a "high" label.

```
# creates a new newdata object so we have it saved somewhere
newdata1 <- newdata

for(i in 1:nrow(newdata)){

if (newdata$White_British[i] < 0.5) {
   newdata$White_British[i] <- "Low";

} else {
   newdata$White_British[i] <- "High";

}
}
```

We can also input multiple else statements.

```
# copies the numbers back to newdata so we can start again
newdata <- newdata1

for(i in 1:nrow(newdata)){

if (newdata$White_British[i] < 0.25) {
   newdata$White_British[i] <- "Very Low";
```

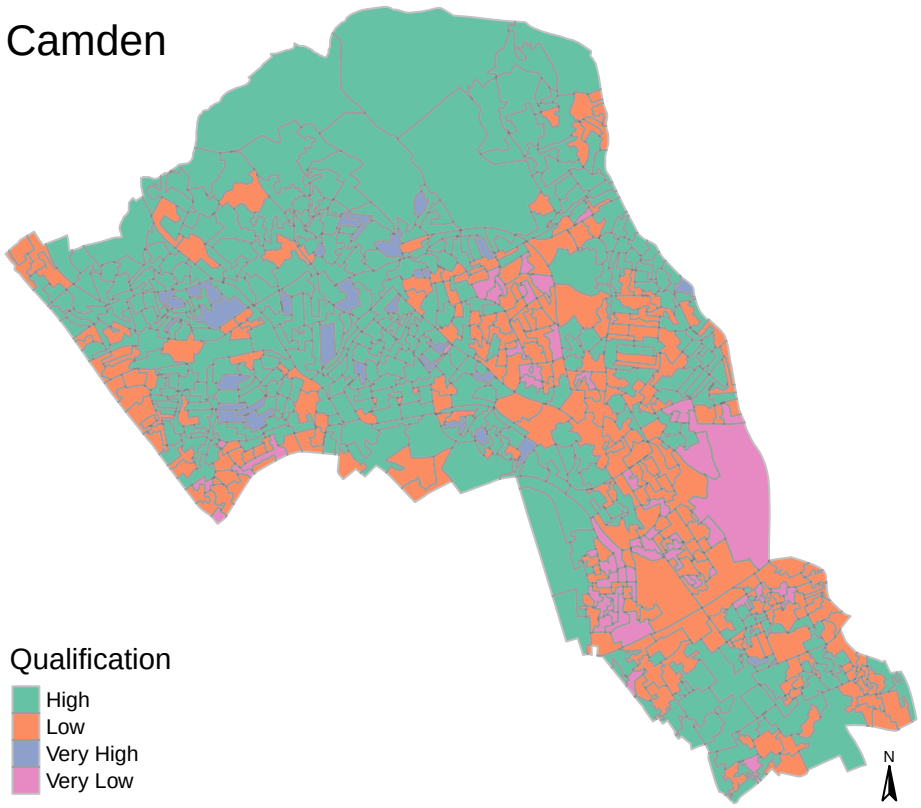
```
} else if (newdata$White_British[i] < 0.50){
   newdata$White_British[i] <- "Low";

} else if (newdata$White_British[i] < 0.75){
   newdata$White_British[i] <- "High";

} else {
   newdata$White_British[i] <- "Very High";

}
}
```

We can also nest multiple loops if we wish to loop through two different arguments. In this example, we are going to loop both the columns and rows for our newdata data frame. This will reassign all of our variables in the data into four interval labels.

```
# copies the numbers back to newdata so we can start again
newdata <- newdata1


for(j in 2: ncol (newdata)){

for(i in 1:nrow(newdata)){

if (newdata[i,j] < 0.25) {
   newdata[i,j] <- "Very Low";

} else if (newdata[i,j] < 0.50){
   newdata[i,j] <- "Low";

} else if (newdata[i,j] < 0.75){
   newdata[i,j] <- "High";

} else {
   newdata[i,j] <- "Very High";

}
}
}
```

We can join the newdata data frame to our output area shapefile to map it using our earlier generated *map* function.

```
# merge our new formatted data with the output areas shapefile
shapefile <- merge(Output.Areas, newdata, by.x = "OA11CD", by.y = "OA")

# runs our predefined map function
map(shapefile, "Qualification", "Set2")
```



You can now adapt these techniques to your own code in the future to avoid writing lengthy and repetitive commands when analysing large multivariate datasets.

Acknowledgements

This work is funded by the UK ESRC Consumer Data Research Centre (CDRC). grant reference ES/L011840/1

References

Bivand, R.S., Pebesma, E.J. and Gómez-Rubio, V. (2008). *Applied spatial Data Analysis with R.* 2nd Edition. Springer, New York, USA.

Brunsdon, C. and Comber, L. (2015). *An Introduction to R for Spatial Analysis and Mapping.* SAGE Publications Ltd, London, UK

Brunsdon, C. and Singleton, A. eds. (2015) *Geocomputation: A Practical Primer.* SAGE Publications Ltd, London, Uk

Everitt, B.S. & T. Hothorn. (2014). *A Handbook of Statistical Analyses Using R (3rd ed.)* Boca Raton, Chapman & Hall

de Smith, M. J. (2015). *Statistical Analysis Handbook: A Comprehensive Handbook of Statistical Concepts, Techniques and Software Tools.* he Winchelsea Press, Winchelsea, UK. Available online: http://www.statsref.com

de Smith, M.J., Goodchild, M.F. and Longley, P. (2015) *Geospatial Analysis: A Comprehensive Guide to Principles, Techniques and Software Tools.* 5th Edition. The Winchelsea Press, Winchelsea, UK. Available online: http://www.spatialanalysisonline.com/HTML/index.html

Kabacoff, R. (2015). *R in Action: Data Analysis and Graphics with R.* Manning Publications Co., Greenwich, USA

Longley, P.A., Goodchild, M., Maguire, D.J. and Rhind, D.W. (2010). *Geographic Information Systems and Science.* 4th Edition. John Wiley & Sons

---

Lansley, G and Cheshire, J (2016). An Introduction to Spatial Data Analysis and Visualisation in R. CDRC Learning Resources. Online: https://data.cdrc.ac.uk/tutorial/an-introduction-to-spatial-data-analysis-and-visualisation-in-r