R and Spatial Data

Prof. Uttam Kumar, Mayur Rajwani

2020-01-17. See github.com/rajwanimayur/SpatialAnalysis.pdf for latest version

Contents

Preface **1**

Why R? 1

1. Introduction **3**

2. Handling Spatial Data in R **3**

The Structure of spatial data in R 4

Basic Plotting 4

Projections: setting and transforming CRS in R 6

3. Data Preparation **7**

4. Making Maps in R **10**

ggplot2 10

ggmap 10

tmap 15

5. Spatial Analysis in R - A Deeper Look **18**

References **23**

Preface

The objective of this tutorial is to provide its readers the necessary skills to visualise diverse range of geographical and non-geographical datasets and make meaningful inferences.

The tutorial is practical in nature, where you will load-in, visualise and manipulate spatial data. The tutorial requires basic knowledge of R, thus if you do not have prior experience of using R, it is advised to follow the tutorial - *R for beginners by Emmanuel Paradis*.

Why R?

1. R is free and open-source.
2. Using R, researchers can reproduce their work and verify other people's work.
3. R has numerous packages available for spatial data analysis, statistical modelling, econometrics, visualising and more.

Now you know the usage and benefits of using R for spatial data analysis, its now time to delve deeper into it. To that end, this tutorial is organised as follows:

1. **Introduction** : provides a guide to R's syntax and preparing you for the tutorial.
2. **Handling Spatial Data in R**: describing basic spatial functions in R
3. **Data Preparation** : which include creating and manipulating spatial data for visualisations.
4. **Data Visualisation - Making Interactive Maps in R** : this section will demonstrate map making with R's advanced visualisation tools, namely **ggmap**.
5. **Analysing Spatial Data** : In this section, we analyse the road accidents data for all states in India, over all the period of four years altogether.

To distinguish between prose and code, please be aware of the following typographic conventions used in this document: R code (e.g. `plot(x, y)`) is written in a `monospace` font and package names (e.g. **rgdal**) are written in **bold**. A double hash (`##`) at the start of a line of code indicates that this is output from R. Lengthy outputs have been omitted from the document to save space, so do not be alarmed if R produces additional messages: you can always look up them up on-line.

> As an advice, we encourage you to play with the code. Try out new things, explore different ways to do similar tasks and enjoy the learning ride.

This tutorial makes use of RStudio, an IDE(Integrated Development Environment) with code editor and tools for debugging and visualization. RStudio makes it easier to use R. To download Rstudio, visit - https://www.rstudio.com

1. Introduction

R has a huge and growing number of spatial data packages. We recommend taking a quick browse on R's main website to see the spatial packages available at: http://cran.r-project.org/web/views/Spatial.html. We highlight few packages below, which are used extensively in this tutorial. One is advised to download all recommended packages beforehand.

- **sp** : This package provides classes and methods for spatial data; utility functions for plotting maps, working with coordinates, etc.
- **ggplot2** : The most popular package for data visualization.
- **ggmap** : extends the plotting package ggplot2 for maps. ggmap provides functions to visualise spatial data on top of static maps from sources like Google Maps, Open Street Maps, cloudmade and stamen.
- **rgdal** : This package provides methods for working with importing and exporting different raster and vector geospatial data formats; Coordinate Reference Systems; projections, etc.
- **rgeos** : R's interface to the powerful vector processing library geos. It provides functions for handling operations on topologies.
- **dplyr** and **tidyr** : fast and concise data manipulation packages
- **broom** : Takes messy output of built-in functions in R, and turns them into tidy data frames.
- **tmap** : Package for rapidly creating beautiful maps.
- **adehabitatHR** : Package providing functions for home-range analysis
- **raster** : Package for reading/writing, analyzing and modelling of gridded spatial data.

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

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

```
x <- c("sp", "ggplot2", "ggmap", "rgdal", "rgeos", "dplyr", "tidyr", "broom", "ggsn", "adehabitatHR", "
# install.packages(x) # warning: uncommenting this may take a number of minutes
lapply(x, library, character.only = TRUE) # load the required packages
```

2. Handling Spatial Data in R

In this section, we will look at mapping and geographical data handling capabilities of R. The aim of this section is to develop basic building blocks for the Spatial data analysis in later sections.

All code and data used for this tutorial can be downloaded from github repository. Click on the "Download ZIP" button on the right hand side of the screen and once downloaded, unzip this to a new folder on your computer.

Open the existing "GISTutorial" project using File -> Open File... on the top menu. Alternatively, you can also use the project menu to open the project or create a new one. It is recommended to use RStudio's projects to organise R work and organise files into sub-folders and avoid digital clutter.

The first file we are going to load into R Studio is the "states" shapefile located in the `Data/States` folder of the project. It is worth looking at this input dataset in your file browser before opening it in R. You will notice that there are several files named "states", all with different file extensions. This is because a shapefile is actually made up of a number of different files, such as .prj, .dbf and .shp. You could also try

opening the file "states.shp" file in a conventional GIS such as QGIS to see what a shapefile contains. You should also open "states.dbf" in a spreadsheet program such as Microsoft Excel, or LibreOffice Calc. to see what this file contains. Once you think you understand the input data, it's time to open it in R. There are a number of ways to do this, the most commonly used and versatile of which is **readOGR**. This function, from the **rgdal** package, automatically extracts the information regarding the data.

rgdal is R's interface to the "Geospatial Abstraction Library (GDAL)" which is used by other open source GIS packages such as QGIS and enables R to handle a broader range of spatial data formats.

```
library(rgdal)
states_raw <- readOGR(dsn = "Data/States/states.shp", stringsAsFactors = FALSE)
states <- states_raw
```

In the second line of code above the **readOGR** function is used to load a shapefile and assign it to a new spatial object called *states*.

readOGR is a *function* of the '**rgdal** package, the first *argument* of which **dsn:** "data source name", the file or directory of the geographic data to be loaded. Recent developments of **rgdal** package doesn't require specification of layer file any longer. *states* is now an spatial data object with boundary data specifying the state name.

The Structure of spatial data in R

Spatial objects like the **states** object are made up of a number of different *slots*. *slots* provides the object oriented representation of data in R. Each slot in an object (here *states*) is a component of the object; like components (that is, elements) of a list, this may be extracted and set, using the function 'slot()' or more often the operator ' "@" '. However, they differ from list components in important ways. First, slots can only be referred to, by name and not by position, and there is no partial matching of names as with list elements.

Lets have a look at slots in **states** object.

```
slotNames(states)
```

```
## [1] "data"        "polygons"    "plotOrder"   "bbox"        "proj4string"
```

In the above slots, the **data** slot and **polygons** are of interest to us. The *data* slot contains non geographic attribute data (in this case the state name), while *polygons* slot (or *lines* slot for line data) contains geographic information of latitude and longitude. Thus the data slot is considered as the attribute table and the geometry slot is the polygons that make up the physical boundaries.

Basic Plotting

Now we have seen structure of our spatial object states, we will now look at plotting them.

```
plot(states) # Not shown, will plot the shapefile
```

plot is one of the most useful functions in R. It can be used to draw maps for spatial data objects as well as conventional graphs. This is often referred as polymorphism.

R also provide powerful subsetting capabilities that can be accessed very concisely using square brackets.

```
#We first see the classes of all variables in spatial dataset
sapply(states@data, class)
```

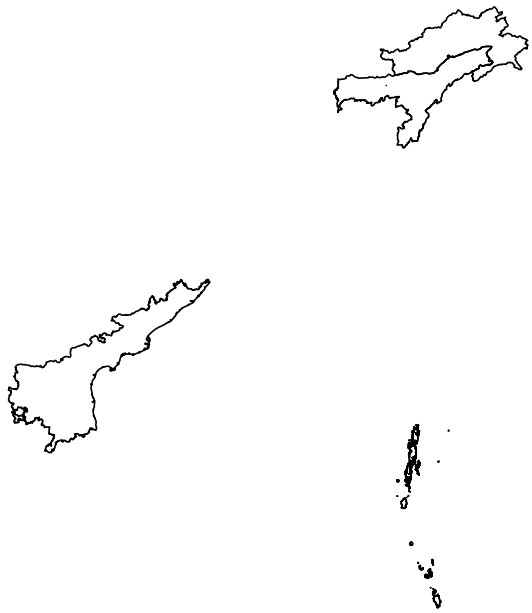


Figure 1: States whose names starts with A

```
##       ST_NM
## "character"
```

As evident, `ST_NM` is of type character, Thus we do not need any form of type conversion. Next we select states whose name starts with "A" only.

```
states@data[startsWith(states$ST_NM, "A"),]
```

```
## [1] "Andaman & Nicobar Island" "Arunanchal Pradesh"
## [3] "Assam"                    "Andhra Pradesh"
```

Next we will plot these selected states.

```
sel <- startsWith(states$ST_NM, "A")
plot(states[sel,])  # See Figure 1
```

The plot is quite useful, but it only display the areas we selected. In order to see the areas that meets the selection criteria, in context with the other areas of the map we simply use the $add = TRUE$ argument after the initial plot.

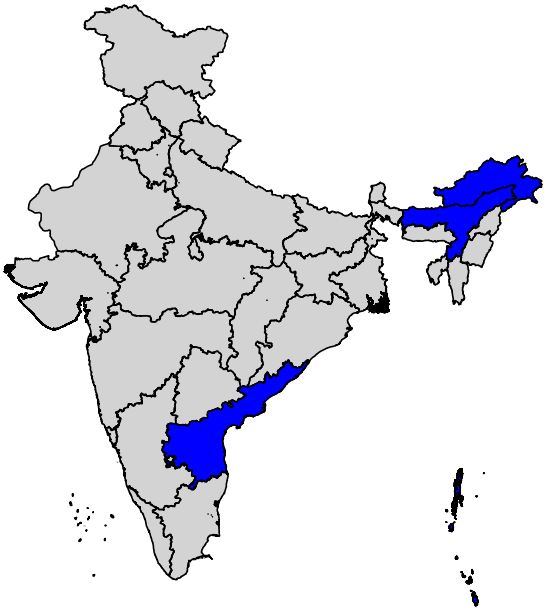


Figure 2: Map of India, highlighting the states whose name starts with 'A'

```
#Plotting initial graph
plot(states, col = "lightgray")
sel <- startsWith(states$ST_NM, "A")
plot(states[sel,], col = "blue", add = TRUE)  #See Figure 2
```

Projections: setting and transforming CRS in R

The next section requires the spatial object to have planar projections. Thus, we first modify our spatial object to have a planar projection and then carry our further exploration of data.

```
states@proj4string
```

```
## CRS arguments:
##  +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
```

The *Coordinate Reference System* (CRS) of spatial objects defines where they are placed on the Earth's surface. The above R code displays the summary of '`proj4string`' slot of `states`. The information that follows represents its CRS. Spatial data should always have a CRS. If no CRS information is provided, and the correct CRS is known, it can be set as follow:

```
# Remove the existing CRS Information first.
proj4string(states) <- NA_character_
proj4string(states) <- CRS("+init=epsg:27700")
```

R issues a warning when the CRS is changed. This is so the user knows that they are simply changing the CRS, not *reprojecting* the data. An easy way to refer to different projections is via EPSG codes.We used CRS system `27700` representing the British National Grid. Also its worth mentioning the commonly used CRS worldwide is 'WGS84' with EPSG code `epsg:4326`.

Now we have projected spatial object, we will now select and plot only zones that are close to centroid of India.

```
library(rgeos)
plot(states, col = "grey")

#Find India geographic centroid
cent_del <- gCentroid(states[states$ST_NM == "NCT of Delhi",])
points(cent_del, cex = 3)

#Set  2° buffer
del_buffer <- gBuffer(spgeom = cent_del, width = 2)

# We first select any intersecting zones
del_central <-  states[del_buffer,]
# Now we plot our selection
plot(del_central, col = "lightblue", add = T)

# Next we highlight the buffer area
plot(del_buffer, add = T, border = "red", lwd = 2)
```

3. Data Preparation

In the following section we shall use the dataset representing road accidents in India Source - https://data.gov.in/node/6619609. The data provides statewise accidents record over the year 2014 to 2017. The available data is a non-spatial data, i.e, it doesn't contain location information of latitude and longitude. The Spatial data we have, contains only names of state as an attribute. It does not contain any useful information. How can we add more infomration to our existing spatial objects? The following section will answer this!

The non-spatial object we are going to join our `states` object contains road accidents records for all States/UTs. The file contains comma seperated values and can be opened using any spreadsheet program like MS Excel.

```
# Read csv file and Create a new data frame object accidents
accidents_raw<- read.csv("Data/Road_Accidents_2017-Annuxure_Tables_4.csv", stringsAsFactors = FALSE)
accidents <- accidents_raw #Avoid Re-Reading of CSV File
```

Note that accidents is a Data Frame containing 36 rows and 19 variables. We can also infer that column *States.UTs* is of type `chr`, i.e a String(Hence, no need for type coercion), and rest other columns are `integer` and `double`. Before we join the two datasets, we check for consistency among the joining variable. Since the state names should ubiquitous, we check for values in `accidents$States.UTS` which do not occur in `states$ST_NM` and vice - versa. We need to handle these missing values accordingly.

Figure 3: Plot highlighting Indian states

```
accidents$States.UTs[!accidents$States.UTs %in% states$ST_NM]
```

```
## [1] "Arunachal Pradesh"         "Andaman & Nicobar Islands"
## [3] "Dadra & Nagar Haveli"      "Delhi"
```

```
states$ST_NM[!states$ST_NM %in% accidents$States.UTs]
```

```
## [1] "Andaman & Nicobar Island" "Arunanchal Pradesh"
## [3] "Dadara & Nagar Havelli"   "NCT of Delhi"
```

We have found the discrpenacies as a typrographical error. Therefore we edit the states name in `states$ST_NM` from R's data editor. The following line of code will open an editor, you need to correct the errors manually. **Note** - Do not change the order of values.

```
states@data <- edit(states@data)
```

Now, we can join our data. We make use of package **dplyr** to join two data frames. One can also use the `merge` function for the same.

```
library(dplyr)  #load dplyr
states@data <- inner_join(states@data, accidents, by = c('ST_NM' = 'States.UTs'))
```

The `*join` command assume, by default, that matching variables have same name. Here we specify the association between two data frames, using the '`by`' argument.

4. Making Maps in R

ggplot2

In this section we will look at **ggplot2** package. The package is implementation of the Grammar of Graphics (Wilkinson 2015) - a general scheme for data visualisation that breaks up graphs into semantic components such as scales, layers and facets. **ggplot2** makes plotting of complex plots easier by providing a programmatic interface for variable specification, displaying of plots and other general visual properties.

ggplot2 is a well-documented package and contains number of default options that match good visualisation practices.

We start with a scatter plot for attribute data in `states` object. As a first step, set up the data we need to plot. For this, select the subset of data that include variables of interest. Here we select variables specifying "Total Number of Persons Injured" in a calendar year. Now our dataset represents a single variable of information, spanned across the duration of four years. Hence, we rename columns to represent only year number, representing number of persons injured in each state for that year.

```
#Create a temporary dataframe for analysis purpose
accidents_df <- states@data[,1:5]
#Rename Columns to Year only
colnames(accidents_df) <- gsub("\\D+\\.\\.\\.", "", colnames(accidents_df))
```

Before we move forward, we `tidy` the data and transform columns into rows, using function `pivot_longer()` from **tidyr** package. We create a single tuple of the form **<ST_NM, Year, Person_Injured>**.

```
library(tidyr)
accidents_df.m <- pivot_longer(accidents_df, colnames(accidents_df)[2:5], names_to = "Year", values_to =
```

We have prepared our data for exploration. We set up a `ggplot` object, which takes dataframe object `df.m` as input. We specify the type of plot as scatter plot, using `geom_point()` function. To define the aesthetics, i.e, variables of graph, use the `aes` function. Use `facet_wrap()` to plot different states' scatter plot, side by side.

```
library(ggplot2)
p1 <- ggplot(data = accidents_df.m[startsWith(accidents_df.m$ST_NM, "M"),],
             aes(x = Year, y = Persons_Injured)) +
  geom_point(aes(size = Persons_Injured, color = Year)) +
  facet_wrap(~ST_NM)
p1 + ggtitle("Number of Persons Injured in Road Accidents") +  theme(legend.position = "bottom")
```

The beauty of **ggplot2** package lies in adding of layers. It allows us to plot an spatial object `df.m`, directly, and add layers to it.

ggmap

This section provides an introduction to mapping spatial data using **ggmap** package for R, which enables the creation of maps with `ggplot`. Based on **ggplot2**, **ggmap** package requires data (including spatial data) to be supplied as `data.frame`. We use `tidy()` method from package `broom` to coerce spatial object to a data frame object, and allow plot() functions to use `Spatial*` objects directly.

```
states_df <- broom::tidy(states)
```

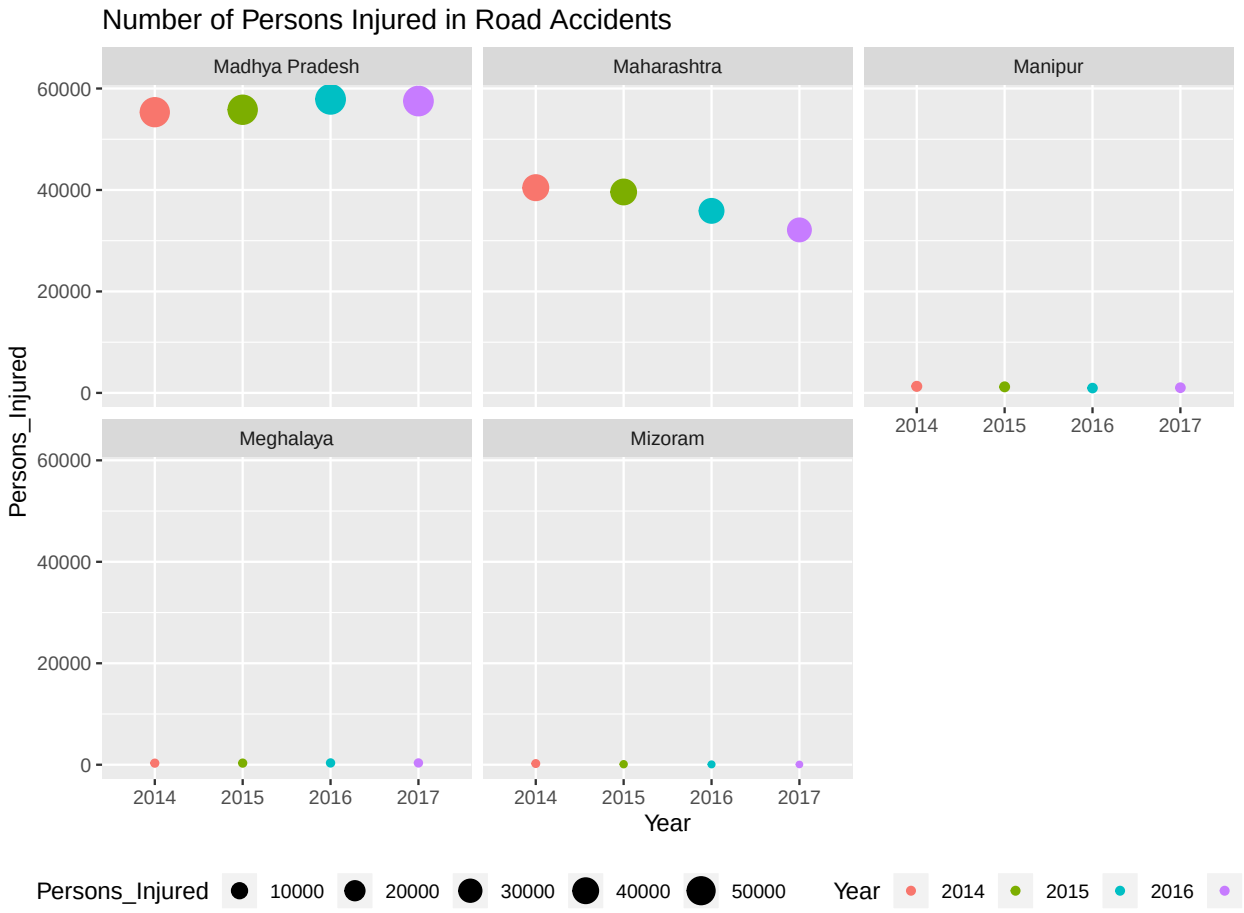


Figure 4: A simple graphic produced with **ggplot2**

```
## Regions defined for each Polygons
```

Note : The `tidy()` method coerces the spatial data object into a data frame. In this process of coercing, we lose the attribute information of states object. To add it back use the `left_join` function from **dplyr** package.

```
library(dplyr)
# head(states_df, n = 2) # Investigate the tide-ied data
states$id <- row.names(states) #Allocate an id variable to sp data
# head(states@data, n = 2) # A check before we join
states_df <- left_join(states_df, states@data)
```

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

At this point we have a geocoded dataframe object `states_df` object containing location and attribute information of road accidents in India. It is now straightforward to produce a map with **ggplot2**.

```
map <- ggplot(data = states_df, aes(x = long, y = lat, group = group, fill = State.UT.wise.Total.Number
  geom_polygon() + # Plot the states
  coord_equal() + # fixed x & y scales
  labs(x = "longitude", y = "latitude", fill = "Number of Persons_Injured") + #Labels
  ggtitle("Road Accidents in India (2015)") + #title
  scale_fill_gradient(low = "white", high = "green") #colours
map #Figure 4
```

You can add more description from other packages as well. For example, you can add a direction symbol using "ggsn" package as following

```
map + ggsn::blank() + ggsn::north(states_df) #Output not shown.
```

Next we draw comparisons of accidental injuries over the years for all states, by faceting the plots on year basis.

```
# Preparing the Spatial Object
states_df <- left_join(states_df, accidents_df.m)
```

```
## Joining, by = "ST_NM"
```

```
ggplot(data = states_df,
       aes(x = long, y = lat, fill = Persons_Injured, group = group)) + #defining Variables
  geom_polygon() + #plot states map
  geom_path(colour = "black", lwd = 0.05) + # states borders
  coord_equal() + #fixed x and y scales
  facet_wrap(~ Year) + # one plot per year
  scale_fill_gradient(low = "white", high = "green",
                      name = "No. of Persons Injured") + # legend options
  theme(axis.text = element_blank(), # remove axis lables
        axis.title = element_blank(), # remove axis titles
        axis.ticks = element_blank()) # remove axis ticks
```

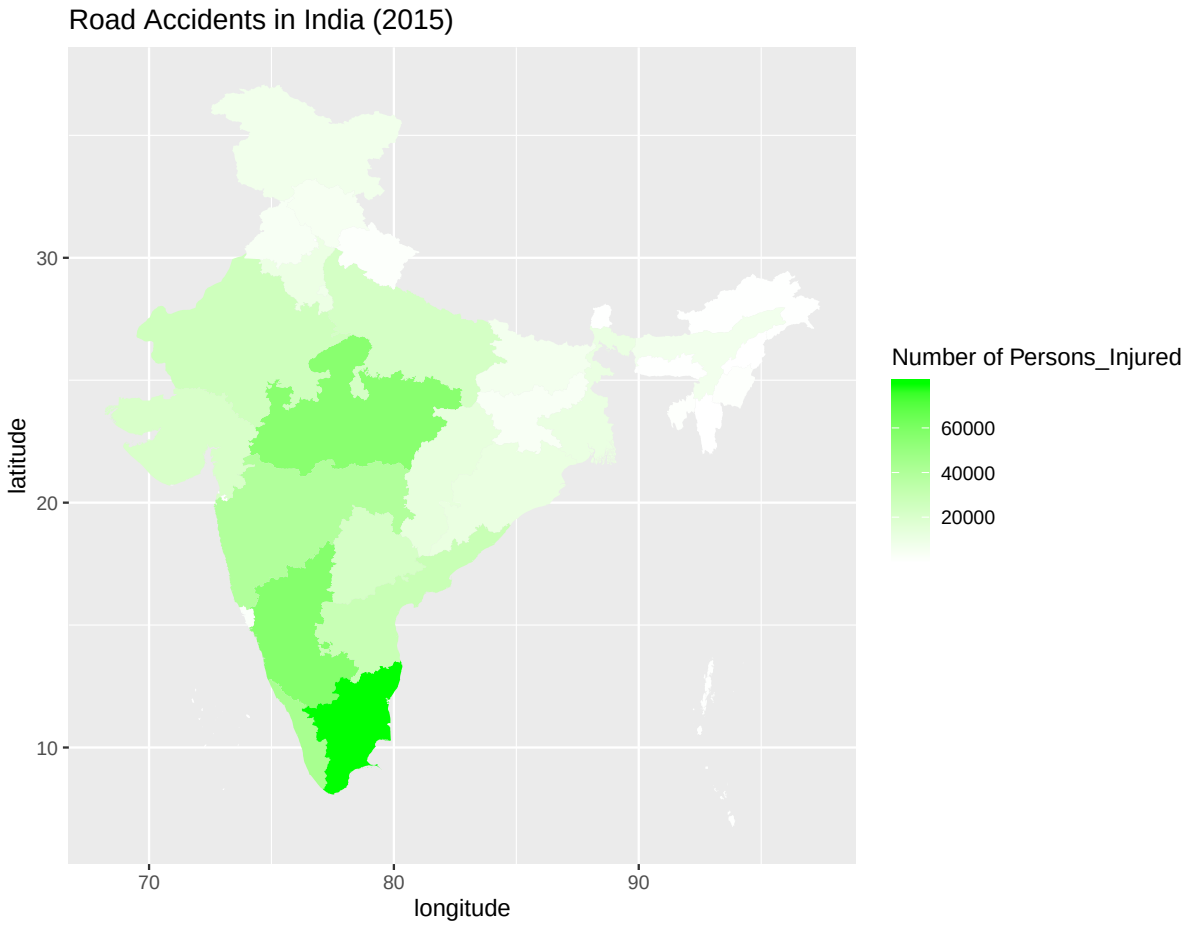


Figure 5: Map of Number of persons injured due to Road Accidents in India

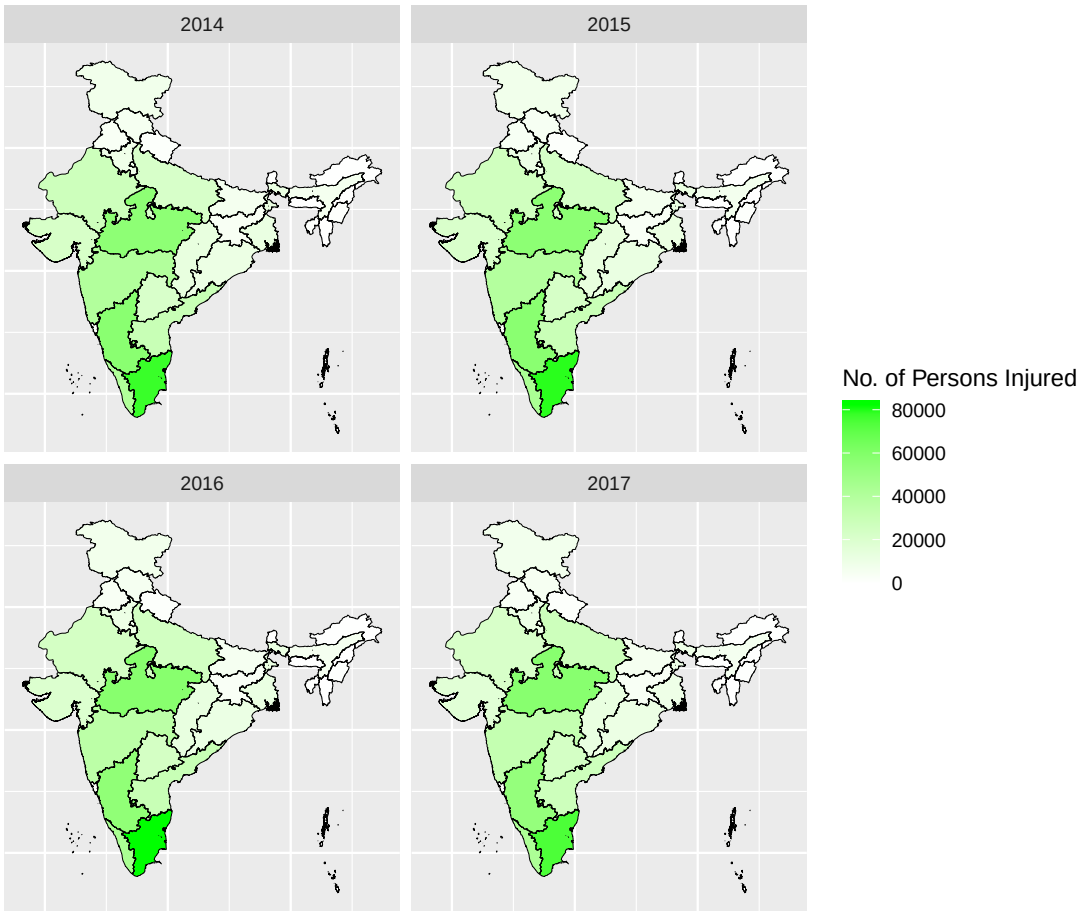


Figure 6: Figure depicting the number of persons injured in India due to road accidents over the years

tmap

In this section, we will look at another map making tool in R, called **tmap**. **tmap** was created to overcome some of the limitations of base graphics and **ggmap**. A concise introduction to tmap can be accessed (after the package is installed) by using `vignette ("tmap-getstarted")` function.

The following section and upcoming sections are inspired, and reproduced from An Introduction to Spatial Data Analysis in R. From here onwards, we use *Camden's Output Area, Census Estimates* dataset, provided at CDRC website.

I. Load the Shapefiles We begin with importing the camden open area shapefile.

```
Output.Areas <- readOGR(dsn = "Data/Camden_oa11/Camden_oa11.shp")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "G:\Mayur\IIIT Bangalore\Docs\Reading Elective-GIS\GISTutorial\Data\Camden_oa11\Camden_oa11.s
## with 749 features
## It has 1 fields
```

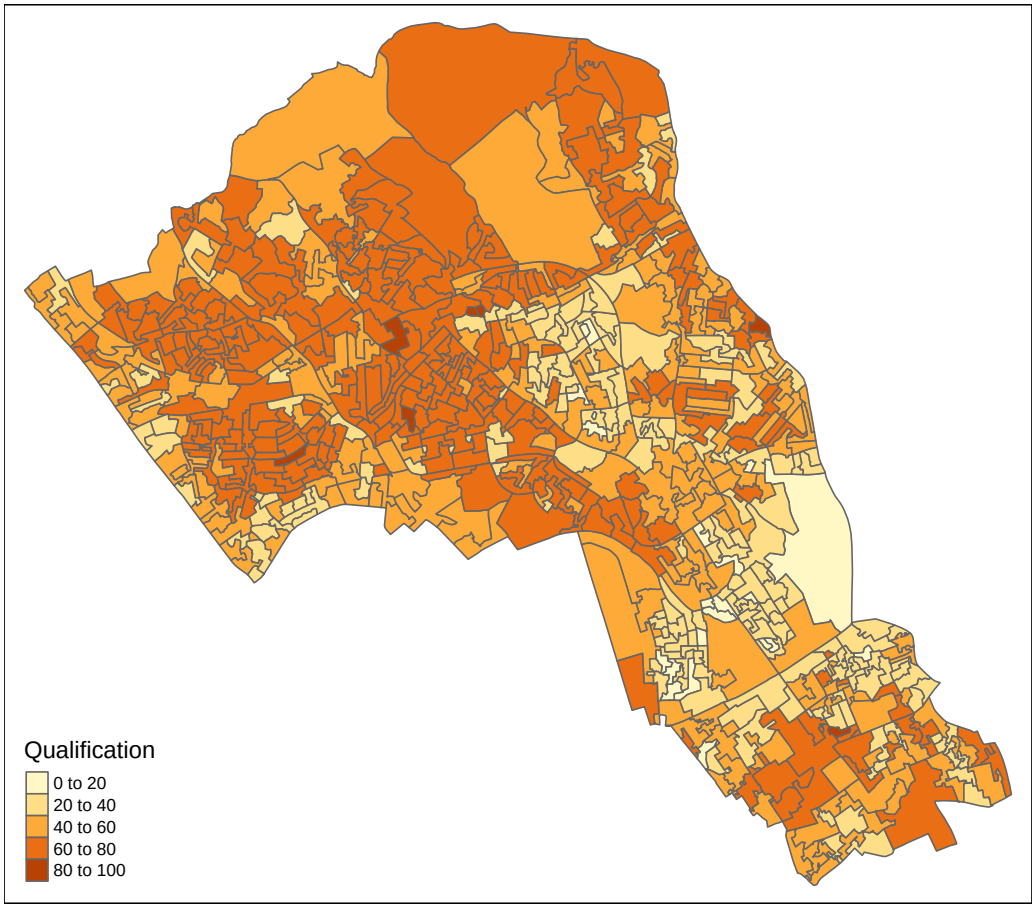
II. Join with Census Data Load the census data from `camden_census_data.csv` and join with camden open areas shapefile.

```
Census.Data <- read.csv("Data/camden_census_data.csv")
OA.census <- merge(Output.Areas, Census.Data, by.x = "OA11CD", by.y = "OA")
```

Next we need to have correct CRS arguments for projections. Most data from the UK is projected using the British National Grid codes, (EPSG:27700/OSGB36/WGS84). However, in this case - the shapefile has correct CRS arguments for Projections, and doesn't need to be specified explicitly.

III. Create the Map Now we are in position to plot the basic maps using **tmap**.

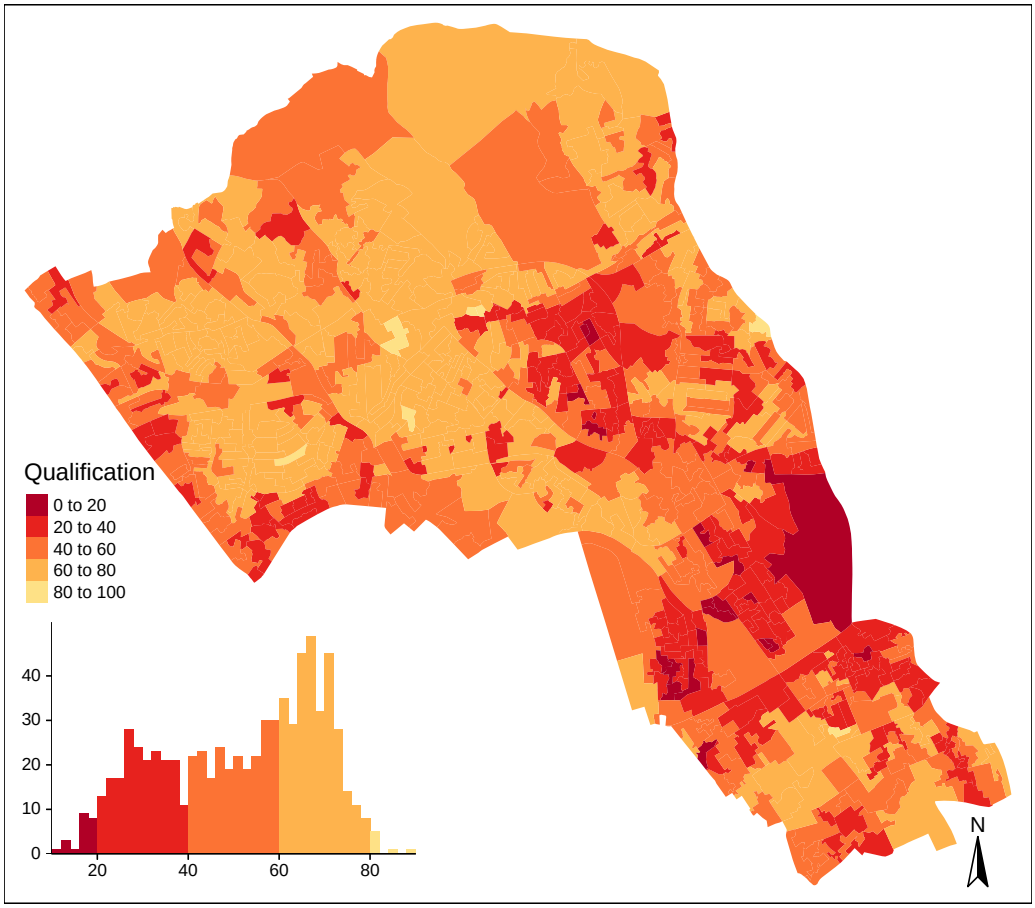
```
# load the pacakges
library(tmap)
qtm(OA.census, fill = "Qualification") #Output not shown
```



The `qtm()` from **tmap** draws a quick plot, with legend. The drawn map, gives us an insight to *qualified* youth percentage in Camden. **tmap** package provides many more functionalities for plotting, which are relatively simple to use when compared with other packages like **ggmap**.

We explore the **tmap** package a little more in details and demonstrate its map drawing capabilities with more statistical infomation.

```
# library(RColorBrewer) # To explore predefined colour ramps using display.brewer.all()
# display.brewer.all()  # To view all color pallets available
tm_shape(OA.census) + tm_fill("Qualification", palette = "-YlOrRd",
                              style = "pretty", #specify color intervals
                              legend.hist = TRUE) + #add histogram legend
  tm_compass()  #Add north arrow
```



The map drawn above, is similar to one drawn using `qtm()`. But, it is more detailed and descriptive about the *qualified* youth in camden. While plotting with **tmap**, we specify color intervals using `style` parameter. Each `style` parameter has an impact on how data is visualised. You can specify following color intervals - - `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 the 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)

5. Spatial Analysis in R - A Deeper Look

In this section, we will run a kernel density estimation(KDE) in R. KDE 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 bases on the number of points within a given distance badnwidth. We will also look at creating and mapping of raster shapefiles.

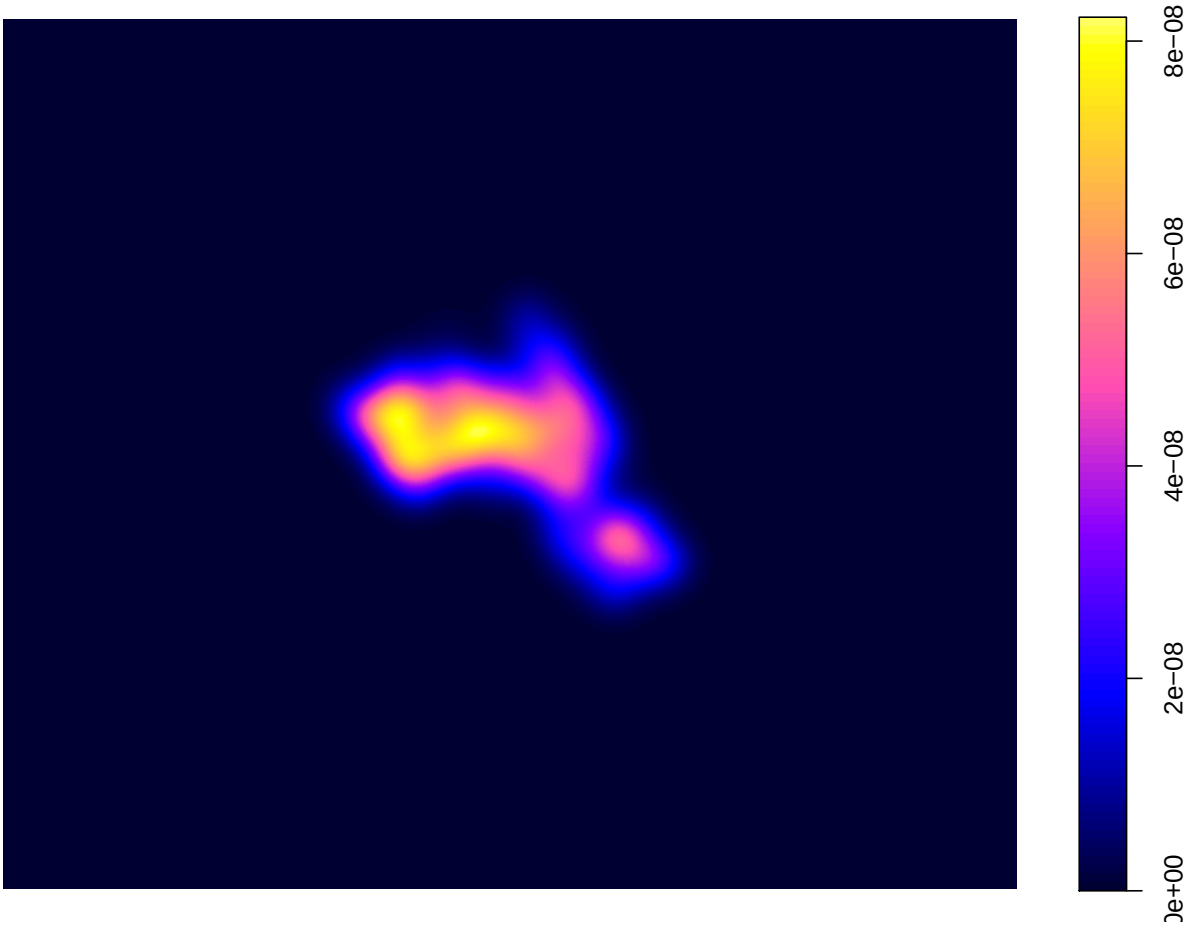
I. Data Preparation. Download the House Sales Shapefile, from CDRC website

```
# load house point files
House.Points <- readOGR(dsn = "Data/Camden_house_sales/Camden_house_sales.shp")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "G:\Mayur\IIIT Bangalore\Docs\Reading Elective-GIS\GISTutorial\Data\Camden_house_sales\Camde
## with 2547 features
## It has 4 fields
```

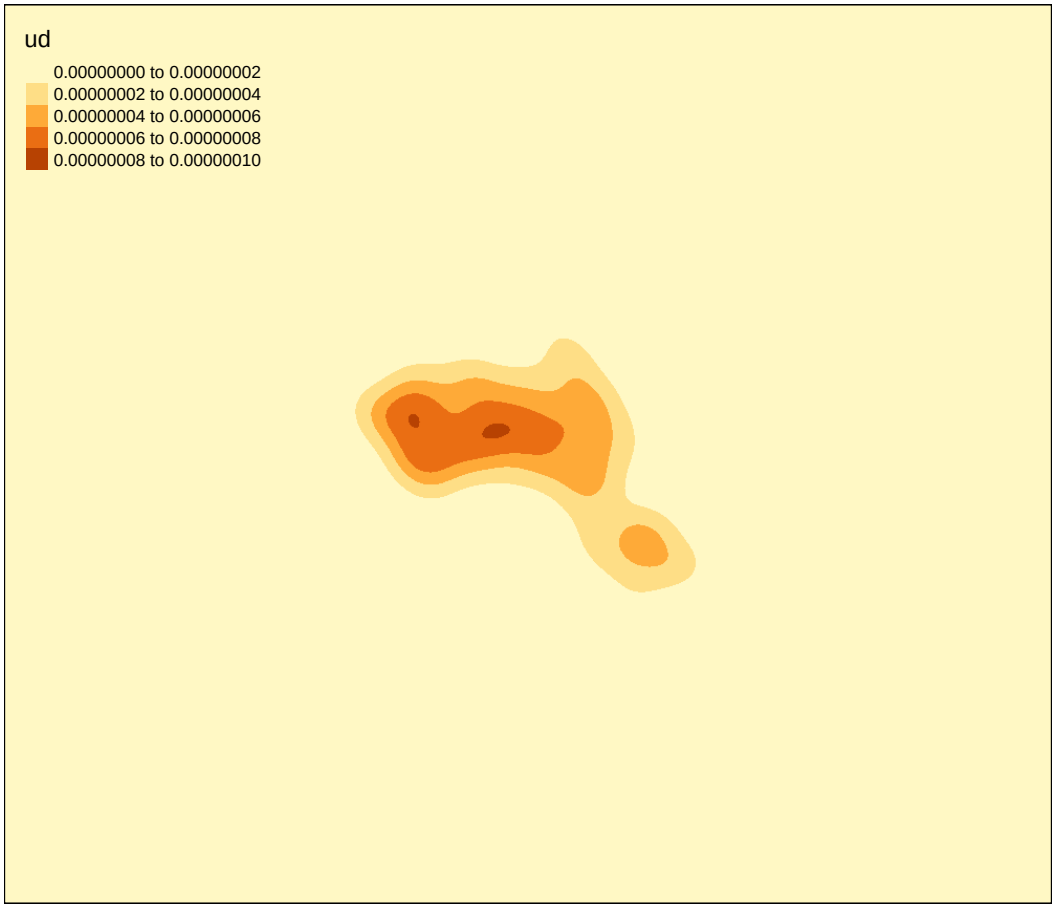
II. Data Exploration The following code run a kernel density estimation for Land Registry house price data. There are several different ways to run this through R, we will use the functions available from the `adehabitatHR` package.

```
# load the 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)
```



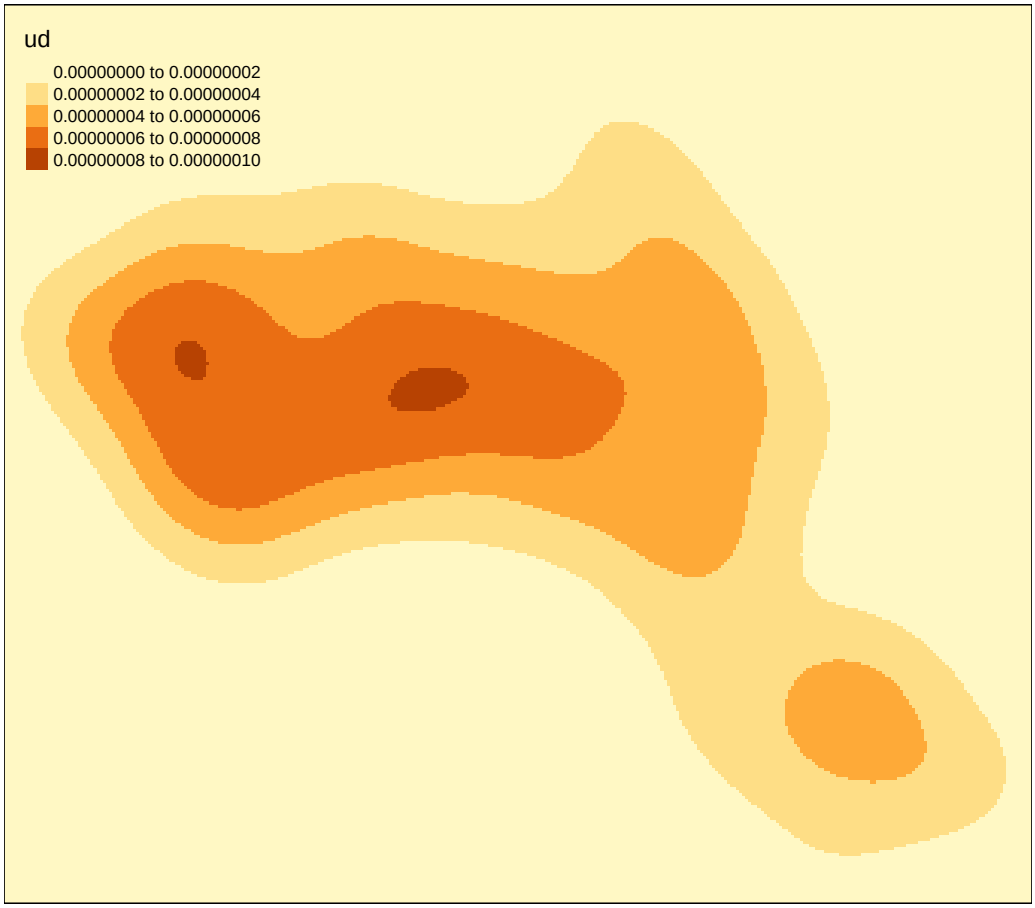
III. Analyse the Data To map the raster in tmap, we first need to ensure it has been projected correctly.

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

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



As evident, the raster includes a lot of empty space, we next 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 <- bbox(Output.Areas)

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



Next we 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 = "Blues") +
  tm_shape(Output.Areas) + tm_borders(alpha = 0.3, col = "white") +
  tm_layout(frame = FALSE)
```

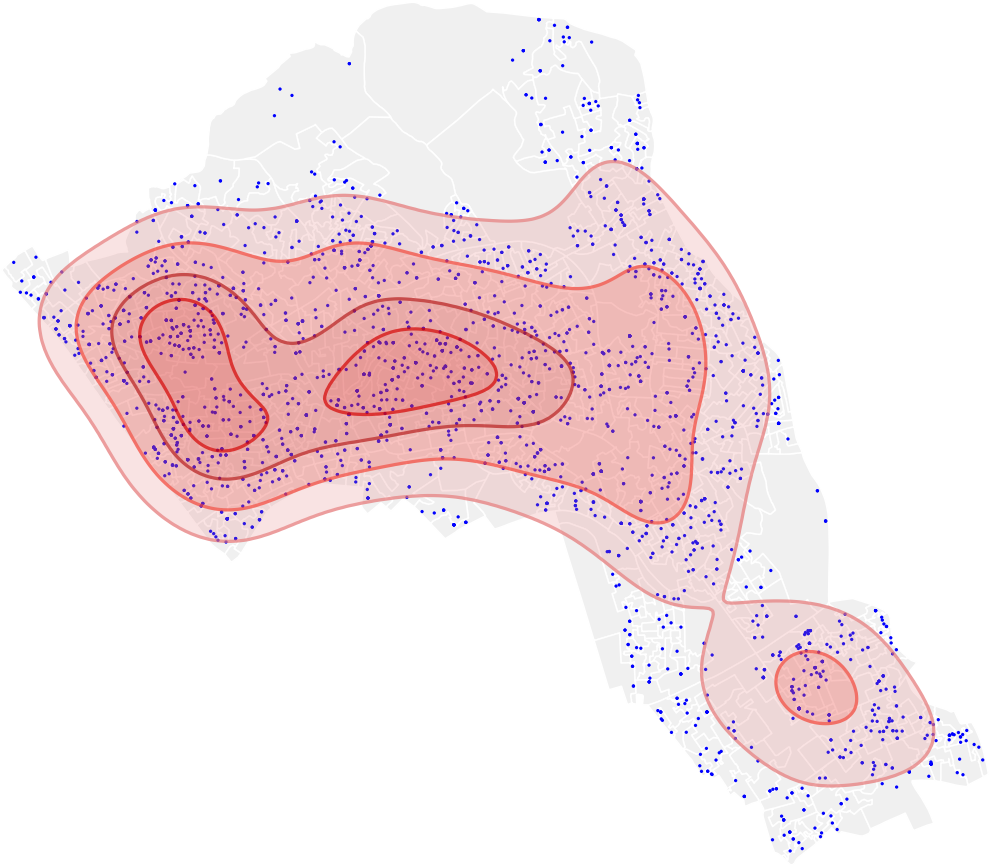
From kernel density estimates, we can also create catchment boundaries to identify homeranges.

```
# compute homeranges for 75%, 50%, 25% and 10% of points,
# note : objects are returned as spatial polygons data frames
range75 <- getverticeshr(kde.output, percent = 75)
range50 <- getverticeshr(kde.output, percent = 50)
range25 <- getverticeshr(kde.output, percent = 25)
range10 <- getverticeshr(kde.output, percent = 10)
```

After we have calculated homeranges, we can plot them together in tmap, Remember, tmap (like ggmap) combines multiple layers, and layer at the bottom of the list is the last one to be printed. Thus last layer printed, appears at the front of the graphic.

```
# the code below creates a map of several layers using tmap
tm_shape(Output.Areas) + tm_borders(alpha=.8, col = "white") +
    tm_fill(col = "#f0f0f0") +
  tm_shape(House.Points) + tm_dots(col = "blue") +
  tm_shape(range75) + tm_borders(alpha = .7, col = "#e57373", lwd = 2) +
    tm_fill(alpha =.2, col = "#e57373") +
  tm_shape(range50) + tm_borders(alpha = .7, col = "#f44336", lwd = 2) +
    tm_fill(alpha =.2, col = "#f44336") +
  tm_shape(range25) + tm_borders(alpha = .7, col = "#b71c1c", lwd = 2) +
    tm_fill(alpha =.1, col = "#b71c1c") +
```

```
tm_shape(range10) + tm_borders(alpha = .7, col = "#d50000", lwd = 2) +
  tm_fill(alpha =.1, col = "#d50000") +
tm_layout(frame = FALSE)
```



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 seperate ranges to compare their distributions - i.e. two species of animals. You can also infer areas of high risk locations, in a situation of public disorder in a city.

References

Lovelace, R., Cheshire, J. and Oldroyd, R. (2017). Introduction to visualising spatial data in R. ebook[Accessed 18 May 2017].

Sadler, J. (2019). Introduction to GIS with R. [online] Jesse Sadler. Available at: https://www.jessesadler.com/post/gis-with-r-intro/ [Accessed 25 Dec. 2019].

Brunsdon, C. and Comber, L. (2015). An Introduction to R for Spatial Analysis and Mapping. 1st ed.

Lansley, G. and Cheshire, J. (2016). An Introduction to Spatial Data Analysis andVisualisation in R. ebook [Accessed 28 Dec. 2019].


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