Introduction to visualising spatial data in R

Prof. Uttam Kumar, Mayur Rajwani

2019-12-25. See github.com/rajwanimayur/SpatialAnalysis.pdf for latest version

Contents

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.

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")
# 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)
```

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

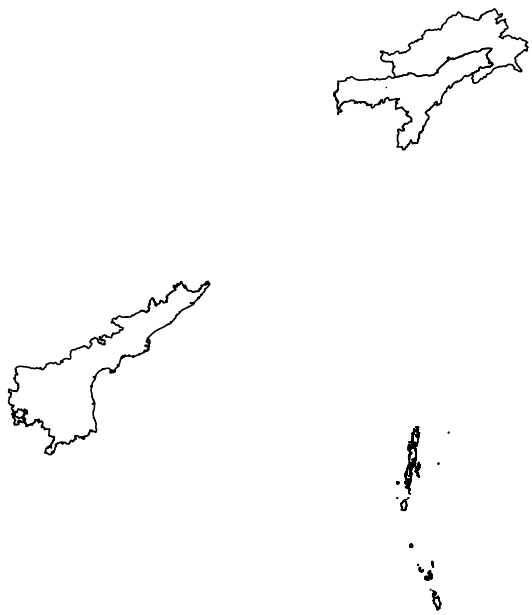


Figure 1: States whose names starts with A

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.

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

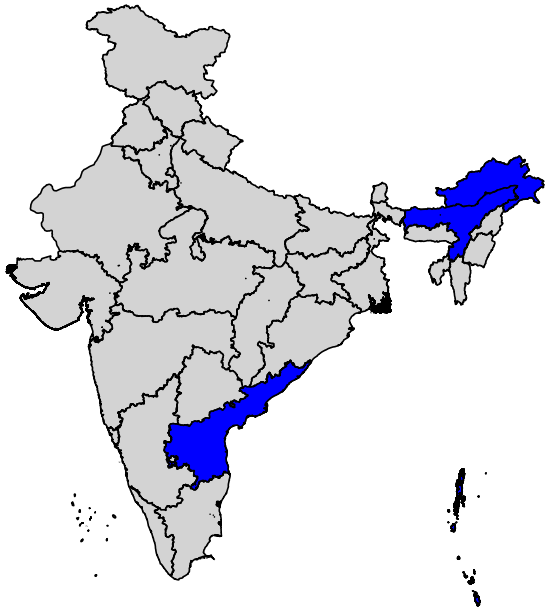


Figure 2: Map of India, highlighting the states whose name starts with ‘A’

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
```

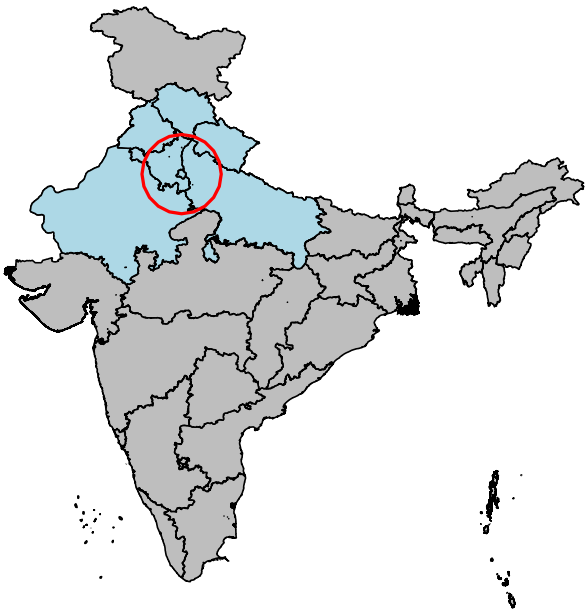
```
## CRS arguments:
##  +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
```

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.



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. Before we delve into the details, we will re-load our "states" spatial object from "states_raw"

```
states <- states_raw
```

The non-spatial object we are going to join our `states` object contains States/UTs road accidents records. 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.

```
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 Interactive Maps in R, using ggmap

ggplot2

In this section we create slightly different plots in R, using **ggplot2** package. The pacakage is implentation 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. It makes plotting of complex plots easier by providing a programmatic interface for variable specification, displaying of plots and other general visual properties. Thus **ggplot2** requires minimal changes if there is an underlying data change or we decide to change from bar plot to scatter plot.

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

We now begin exploring **ggplot2** with a scatter plot for attribute data in `states` object. We first set up the data we need to plot. For this, we first subset the data to few columns. Here we select variables that specify the "Total Number of Persons Injured" in a calendar year. Next we rename those columns to represent only year number, since our dataset now represent only single feature, namely "Total Number of Persons Injured"

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

Next we `tidy` the data and `'gather'`(from **tidyr** package) columns values into rows. This creates a single variable called **Year**, values of which are stored in another variable **Persons_Injured**. This allows us to create a single tuple of form <ST_NM, Year, Person_Injured> representing all the information contained in our dataset. It makes visualization of our data easy.

```
library(tidyr)
df.m <- gather(df, key = "Year", value = "Persons_Injured", colnames(df)[2:5])
```

We will require this dataframe in latter section, hence We save our `df.m` object for latter use.

```
saveRDS(object = df.m, file = "Data/Statewise_Yearly_Road_Accidents_India.Rds")
```

Now we are ready for setting up our `ggplot` object. The `ggplot` object takes `df.m` (a dataframe object) as input. We then specify the type of plot we want, in our case we draw a scatter plot (using `geom_point()` function). We also define the aesthetics, i.e, variables of graph. Here we call variables *Year* and *Persons_Injured* as variables x and y of our graph. Next we plot different states' scatter plot, side by side, using `facet_wrap` function.

```
library(ggplot2)
p1 <- ggplot(data = df.m[startsWith(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. Thus we can plot ggplot object(namely `p1`) directl, or add more layers like `title`, `theme` of graph. Thus, we see the ease of use that **ggplot2** package provides.

We are now in position to plot spatial objects using **gmap** package, which is based on **ggplot2** package.

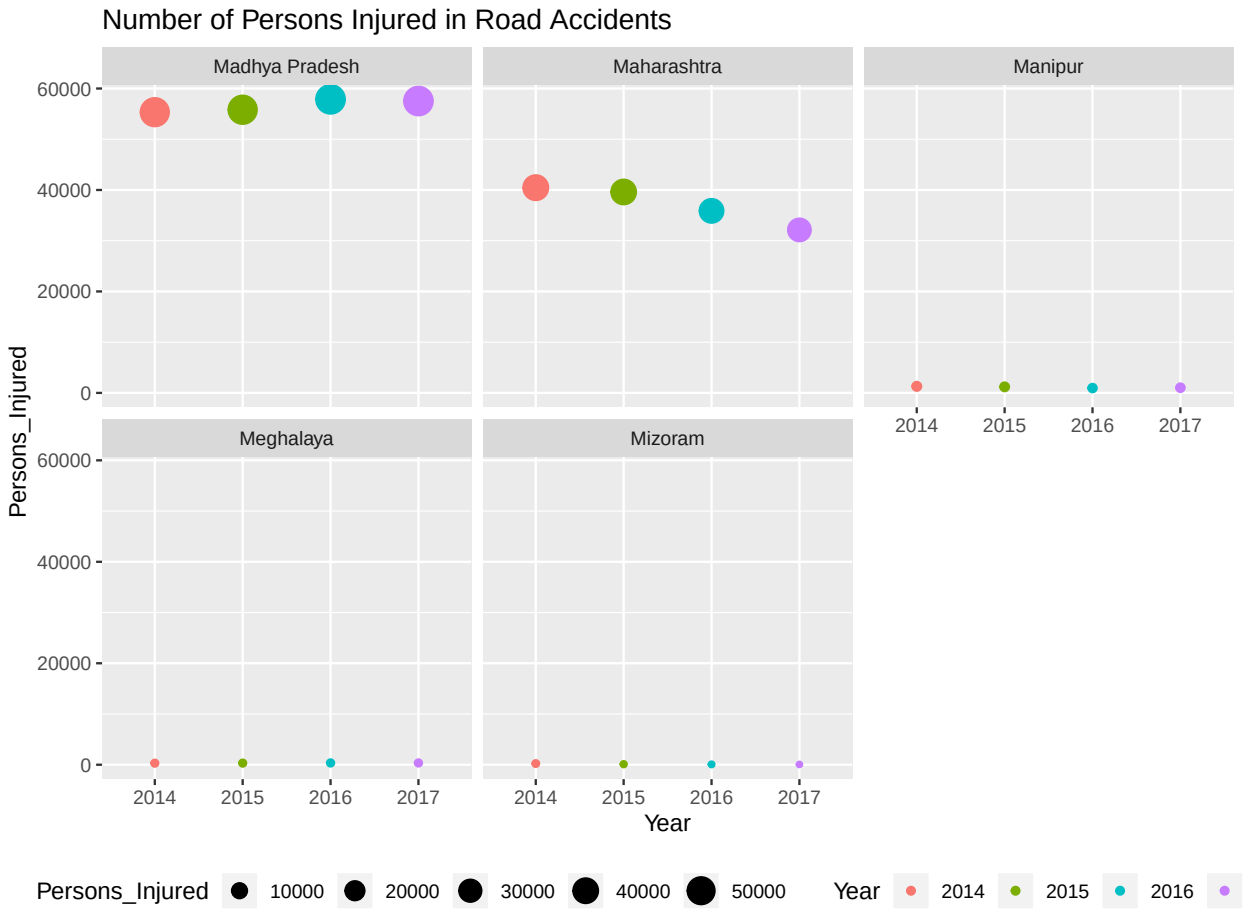


Figure 3: A simple graphic produced with **ggplot2**

ggmap

In the following steps we will create a map to show the total number of persons injured in road accidents in India(instead of few selected states). This will require the usage of **ggmap** package, which is based on the **ggplot2** package. Like **ggplot2**, **ggmap** package also requires 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. The generic plot() functions then, can use `Spatial*` objects directly.

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

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

Note : The `tidy()` method coerces the spatial data object into a data frame. In this process of coercing, we lost the attribute information of states object. We add it back using 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"
```

Now our `states_df` object contains geospatial cooridnates information alongside the 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
```

We can add more description from other packages. As an example, you can add a direction symbol using "ggsn" package as following

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

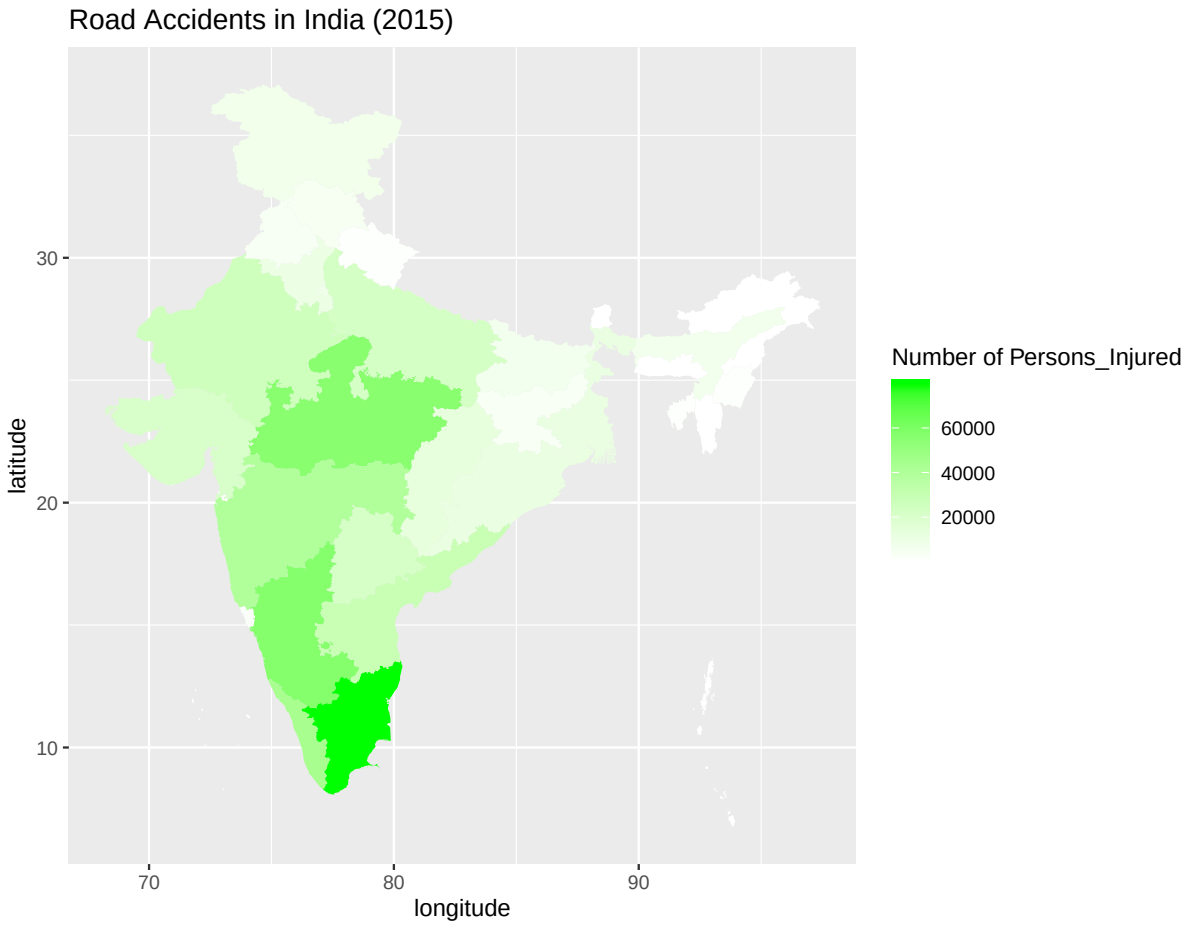


Figure 4: Map of Number of persons injured due to Road Accidents in India

5. Analysing Spatial Data

In the previous sections, we have learnt the nuts and bolts of spatial data analysis. Its now time to create put things together and create a bigger picture, conveying more information. So far we have drawn plots and graphs with limited information to display. In the **ggplot2** example, we drew the comaprisons of number of persons injured in road accidents among few states. The infograph was incapable to plot such a graph for every state, due to plotting space constraint. Hence we were restricted to fewer states to avoid clutter. Later on we draw the map to represent information for all the states, but for a single year.

We are now in position to do some analysis, leveraging the power of spatial data. We make use of faceting for maps, and draw comparisons over the years for all states - side by side. As a first step - we need to prepare the data for faceting. Remember we saved our `df.m` object in section 4. Its time now to fetch it back.

```
# load the saved R object
accidents_df<- readRDS(file = "Data/Statewise_Yearly_Road_Accidents_India.Rds")
```

Next we prepare our data for plotting. This involves converting the spatial object into a data frame object, and join it with non-spatial data frame containg more attributes. Do not forget to manually correct the typographical errors.

```
# Creating a data frame for plotting
states <-  states_raw

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

```
## # A tibble: 6 x 7
##    long   lat order hole  piece group id
##   <dbl> <dbl> <int> <lgl> <chr> <chr> <chr>
## 1  92.9  12.9     1 FALSE 1     0.1   0
## 2  92.9  12.9     2 FALSE 1     0.1   0
## 3  92.9  12.9     3 FALSE 1     0.1   0
## 4  92.9  12.9     4 FALSE 1     0.1   0
## 5  92.9  12.9     5 FALSE 1     0.1   0
## 6  92.9  12.9     6 FALSE 1     0.1   0
```

```
states@data <- edit(states@data)
states$id <-  row.names(states)
head(states@data)
```

```
##                        ST_NM id
## 0 Andaman & Nicobar Islands  0
## 1         Arunachal Pradesh  1
## 2                     Assam  2
## 3                     Bihar  3
## 4                Chandigarh  4
## 5              Chhattisgarh  5
```

```
states_df <- left_join(states_df, states@data)
head(states_df)
```

```
## # A tibble: 6 x 8
##    long   lat order hole  piece group id    ST_NM
##   <dbl> <dbl> <int> <lgl> <chr> <chr> <chr> <chr>
## 1  92.9  12.9     1 FALSE 1     0.1   0     Andaman & Nicobar Islands
## 2  92.9  12.9     2 FALSE 1     0.1   0     Andaman & Nicobar Islands
## 3  92.9  12.9     3 FALSE 1     0.1   0     Andaman & Nicobar Islands
## 4  92.9  12.9     4 FALSE 1     0.1   0     Andaman & Nicobar Islands
## 5  92.9  12.9     5 FALSE 1     0.1   0     Andaman & Nicobar Islands
## 6  92.9  12.9     6 FALSE 1     0.1   0     Andaman & Nicobar Islands
```

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

We now plot the data and see the results.

```
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
```

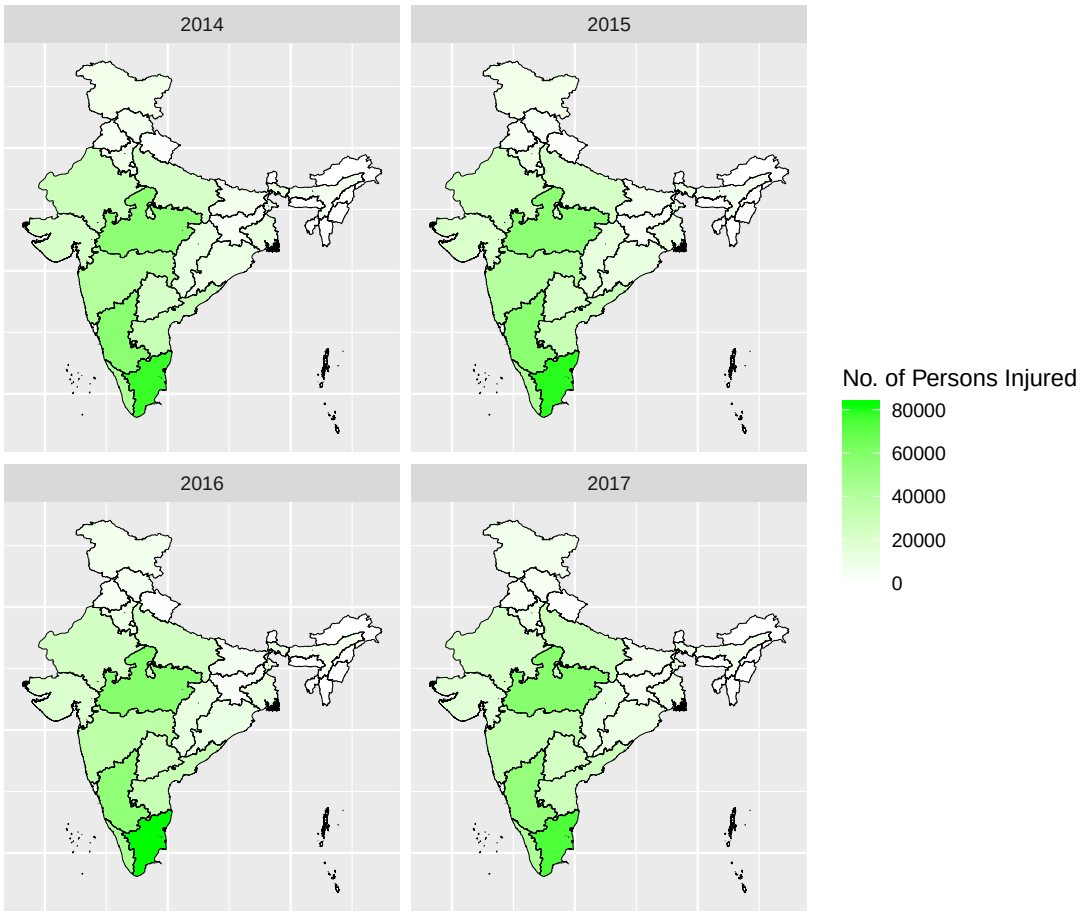


Figure 5: Figure depicting the number of persons injured in India due to road accidents over the years

References

Lovelace, R., Cheshire, J. and Oldroyd, R. (2017). Introduction to visualising spatial data in R. [ebook] Available at: https://github.com/Robinlovelace/Creating-maps-in-R [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.