India Spatial Analysis

Dr. Jeffrey Strickland

8/28/2018

Key Libraies for Spatial Analysis

- ggmap: extends the plotting package ggplot2 for maps
- rgdal: R's interface to the popular C/C++ spatial data processing library gdal
- rgeos: R's interface to the powerful vector processing library geos
- maptools: provides various mapping functions
- dplyr and tidyr: fast and concise data manipulation packages
- tmap: a new packages for rapidly creating beautiful maps
- install.packages(x)
- install.packages(c("rgdal", "maptools", "dplyr", "tidyr", "tmap", "tmaptools", "rgeos"))

Loading Libraries

```
library(maptools)

library(dplyr)

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

library(rgeos)

library(ggplot2)
library(ggmap)
library(raster)

library(DT)
```

Import India shape file (.SHP)

The shapefile format is a popular geospatial vector data format for geographic information system (GIS) software. It is developed and regulated by Esri as a (mostly) open specification for data interoperability among Esri and other GIS software products. The shapefile format is a digital vector storage format for storing geometric location and associated attribute information. This format lacks the capacity to store topological information. However, due to its simplicity, it is now possible to read and write geographical datasets using the shapefile format with a wide variety of software, including R.

The term "shapefile" is quite common but is misleading since the format consists of a collection of files with a common filename prefix, stored in the same directory. The three mandatory files have filename extensions .shp, .shx, and .dbf. The actual shapefile relates specifically to the .shp file, but alone is incomplete for distribution as the other supporting files are required.

Mandatory files

.shp — shape format; the feature geometry itself

.shx — shape index format; a positional index of the feature geometry to allow seeking forwards and backwards quickly

.dbf — attribute format; columnar attributes for each shape, in dBase IV format

The files necessary for performing the operations contained in this paper are located at https://github.com/stricje1/VIT_University/blob/master/Data_Analytics_2018/data/INDIA_geo.zip. It is best to download these files to your local machine and then configure your path in R to your computer.

```
mydir<-"C:/Users/jeff/Documents/Crime Analysis/India_data"
ind <- readOGR(dsn = mydir, layer = "INDIA")

## OGR data source with driver: ESRI Shapefile
## Source: "C:\Users\jeff\Documents\Crime Analysis\India_data", layer:
"INDIA"
## with 35 features
## It has 1 fields
```

Import India Population Grid Files (.GRD)

The first file we are going to load into R Studio is the “INDIA” shapefile located in the “data” folder of the project. It is worth looking at this input dataset in your file browser before opening it in R.

```
ind_pop<- raster("C:/Users/jeff/Documents/Crime
Analysis/India_data/ind_pop.grd")
ind_msk_pop<- raster("C:/Users/jeff/Documents/Crime
Analysis/India_data/ind_msk_pop.grd")
```

Comments on Imported Data

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 “**ind**”; short for London. readOGR is a function which accepts two arguments: **dsn** which stands for “data source name” and specifies the directory in which the file is stored, and layer which specifies the file name (note that there is no need to include the file extension `.shp`). The file we assigned to the ind object contains the population of London Boroughs in 2001 and the percentage of the population participating in sporting activities.

Examining the Output

Look at the output created (note the table format of the data and the column names). There are two important symbols at work in the above block of code: the `@` symbol in the first line of code is used to refer to the data slot of the `ind` object. The `$` symbol refers to the `Partic_Per` column (a variable within the table) in the data slot, which was identified from the result of running the first line of code.

Some Comments about Spatial Objetcs

Spatial objects like the ind object are made up of several different slots, the key slots being `@data` (non-geographic attribute data) and `@polygons` (or `@lines` for line data). The data slot can be thought of as an attribute table and the geometry slot is the polygons that make up the physical boundaries. Specific slots are accessed using the `@ symbol`. We analyze the INDIA object with some basic commands.

Shapefile Structure

It is a data frame, `SpatialPolygonsDataFrame`, bearing this structure:

```
structure(ind)
```

```
## class       : SpatialPolygonsDataFrame
## features    : 35
## extent      : 68.1202, 97.41516, 6.754256, 37.13564  (xmin, xmax, ymin,
##               ymax)
## coord. ref. : +proj=longlat +ellps=WGS84 +no_defs
## variables   : 1
## names       :                       ST_NAME
## min values  : ANDAMAN AND NICOBAR ISLANDS
## max values  :                  West Bengal
```

All shapefiles have at least two slots, a bounding box and coordinate system (CRS). Using `str(ind@polygons)`, we get a stream of polygon slots, which show five slots (there are about 800 of these or 4000 slots).

The Data Slots

First Slot for Data Object

In the shape file structure, `data` occupies the first slot, which consist of 35 factors of State Names for India.

`str(ind@data)` yields:

```
## 'data.frame':       35 obs. of  1 variable:
##  $ ST_NAME: Factor w/ 35 levels "ANDAMAN AND NICOBAR ISLANDS",..: 1 2 3 4
##             5 6 7 8 9 10 ...
```

Second Slot for Polygons Object

The second slot is for `polygons`. Things to note are: (1) labpt or the centroid of the polygon, given by longitude and latitude; (2) the area of the polygon; (3) a logical indicating a hole of ring as the outer boundary; (4) a ring direction (+ or -); and (5) a numeric array with two columns of ordered coordinates. The bulk of the India shapefile is comprised of the polygon data.

`str`(ind`@`polygons) prints out more data than we could hope to read, so we included on piece below.

```
$ :Formal class 'Polygons' [package "sp"] with 5 slots
  .. ..@ Polygons :List of 8
  .. .. ..$ :Formal class 'Polygon' [package "sp"] with 5 slots
  .. .. .. .. ..@ labpt  : num [1:2] 76.1 19.4
  .. .. .. .. ..@ area   : num 26.4
  .. .. .. .. ..@ hole   : logi FALSE
  .. .. .. .. ..@ ringDir: int 1
  .. .. .. .. ..@ coords : num [1:32323, 1:2] 80.4 80.4 80.4 80.4 80.4 ...
  .. .. ..$ :Formal class 'Polygon' [package "sp"] with 5 slots
  .. .. .. .. ..@ labpt  : num [1:2] 72.8 19.2
  .. .. .. .. ..@ area   : num 1.11e-10
  .. .. .. .. ..@ hole   : logi TRUE
  .. .. .. .. ..@ ringDir: int -1
  .. .. .. .. ..@ coords : num [1:4, 1:2] 72.8 72.8 72.8 72.8 19.2 ...
  .. .. ..$ :Formal class 'Polygon' [package "sp"] with 5 slots
  .. .. .. .. ..@ labpt  : num [1:2] 72.8 19.3
  .. .. .. .. ..@ area   : num 2.83e-10
  .. .. .. .. ..@ hole   : logi TRUE
  .. .. .. .. ..@ ringDir: int -1
  .. .. .. .. ..@ coords : num [1:4, 1:2] 72.8 72.8 72.8 72.8 19.3 ...
  .. .. ..$ :Formal class 'Polygon' [package "sp"] with 5 slots
  .. .. .. .. ..@ labpt  : num [1:2] 72.8 19.2
  .. .. .. .. ..@ area   : num 4.35e-10
  .. .. .. .. ..@ hole   : logi TRUE
  .. .. .. .. ..@ ringDir: int -1
  .. .. .. .. ..@ coords : num [1:4, 1:2] 72.8 72.8 72.8 72.8 19.2 ...
  .. .. ..$ :Formal class 'Polygon' [package "sp"] with 5 slots
  .. .. .. .. ..@ labpt  : num [1:2] 72.9 19
  .. .. .. .. ..@ area   : num 5.45e-10
  .. .. .. .. ..@ hole   : logi TRUE
  .. .. .. .. ..@ ringDir: int -1
  .. .. .. .. ..@ coords : num [1:4, 1:2] 72.9 72.9 72.9 72.9 19 ...
  .. .. ..$ :Formal class 'Polygon' [package "sp"] with 5 slots
  .. .. .. .. ..@ labpt  : num [1:2] 72.8 19.6
  .. .. .. .. ..@ area   : num 1.45e-09
  .. .. .. .. ..@ hole   : logi TRUE
  .. .. .. .. ..@ ringDir: int -1
  .. .. .. .. ..@ coords : num [1:4, 1:2] 72.8 72.8 72.8 72.8 19.6 ...
  .. .. ..$ :Formal class 'Polygon' [package "sp"] with 5 slots
  .. .. .. .. ..@ labpt  : num [1:2] 72.8 19.2
  .. .. .. .. ..@ area   : num 1.02e-07
  .. .. .. .. ..@ hole   : logi TRUE
  .. .. .. .. ..@ ringDir: int -1
  .. .. .. .. ..@ coords : num [1:6, 1:2] 72.8 72.8 72.8 72.8 72.8 ...
  .. .. ..$ :Formal class 'Polygon' [package "sp"] with 5 slots
  .. .. .. .. ..@ labpt  : num [1:2] 72.9 19.3
  .. .. .. .. ..@ area   : num 0.000204
  .. .. .. .. ..@ hole   : logi FALSE
  .. .. .. .. ..@ ringDir: int 1
  .. .. .. .. ..@ coords : num [1:19, 1:2] 72.9 72.9 72.9 72.9 72.9 ...
```

```
  .. ..@ plotOrder: int [1:8] 1 8 7 6 5 4 3 2
  .. ..@ labpt    : num [1:2] 76.1 19.4
  .. ..@ ID       : chr "19"
  .. ..@ area     : num 26.4
```

Third slot for Plotting Order

This is an integer vector of `plotOrder` or the plotting order of the 800 polygons. **str**(ind@plotOrder) yields:

```
## int [1:35] 29 19 20 2 33 14 11 16 26 7 ...
```

Fourth Slot for Bounding Box Object

The fourth slot is for a `bbox` or bounding box or a 2x2 matrix with named dimensions of the max & min of the X and Y coordinates. The bounding box of the map produced by the INDIA shapefile is key slot, called by

str(ind@box):

```
##         min      max
## x 68.120198 97.41516
## y  6.754256 37.13564
```

Fifth Slot for Class CRS

The fifth slot is for the proj4string of class CRS. **str**(ind@proj4string) yields:

```
## CRS arguments: +proj=longlat +ellps=WGS84 +no_defs
```

Headers

The head function in the first line of the code above simply means "show the first few lines of data" (try entering head(ind@data), see ?head for more details).

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

```
##                        ST_NAME
## 0 ANDAMAN AND NICOBAR ISLANDS
## 1              Andhra Pradesh
```

Classes

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

```
sapply(ind@data, class)
```

```
##  ST_NAME
## "factor"
```

Explore Spatial Structure

- To explore ind object further, try typing

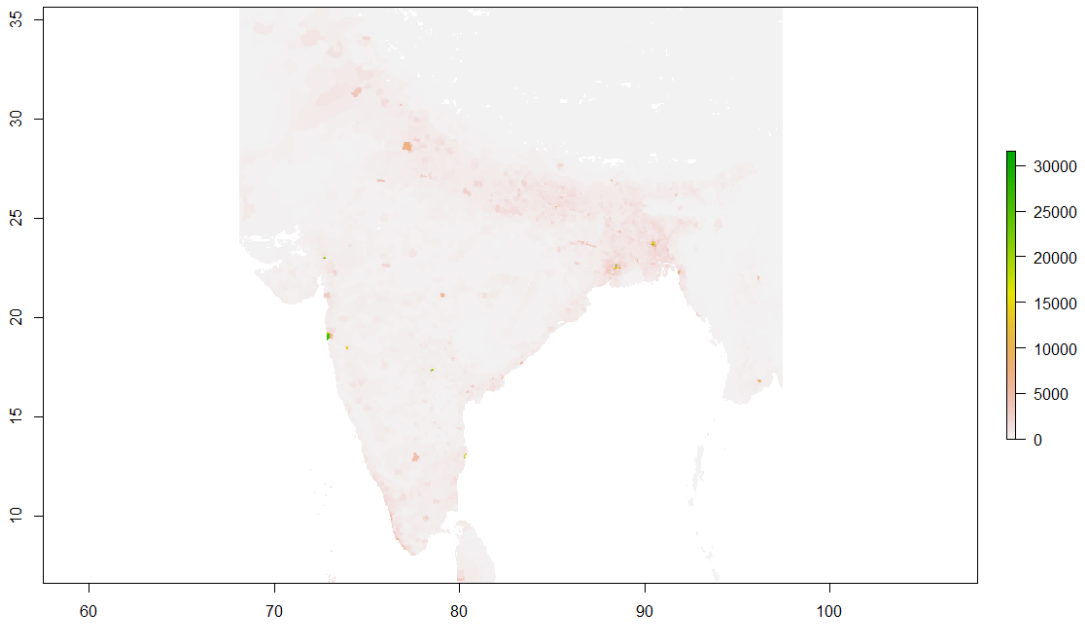
```
nrow(ind)

## [1] 35

ncol(ind)

## [1] 1

ind@proj4string

## CRS arguments: +proj=longlat +ellps=WGS84 +no_defs
```

Plotting Layers

Now we have seen something of the structure of spatial objects in R, let us look at plotting them. Note, that plots use the geometry data, contained primarily in the @polygons slot.

```
plot(ind)
```

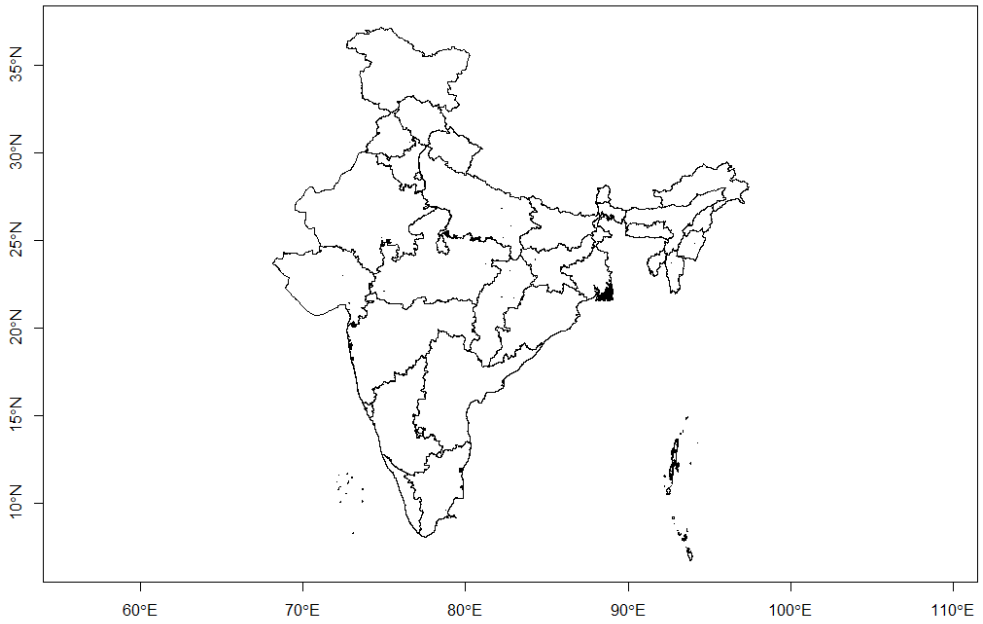
`plot(ind_pop)` prints the population mask layer for India. Perhaps you notice the file extension, `grd`, when we loaded `ind_msk_pop.grd` (`grd` is short for `grid`). Mumbai (formerly called Bombay) is India's most densely populated city. Located on the southwest coast of India, Mumbai appears as a small light-green dot on the population mask, indicating a population size of approximately 20,000 x 1000 = 20,000,000 or 20 million. In 2011, the Indian census record its population as 18.41 million.




```
raster::extent(ind)
```

```
## class       : Extent
## xmin        : 68.1202
## xmax        : 97.41516
## ymin        : 6.754256
## ymax        : 37.13564
```

```
plot(ind, xlim = c(68.1202, 97.41516), axes = TRUE)
```



Apply Plotting Enhancements

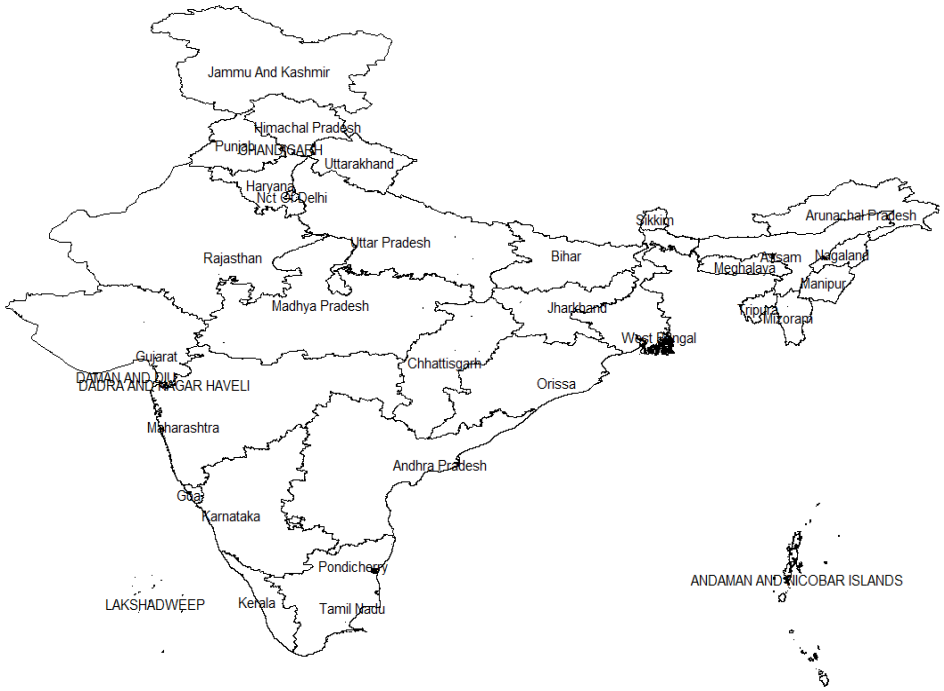
We will consider this as our base map.

```
map <- ggplot() + geom_polygon(data = ind, aes(x = long, y = lat, group =
      group), colour = "black", fill = NA)

## Regions defined for each Polygons

map + theme_void()
```

Plot Map with Enhancements

Now, we will begin enhancing our base map.

```
map <- ggplot() + geom_polygon(data = ind, aes(x = long, y = lat, group =
      group), colour = "black", fill = NA)

## Regions defined for each Polygons

cnames <- aggregate(cbind(long, lat) ~ id, data=shp_df, FUN=mean)
map + geom_text(data = cnames, aes(x = long, y = lat, label = id),
      size = 4) + theme_void()
```



Setup Bounding Box for Indian Map

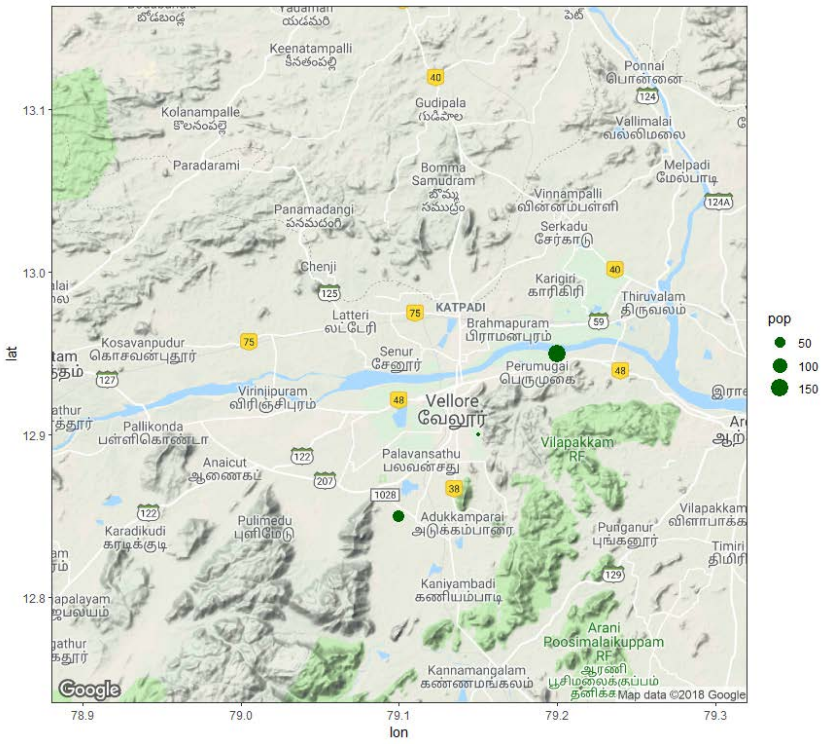
```
subscr<-data.frame(lon=c(79.1,79.15,79.2),lat=c(12.85,12.9,12.95),
      pop=c(58,12,150))
coordinates(subscr)<-~lon+lat
proj4string(subscr)<-CRS("+init=epsg:4326")
lon <- c(78.9,79.3)
lat <- c(12.7,13.2)
map1<-get_map(location = c(lon[1], lat[2], lon[2], lat[1]), zoom=11)

## Map from URL :
http://maps.googleapis.com/maps/api/staticmap?center=12.95,79.1&zoom=11&size=
640x640&scale=2&maptype=terrain&language=en-EN&sensor=false
```

Setup Bounding Box Plot

```
p <- ggmap(map1) +
      geom_point(data = as.data.frame(subscr),
      aes(x = lon, y = lat, size=pop),
            colour = "darkgreen") +
      theme_bw()
```

Plot Bounding Box Map

```
print(p)
```



Plot India Map with Grey Fill

This requires `rgeos` and will be a base-map for this section of work.

```
ind <- readOGR(dsn = mydir, layer = "INDIA")

## OGR data source with driver: ESRI Shapefile
## Source: "C:\Users\jeff\Documents\Crime Analysis\India_data",
## layer: "INDIA"
## with 35 features
## It has 1 fields

plot(ind, col = "grey")
```

Find Geographic centroids for Indian States

We will calculate centroids for India and each of its states. This will allow us to perform several tasks including dividing the country into quadrants and color-coding states.

```
cent_ind <- gCentroid(ind[ind$ST_NAME == "Madhya Pradesh",])
cent_ind

## class       : SpatialPoints
## features    : 1
## extent      : 78.28915, 78.28915, 23.54119, 23.54119  (xmin, xmax, ymin,
##               ymax)
## coord. ref. : +proj=longlat +ellps=WGS84 +no_defs
```

Plot with Fill and Borders

```
## OGR data source with driver: ESRI Shapefile
## Source: "C:\Users\jeff\Documents\Crime Analysis\India_data",
## layer: "INDIA"
## with 35 features
## It has 1 fields

## Warning in gBuffer(spgeom = cent_ind, width = 10000): Spatial object is

## not projected; GEOS expects planar coordinates
```

Selecting Quadrants for India

- The code below should help understand the way spatial data work in R.
- Find the centre of the India area

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

Test whether or not a coordinate is east or north of the centre

```
east <- sapply(coordinates(ind)[,1], function(x) x > lat)
north <- sapply(coordinates(ind)[,2], function(x) x > lng)
ind@data$quadrant[east & north] <- "northeast"
```

Test whether or not a coordinate is east or north of the centre

```
east <- sapply(coordinates(ind)[,1], function(x) x > lat)
south <- sapply(coordinates(ind)[,2], function(x) x < lng)
ind@data$quadrant[east & south] <- "southeast"
```

Test whether or not a coordinate is east or north of the centre

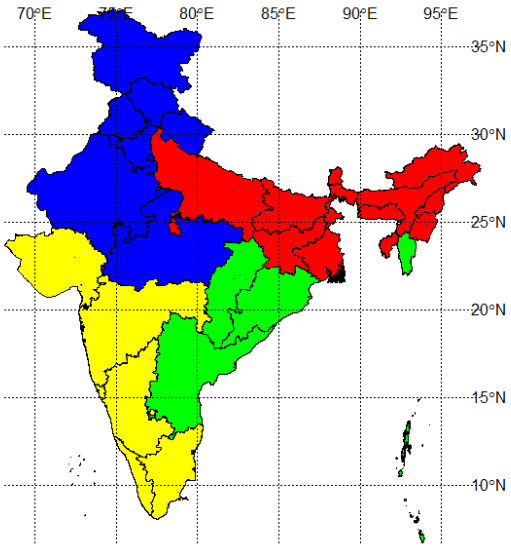
```
west <- sapply(coordinates(ind)[,1], function(x) x < lat)
north <- sapply(coordinates(ind)[,2], function(x) x > lng)
ind@data$quadrant[west & north] <- "northwest"
```

Test whether or not a coordinate is east or south of the centre

```
west <- sapply(coordinates(ind)[,1], function(x) x < lat)
south <- sapply(coordinates(ind)[,2], function(x) x < lng)
ind@data$quadrant[west & south] <- "southwest"
```

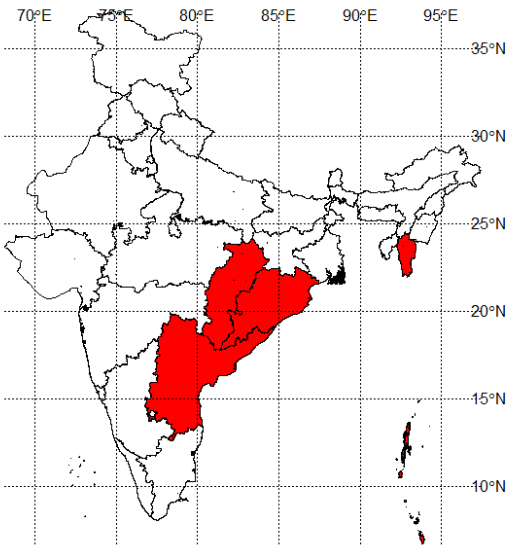
Plot Quadrants with Different Fill Collors

```
plot(ind)
plot(ind[east & north,], col = "red", add = TRUE)
plot(ind[east & south,], col = "green", add = TRUE)
plot(ind[west & north,], col = "blue", add = TRUE)
plot(ind[west & south,], col = "yellow", add = TRUE)
llgridlines(ind, lty= 3, side ="EN", offset = -0.5)
```



Plot SE Quadrants with Red Fill Collors

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



Creating New R Object

Alongside visualisation and interrogation, a GIS must also be able to create and modify spatial data. R's spatial packages provide a very wide and powerful suite of functionality for processing and creating spatial data. R objects can be created by entering the name of the class we want to make. Vector and `data.frame` objects for example, can be created as follows:

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

Check the class of these new objects using class():

```
class(vec)

## [1] "numeric"

class(df)

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

Creating New Spatial Data

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

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

Comment on Creating Spatial Data

We have just created a spatial points object, one of the fundamental data types for spatial data. (The others are lines, polygons and pixels, which can be created by SpatialLines,SpatialPolygons and SpatialPixels, respectively.) Each type of spatial data has a corollary that can accepts non-spatial data, created by adding DataFrame. SpatialPointsDataFrame(), for example, creates points with an associated data.frame. The

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

Add Data to an Existing Spatial Data File

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

```
class(sp1)
```

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

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

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

Projections: setting and transforming CRS in R

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

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

Comments on CRS Changes

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. Under this system 27700 represents the British N 'WGS84' (epsg:4326) is a very commonly used CRS worldwide.

Find EPSG Codes

The following code shows how to search the list of available EPSG codes and create a new version of ind in WGS84:3

```
proj4string(ind)
```

```
## [1] "+proj=longlat +ellps=WGS84 +no_defs"
```

Convert the Coordinates

spTransform converts the coordinates of ind into the widely used WGS84CRS.

```
CRS("+init=epsg:4326")
```

```
## CRS arguments:
##  +init=epsg:4326 +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84
## +towgs84=0,0,0
```

```
CRS("+init=epsg:3785")
```

```
## CRS arguments:
##  +init=epsg:3785 +proj=merc +a=6378137 +b=6378137 +lat_ts=0.0
## +lon_0=0.0 +x_0=0.0 +y_0=0 +k=1.0 +units=m +nadgrids=@null
## +no_defs
```

Change the EPSG

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

```
##      code     note                                          prj4
## 249  4326 # WGS 84          +proj=longlat +datum=WGS84 +no_defs
## 5311 4978 # WGS 84 +proj=geocent +datum=WGS84 +units=m +no_defs
```

```
ind84 <- spTransform(ind, CRS("+init=epsg:3785")) # reproject
ind84
```

```
## class       : SpatialPolygonsDataFrame
## features    : 35
## extent      : 7583106, 10844206, 753627.8, 4458031  (xmin, xmax, ymin,
ymax)
## coord. ref. : +init=epsg:3785 +proj=merc +a=6378137 +b=6378137 +lat_ts=0.0
+lon_0=0.0 +x_0=0.0 +y_0=0 +k=1.0 +units=m +nadgrids=@null +no_defs
## variables   : 2
## names       :                      ST_NAME,  quadrant
## min values  : ANDAMAN AND NICOBAR ISLANDS,  northest
## max values  :                  West Bengal, southwest
```

Saving CRS Formats

Now we've transformed ind into a more widely used CRS, it is worth saving it. R stores data effciently in .RData or .Rds formats. The former is more restrictive and maintains the object's name, so we use the latter

```
saveRDS(object = ind84, file = "C:/Users/jeff/Documents/Crime
Analysis/Creating-maps-in-R-master/data/ind84.Rds")
```

Reset INDIA Shapefile

To reaffirm our starting point, let's re-load the "INDIA" shapefile as a new object and plot it: Create new object called "ind" from "INDIA" shapefile

```
ind <- readOGR(dsn = mydir, "INDIA")

## OGR data source with driver: ESRI Shapefile
## Source: "C:\Users\jeff\Documents\Crime Analysis\India_data", layer:
"INDIA"
## with 35 features
## It has 1 fields

plot(ind) # Not shown
```

We are not showing the plot, but we can check that the 35 rows of data are still in the file.

```
nrow(ind)

## [1] 35
```

##Joing Non-Spatial and Spatial data

The non-spatial data we are going to join to the ind object contains records of population in India. This is stored in a comma separated values (.csv) file called "`India_pop_2011`".
If you open the file in a separate spreadsheet application first, we can see each row represents a single city's population. We are going to use a function called aggregate to aggregate the population at the city, ready to join to our spatial **`ind`** dataset. A new object called india_pop is created to store this data.

India 2011 Population

The 15th Indian Census was conducted in two phases, house listing and population enumeration. House listing phase began on 1 April 2010 and involved collection of information about all buildings. Information for National Population Register was also collected in the first phase, which will be used to issue a 12-digit unique identification number to all registered Indian residents by Unique Identification Authority of India (UIDAI).

The second population enumeration phase was conducted between 9 and 28 February 2011. Census has been conducted in India since 1872 and 2011 marks the first-time biometric information was collected. According to the provisional reports released on 31 March 2011, the Indian population increased to 1.21 billion with a decadal growth of 17.64%.

```
india_pop <- read.csv("C:/Users/jeff/Documents/Crime
Analysis/India_data/India_pop_2011.csv", stringsAsFactors = FALSE)
datatable(india_pop, options = list(scrollX='400px')) # retail price
information about key commodities by city
```

Show 10 entries Search:

	Rank	Date	Centre	ST_NAME	Region	Country	X	Y	Pop_2011	X.1
1	64	2011	Port Plair	ANDAMAN AND NICOBAR ISLANDS	SOUTH	INDIA	92.4416	11.4006	100186	
2	13	2011	VISAKHAPATNAM	Andhra Pradesh	SOUTH	INDIA	83.218482	17.686816	2035922	
3	28	2011	VIJAYWADA	Andhra Pradesh	SOUTH	INDIA	80.648015	16.506174	1034358	
4	66	2011	Itanagar	Arunachal Pradesh	EAST	INDIA	93.3712	27.06	59490	
5	32	2011	GUWAHATI	Assam	NORTH EAST	INDIA	91.736237	26.144518	963429	
6	17	2011	PATNA	Bihar	EAST	INDIA	85.137565	25.594095	1683200	
7	45	2011	BHAGALPUR	Bihar	EAST	INDIA	86.98243	25.347799	398138	
8	52	2011	PURNIA	Bihar	EAST	INDIA	87.475255	25.777139	280547	
9	34	2011	CHANDIGARH	CHANDIGARH	NORTH	INDIA	76.779419	30.733315	960787	
10	30	2011	RAIPUR	Chhattisgarh	WEST	INDIA	81.629641	21.251384	1010087	

Showing 1 to 10 of 69 entries Previous 1 2 3 4 5 6 7 Next

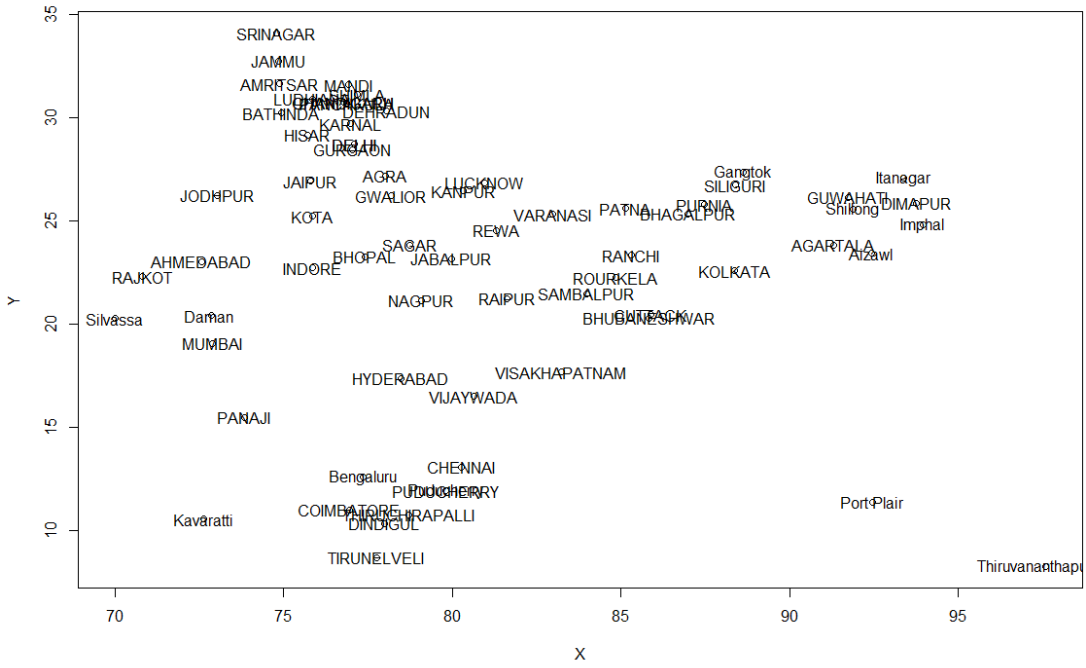
Joining Tables

We use left_join because we want the length of the data frame to remain unchanged, with variables from new data appended in new columns (see `?left_join`). The *join commands (including inner_join and anti_join) assume, by default, that matching variables have the same name. Here we will specify the association between variables in the two data sets:

```
head(ind$ST_NAME) # dataset to add to (results not shown)

## [1] ANDAMAN AND NICOBAR ISLANDS Andhra Pradesh
## [3] Arunachal Pradesh           Assam
## [5] Bihar                       CHANDIGARH
## 35 Levels: ANDAMAN AND NICOBAR ISLANDS ... West Bengal

head(india_pop$ST_NAME) # the variables to join

## [1] "ANDAMAN AND NICOBAR ISLANDS" "Andhra Pradesh"
## [3] "Andhra Pradesh"              "Arunachal Pradesh"
## [5] "Assam"                       "Bihar"

# head(left_join(ind@data, crime_ag)) # test it works
ind@data <- left_join(ind@data, india_pop, by = c('ST_NAME' = 'ST_NAME'))

## Warning: Column `ST_NAME` joining factor and character vector, coercing
## into character vector
```

Basic Scatterplots

```
attach(india_pop)
plot(X,Y)  + text(X,Y, labels=india_pop$Centre)
```




```
## integer(0)
```

India.shp Map Rescaled with lat/long Grids

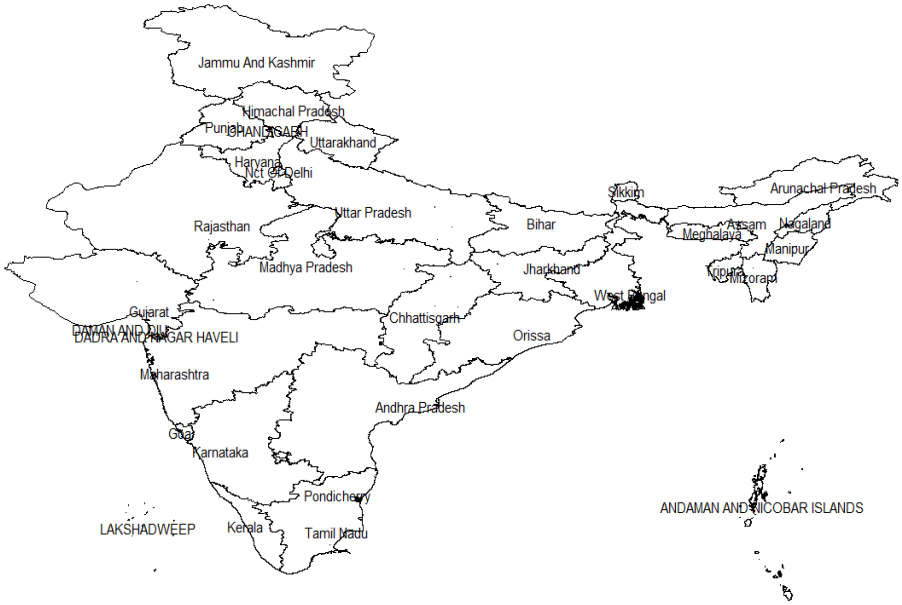
```
require(ggplot2)
map <- ggplot() + geom_polygon(data = ind, aes(x = long, y = lat, group =
group), colour = "black", fill = NA)

# We printed this one earlier and will not show it here

## Regions defined for each Polygons

map + theme_void()
```

Create C=Shape Dataframe

India.shp Map with States

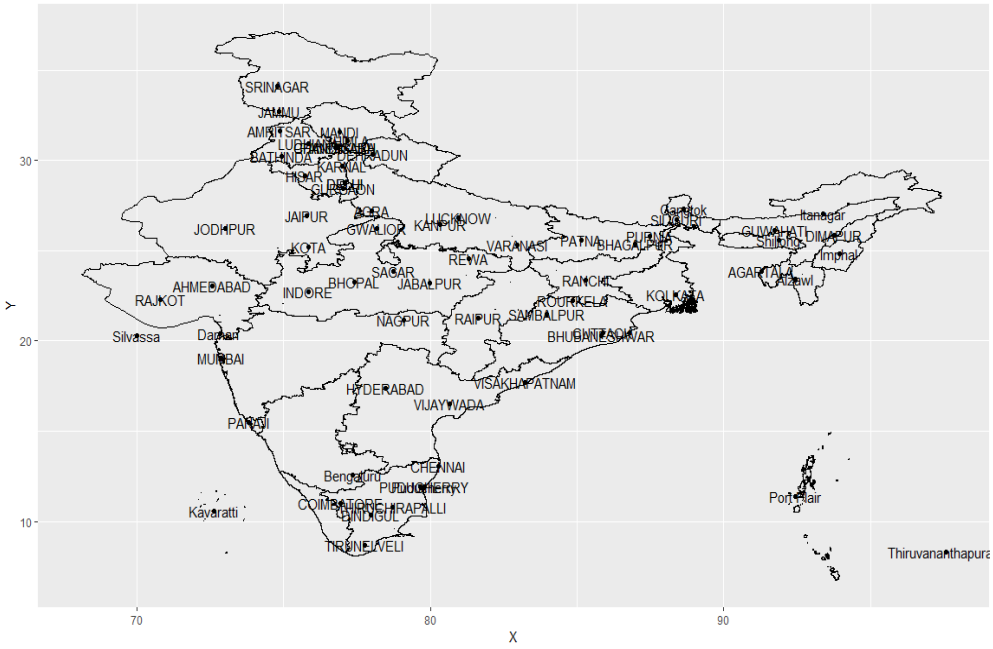
```
map <- ggplot() + geom_polygon(data = ind, aes(x = long, y = lat, group =
group), colour = "black", fill = NA)

## Regions defined for each Polygons
```

```
cnames <- aggregate(cbind(long, lat) ~ id, data=shp_df, FUN=mean)
map + geom_text(data = cnames, aes(x = long, y = lat, label = id), size = 4)
+ theme_void()
```



India.shp Map with Cities

```
ggplot(india_pop, aes(X,Y)) + geom_point() + geom_text(aes(label=Centre)) +
  geom_polygon(data = ind, aes(x = long, y = lat, group = group), colour =
"black", fill = NA)

## Regions defined for each Polygons
```

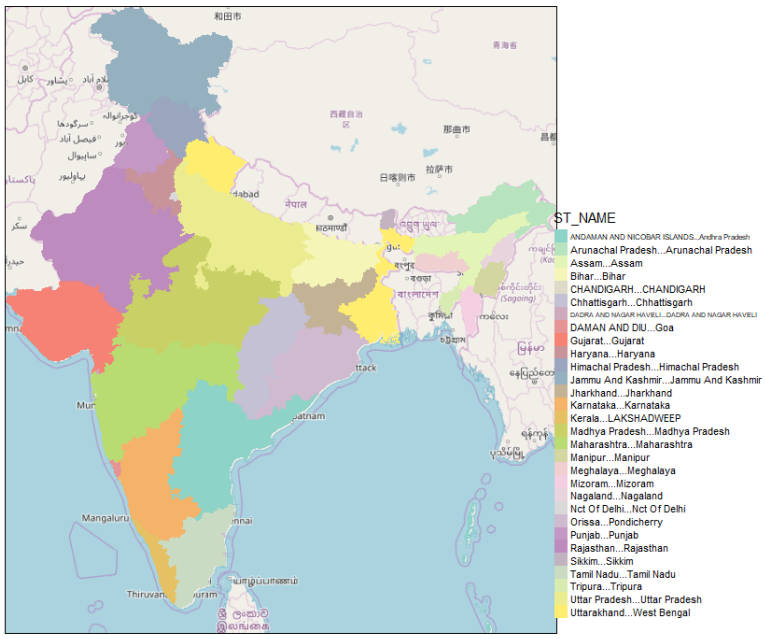


India Map with Sates Setup

```
library(tmaptools)
ind_wgs = spTransform(ind, CRS("+init=epsg:4326"))
osm_tiles = read_osm(bbox(ind_wgs))
ind_wgs$ST_NAME <- ind$ST_NAME
```

Plot India Map with States (colored)

```
tm1<-tm_shape(osm_tiles) +
  tm_raster() +
  tm_shape(ind_wgs) +
  tm_fill("ST_NAME", legend.show=TRUE) +
  tm_layout(legend.position = c(0.99,0.02))
tm1
```

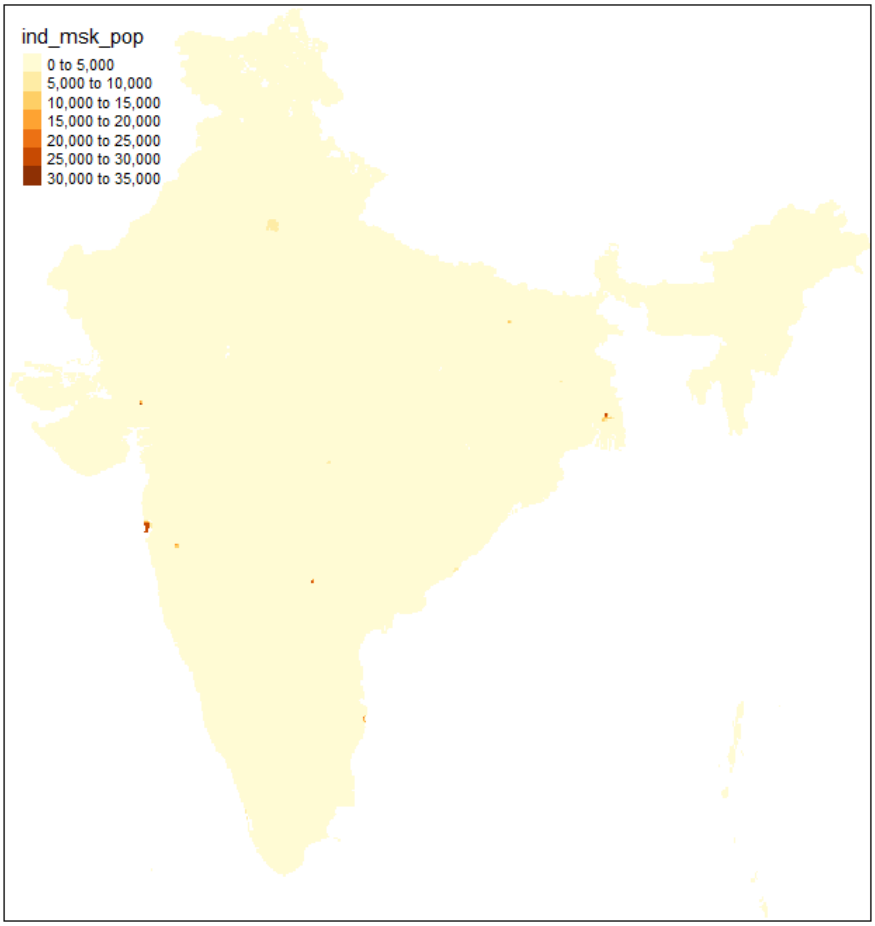


image

India Population Mask (colored)

Mumbai appears more vividly on this population mask as a hot-read spot. Kolkata, formerly Calcutta, appears on the northeast coast and Chennai on the southeast cost as lighter shades of red. Delhi appears in the north as a larger orange area. Between Mumbai and Chennai is the red dot of Hyderabad, India's sixth largest city. North of Mumbai is Ahmedabad, another red dot. According to the scale, the hotter the dot, the more populous the city.

```
tm2<-tm_shape(ind_msk_pop) +
  tm_raster()
tm2
```



Define Popups for Leaflet Map

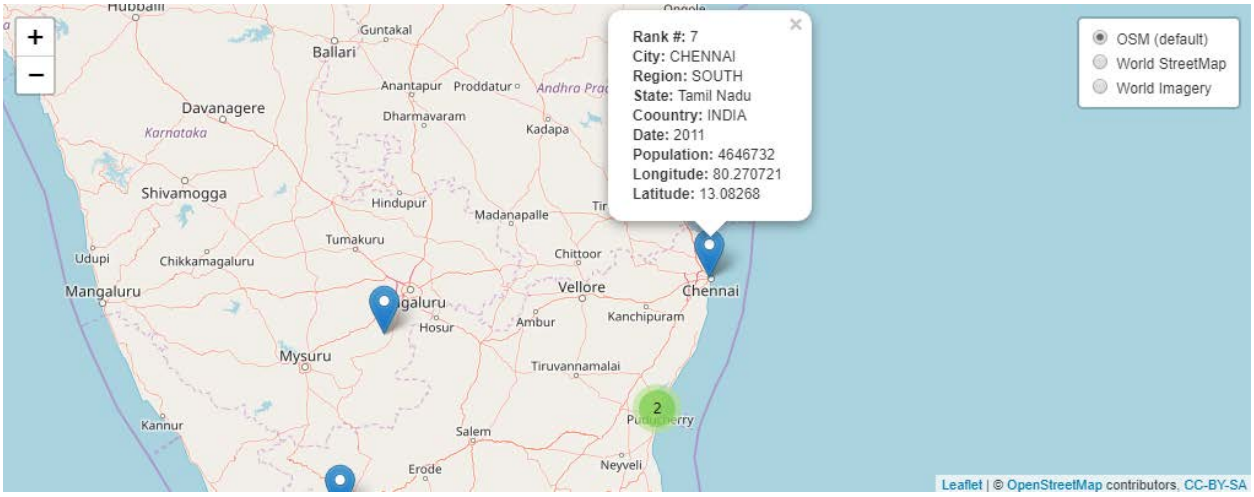
The following code defines the "popups" we will use in a **leaflet** generated map.

```
india_pop$popup <- paste("<b>Rank #: </b>", india_pop$Rank,
                         "<br>", "<b>City: </b>", india_pop$Centre,
                         "<br>", "<b>Region: </b>", india_pop$Region,
                         "<br>", "<b>State: </b>", india_pop$ST_NAME,
                         "<br>", "<b>Coountry: </b>", india_pop$Country,
                         "<br>", "<b>Date: </b>", india_pop$Date,
                         "<br>", "<b>Population: </b>", india_pop$Pop_2011,
                         "<br>", "<b>Longitude: </b>", india_pop$X,
                         "<br>", "<b>Latitude: </b>", india_pop$Y)
```

India Leaflet Population Map

Using `leaflet`, we now generate a map of India with popups defined above. The figure is zoomed-in to southern India and show the popup for Chennai.

```
library(leaflet)

leaflet(india_pop, width = "100%") %>% addTiles() %>%
  addTiles(group = "OSM (default)") %>%
  addProviderTiles(provider = "Esri.WorldStreetMap",group = "World
StreetMap") %>%
  addProviderTiles(provider = "Esri.WorldImagery",group = "World Imagery")
%>%
  # addProviderTiles(provider = "NASAGIBS.ViirsEarthAtNight2012",group =
"Nighttime Imagery") %>%
  addMarkers(lng = ~X, lat = ~Y, popup = india_pop$popup, clusterOptions =
markerClusterOptions()) %>%
  addLayersControl(
    baseGroups = c("OSM (default)","World StreetMap", "World Imagery"),
    options = layersControlOptions(collapsed = FALSE)
  )
```



Summary

This has been a concise overview of spatial mapping in R. We have only touched the tip of the iceberg and I am sure you already see the applications.