ONLINE COURSE

Introduction to GIS

Manipulating and Mapping Geospatial Data in R

Feedback, questions, comments or requests?
Email us at **humansofdata@atlan.com**

atlan

LESSON 1

Use Cases of Geospatial Data

This lesson was written by Sean Angiolillo and was last updated on 29 Jan. 2019.

When you hear "geospatial data", what comes to your mind? For many people, it's ordinary maps, either of physical or human geography. Maps are one important output of geospatial data, but they can be used for so much more. In fact, geospatial data is an overlooked and underappreciated aspect of today's "big data" revolution.

Geospatial Data Is Everywhere

Many of today's "big data" innovations and ideas are rooted in geospatial data sources. Most people always carry at least one device, such as a smartphone, that tracks their location — the result is geospatial data. Entire industries like the Internet of Things (IoT), drones, and autonomous cars rely on this sort of reliable, real-time geospatial data. Similarly, entire concepts like "smart cities" are built around the idea of using geospatial data.

The growth of geospatial data is fairly recent, but its effect can already be seen in many of the products we use on a daily basis. Google search results are more local, Ubers arrive faster, and we can even track the exact location of our Grubhub or Zomato order for free. All of these products and services use geospatial data to tailor the product experience to each user.

Before we dive into working with geospatial data in R, let’s talk about why you should even be interested in learning about geospatial data by highlighting a quick overview of both business and public use cases.

> LEARN MORE:
>
> The greater accessibility of geospatial data has helped spur a wide variety of use cases. One very enthusiastic website has a list of over 1,000 use cases of geospatial data, including fields like retail, health care, transportation, and governance. ESRI also maintains a catalogue of interesting case studies using geospatial data.

How Businesses Use Geospatial Data

If you're familiar with how businesses use data to make better-informed decisions, it's not a far leap to imagine how geospatial data fits into this picture.

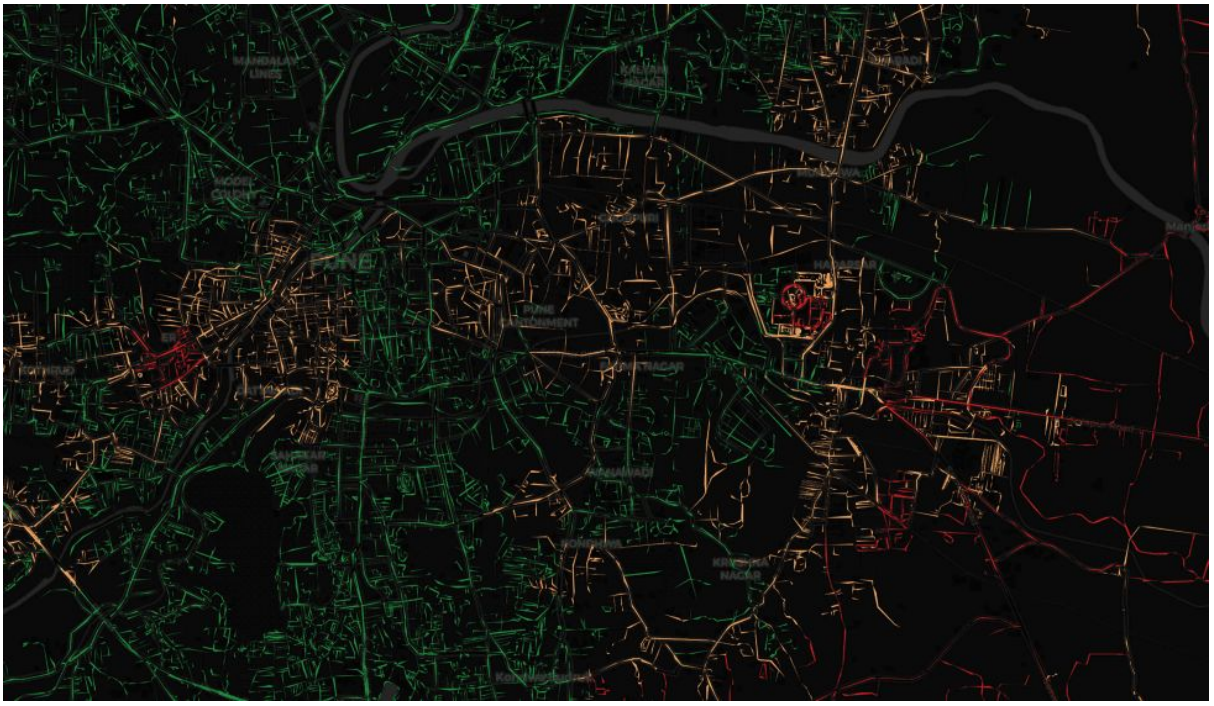
Perhaps the most obvious place to start when discussing how businesses can use geospatial data is improved **transportation and logistics planning**. Companies can use real-time geospatial data to optimize their supply chains and thereby reduce costs. In addition, geospatial data allows businesses to not just react to weather and climate patterns in real time, but also make predictions. For instance, analyzing historical weather trends can help a retailer better anticipate customer demand and adjust supply accordingly.

In another sense, geospatial data can significantly improve **market segmentation** efforts. All businesses care about refining their sales and marketing to more effectively reach customers. For example, many businesses rely on predictive models that try to identify likely customers based on factors such as purchase history or demographic information. The ability to add location to these models can significantly increase returns by increasing customer retention, reducing churn, and finding new customers.

Next, geospatial data can play a key role in a business' **risk analysis** efforts. The idea of modeling risk to mitigate potential exposure is important in many industries, and geospatial data fits well into such models. For example, it can be used to more accurately assess which properties may be at greater risk due to environmental damage and extreme weather events, such as flooding.

In a similar vein to risk analysis is **fraud detection and prevention**. Fraud detection, particularly in industries like credit cards, is one area where machine learning models made an early contribution for their ability to sift through huge mounds of data and

flag irregular or suspicious transactions. Incorporating geospatial data into these kinds of models can help further identify the signal in the noise.



One example of geospatial data is this road data for Pune, a city in India, color-coded to show which roads are well-lit at night. (Source: Atlan.)

Lastly, geospatial data can play a role in almost any kind of **optimization** exercise. Take the example of identifying new locations for a business. Whether determining where to open the next branch of a chain restaurant or Amazon HQ2, it would be foolish to ignore geospatial data. It allows companies to spatially visualize simultaneous layers of analysis, such as target market size, number of competitors, public amenities and infrastructure, and environmental risk factors.

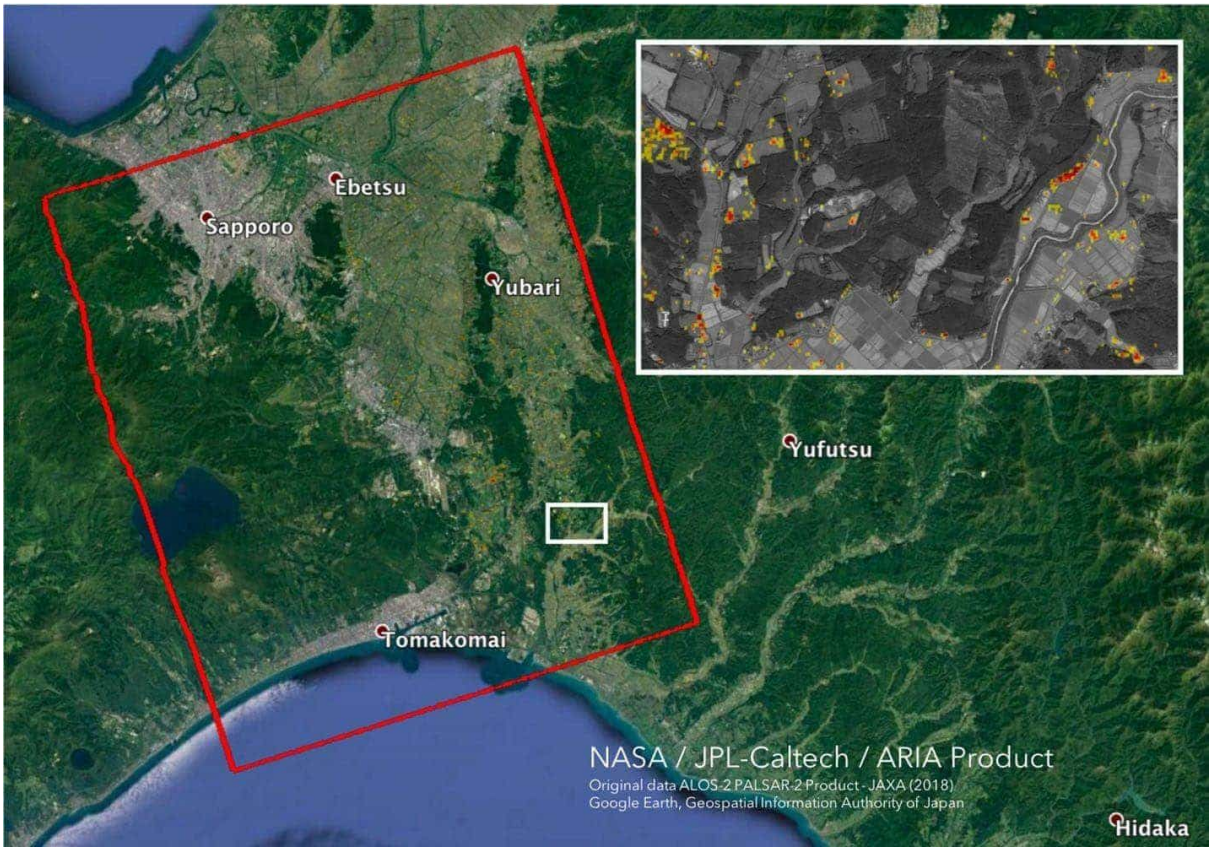
> LEARN MORE:
>
> Many people have explained how businesses are incorporating geospatial data into analytics solutions, but Fern Halper's TDWI bulletin on use cases of geospatial analytics is an especially useful place to begin.

How the Public Sector Uses Geospatial Data

Many of the business use cases have a similar corollary in the public or nonprofit domain.

While a business may use geospatial data for logistics planning, a public agency can use geospatial data for better **urban and rural planning** to improve land use, environmental protection, pollution levels, and liveability. If a business can optimize its fleet of delivery trucks, a public agency can optimize emergency response teams and evacuation routes after a natural disaster. Even basic public functions like trash collection or the rates charged for parking spaces can be optimized based on some level of geospatial data.

Businesses may use geospatial data for market segmentation, but public agencies can use the same principles to better **target constituents**. For example, public agencies can work with geospatial data to better identify at-risk populations for mosquito-borne diseases based on proximity to areas with poor water drainage. Public health missions can then more accurately target their outreach to these populations. Alternatively, much like a business targeting customers, political parties have used geospatial data to target potential voters.



An example of geospatial data is this Damage Proxy Map (DPM) of areas in Hokkaido, Japan, that were likely damaged by the M6.6 earthquake on September 5, 2018. (Source: NASA's Jet Propulsion Laboratory.)

Like a business, a public agency may want to assess risk to **the public and infrastructure**. A new use case, known as predictive policing, is under trial in certain parts of the world. The idea is for police agencies to assess the risk of crime in a given area through geospatial and temporal analysis of past crimes, and thereby

adjust how many police are on call and where they are stationed. While this raises a number of important ethical concerns, it highlights the far-reaching implications of geospatial data.

A business might use geospatial data in its fraud detection efforts, but a public agency might use geospatial data to track **illegal construction**. The Delhi Government in fact had a project to achieve just this by creating what is called the Delhi State Spatial Data Infrastructure (DSSDI).

If businesses can use geospatial data to find strategic locations, public agencies can do the same to determine the location of important **public infrastructure** like new roads, hospitals, schools and voting booths. We even used geospatial data in our effort to locate where to position 10,000 new LPG (liquefied petroleum gas) distribution centers across India.

One last use case in the public sector is the idea of investing in **geospatial data as a public good**. For example, one important form of geospatial data is weather data. More accurate, localized, and accessible weather reports can have a large impact on the agricultural sector. By publishing digitized and spatially-mapped land records, governments can mitigate land acquisition disputes. Geospatial data on environmental metrics like air quality or carbon emissions is another example of a public good. In the case of India, major government initiatives like Digital India and Smart Cities rely on these geospatial technologies.

> LEARN MORE:
>
> One useful resource on how GIS technology can improve city services is this McKinsey brief. Or for analysis focused on the Indian context, this FICCI report highlights 40 cases on geospatial technologies in different sectors.

Final Thoughts

If this brief overview of geospatial data use cases has piqued your interest, be sure to keep reading. Starting in the next lesson, you'll quickly get your hands dirty working directly with geospatial data in R.

View this lesson online

This lesson was written by Sean Angiolillo and was last updated on 29 Jan. 2019.

The previous lesson looked at the many growing use cases for geospatial data. Now we'll get started manipulating geospatial data in R using the `sf` package. You'll learn how to import spatial data, combine attribute data into existing geospatial objects, calculate area, and simplify spatial dataframes before plotting them (which is the subject of the next lesson).

Using R as a GIS

Until recently, serious work with geospatial data required an often-proprietary desktop GIS (Geographic Information System), such as ArcGIS. Now, however, GIS capabilities in R have greatly advanced.

In many ways, the benefits of using R over a desktop GIS are similar to the benefits of using R over Excel for data analysis.

- R is **free and open source**, which makes it easier and cheaper to get started, compared to the expensive licenses needed for many desktop GIS. This has helped spread geospatial data analysis beyond the domain of only GIS specialists, thereby opening opportunity to a wider ecosystem of contributors.
- As a full-fledged programming language, the command line interface of R has greater **flexibility** than the point-and-click interface of a desktop GIS. This means there's no restriction on what's ultimately achievable.
- R is **reproducible** in a way that point-and-click interfaces are not. This is, of course, key to the scientific process.

- The greater **shareability** of R scripts and packages encourages a faster development cycle and a more collaborative workflow.

Many of these advantages apply to any open-source programming language, such as Python, which shares an API with many desktop GIS. However, R, designed as an environment for statistical computing, is particularly well-suited for spatial statistics and offers unmatched access to a huge ecosystem of statistical libraries. R's data visualization libraries in particular are a key advantage when it comes to mapping geospatial data.

> LEARN MORE:
>
> See "Why Geocomputation with R?" for a greater discussion of the merits of R for GIS.

Getting Started with R as a GIS

Before working with a new domain area in R, it's useful to check its associated CRAN Task View for an overview of relevant packages. The CRAN Task View for Analysis of Spatial Data lists dozens of packages relating to geospatial data. Until recently, the first package mentioned was `sp`, but now it first shows the `sf` package and notes that maintenance of `sp` will continue.

What package should you use for geospatial data? The answer could differ based on your objectives, but the `sf` package is likely the best place to get started. The **`sf`** (Simple Features) package provides a class system for geographic vector data. It is the successor to the `sp` package and is quickly being adopted by many other packages for geospatial data.

Let's start exploring some of the package's features with simple state-level Indian population and economic data.

> LEARN MORE:
>
> Never used the `sf` package? Get started with the recently completed open source book under development, *Geocomputation with R* by Robin Lovelace,

> Jakub Nowosad and Jannes Muenchow. In particular, Chapter 2 gives a great introduction to what simple features are and the structure of `sf` objects.
>
> Spatial data enthusiasts are also excited about the announcement that `sf` package authors Edzer Pebesma and Roger Bivand are currently working on an open source book of their own, *Spatial Data Science*. Drafts of the first eight chapters of what is sure to be a key resource for the field are now available.

Creating sf Objects

For tidyverse users, one of the most exciting aspects of the `sf` package is the ability to work with geospatial data in a tidy workflow. Unlike its predecessor package `sp`, with the `sf` package, geospatial and attribute data can be stored together in a **spatial dataframe**, where the object's geometry occupies a special list-column. In addition to being faster, this lets you manipulate an `sf` object via `magrittr` pipes like an ordinary dataframe, or at least one with a few special characteristics.

It's certainly possible to create your own `sf` objects with functions from the package like `st_point()`, `st_linestring()`, and `st_polygon()`. But in most cases we only have to read in existing spatial data. That is generally done with the `st_read()` function.

The shapefiles used in this demonstration can be found in this blog's associated GitHub repository.

```
library(sf)
my_sf <- st_read("india_states_2014/india_states.shp")
```

```
## Reading layer `india_states' from data source `/Users/seanangiolillo/
Library/Mobile Documents/com~apple~CloudDocs/viz_india/india_states_201
4/india_states.shp' using driver `ESRI Shapefile'
## Simple feature collection with 36 features and 8 fields
## geometry type:    MULTIPOLYGON
## dimension:        XY
## bbox:             xmin: 68.11009 ymin: 6.755698 xmax: 97.4091 ymax: 3
7.0503
## epsg (SRID):      4326
## proj4string:      +proj=longlat +datum=WGS84 +no_defs
```

> NOTE:
>
> All `sf` functions begin with `st_*` to help users identify them.

In general, you can find administrative boundary data from GADM. It maintains open-source, current administrative boundary data for most countries. It's possible to download spatial data directly from the GADM website, but using the `GADMTools` package helps ensure your workflow is reproducible. Specifying `level = 1` returns state-level boundaries.

> LEARN MORE:
>
> Robin Wilson's website provides a great list of free GIS data sources, covering both physical and human geography.

The code below will download shapefiles for Indian states. However, since Kashmir isn't included in India's borders, we'll use the shapefiles in the repository.

```
library(GADMTools)
india_wrapper <- gadm.loadCountries("IND", level = 1, basefile = "./")
# check your directory for a file "IND_adm1.rds" after running this comm
and
```

Inspecting Objects

Before working with this `sf` object, let's briefly compare the `sf` and `sp` packages.

First, we'll convert the `sf` object to a `SpatialPolygonsDataFrame`, an S4 class defined by the `sp` package. We can do this with `sf::as()`.

```
my_spdf <- as(my_sf, "Spatial")
class(my_spdf)
```

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

Now we can briefly inspect the structure of a `SpatialPolygonsDataFrame`.

```
str(my_spdf, max.level = 2)
```

```
## Formal class 'SpatialPolygonsDataFrame' [package "sp"] with 5 slots
##   ..@ data       :'data.frame': 36 obs. of 8 variables:
##   ..@ polygons   : List of 36
##   ..@ plotOrder  : int [1:36] 8 30 26 35 2 29 25 1 31 18 ...
##   ..@ bbox       : num [1:2, 1:2] 68.11 6.76 97.41 37.05
##   .. ..- attr(*, "dimnames")=List of 2
##   ..@ proj4string : Formal class 'CRS' [package "sp"] with 1 slot
```

We notice it has 5 "slots" (each prefaced by the `@` symbol). The first slot should look familiar. The data slot holds a dataframe with 36 observations of 8 variables. We can extract any of these slots using the `@` symbol like we'd normally do with the `$` symbol.

```
library(tidyverse)
glimpse(my_spdf@data)
```

```
## Observations: 36
## Variables: 8
## $ id         <dbl> 28, 1, 2, 3, 5, 6, 4, 8, 12, 7, 11, 13, 17, 14, 2
0, ...
## $ name       <fct> Andhra Pradesh, Jammu & Kashmir, Himachal Pradesh,
P...
## $ abbr       <fct> AP, JK, HP, PB, UT, HR, CH, RJ, AR, DL, SK, NL, M
L, ...
## $ geo_level  <int> 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3...
## $ minx       <fct> 76.7600000000000051, 72.5742599999999953, 75.60502
99...
## $ miny       <fct> 12.6230899999999995, 32.2647800000000018, 30.38166
00...
## $ maxx       <fct> 84.7149699999999939, 80.3008899999999954, 79.05115
99...
## $ maxy       <fct> 19.1614800000000010, 37.0503000000000000, 33.22191
00...
```

The following slots hold `polygons`, `plotOrder`, `bbox` and `proj4string`. We'll return to these in the context of `sf` objects. For now, just note that the data and aspects of the object's geometry are held in separate slots. This is not compatible with the tidyverse-style workflow to which many of us have grown accustomed.

While we could continue working with this format, let's convert it back to an `sf` object with the `st_as_sf()` function and inspect the difference.

```
ind_sf <- st_as_sf(my_spdf)
class(ind_sf)
```

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

The former `SpatialPolygonsDataFrame`, a class defined by the `sp` package, now has two simultaneous classes: `sf` and `data.frame`. Printing the first few observations tells us a lot about the object.

```
head(ind_sf, 3)
```

```
## Simple feature collection with 3 features and 8 fields
## geometry type:    MULTIPOLYGON
## dimension:        XY
## bbox:             xmin: 72.57426 ymin: 12.62309 xmax: 84.71497 ymax:
37.0503
## epsg (SRID):      4326
## proj4string:      +proj=longlat +datum=WGS84 +no_defs
##   id               name abbr geo_level                minx
## 1 28     Andhra Pradesh    AP         3 76.7600000000000051
## 2  1    Jammu & Kashmir    JK         3 72.5742599999999953
## 3  2  Himachal Pradesh    HP         3 75.6050299999999993
##                   miny                maxx                maxy
## 1 12.6230899999999995 84.7149699999999939 19.1614800000000010
## 2 32.2647800000000018 80.3008899999999954 37.0503000000000000
## 3 30.3816600000000001 79.0511599999999959 33.2219100000000012
##                           geometry
## 1 MULTIPOLYGON (((80.0553 15....
## 2 MULTIPOLYGON (((77.57808 35...
## 3 MULTIPOLYGON (((75.80153 32...
```

- It has 36 features (depending on how many we print) and 8 fields (our attributes).
- The `geometry type` is a multipolygon because the geometries represent the shapes of various areas. Other common geometry types include points, lines, and their “multi-” counterparts.
- `bbox` gives the object’s bounding box dimensions.
- `epsg` and `proj4string` describe the coordinate reference system (CRS). Note that this is a geographic CRS (measured in longitude and latitude) as opposed to a projected CRS.

> LEARN MORE:
>
> For more information on coordinate reference systems, see Section 2.4 of *Geocomputation with R*.

We could directly access information about the object's spatial features with functions like `st_geometry_type()`, `st_dimension()`, `st_bbox()` and `st_crs()`. Moreover, familiar functions like `glimpse()` or `View()` that we'd use to explore a dataframe also work on `sf` objects.

If you further inspect this object, you can see that it has a few attribute columns giving an abbreviation and bounding box for each state. Most importantly, the last column holds each state's geometry in a list-column.

```
glimpse(ind_sf)
```

```
## Observations: 36
## Variables: 9
## $ id        <dbl> 28, 1, 2, 3, 5, 6, 4, 8, 12, 7, 11, 13, 17, 14, 20,
...
## $ name      <fct> Andhra Pradesh, Jammu & Kashmir, Himachal Pradesh,
P...
## $ abbr      <fct> AP, JK, HP, PB, UT, HR, CH, RJ, AR, DL, SK, NL, ML,
...
## $ geo_level <int> 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3...
## $ minx      <fct> 76.7600000000000051, 72.5742599999999953, 75.605029
9...
## $ miny      <fct> 12.6230899999999995, 32.2647800000000018, 30.381660
0...
## $ maxx      <fct> 84.7149699999999939, 80.3008899999999954, 79.051159
9...
## $ maxy      <fct> 19.1614800000000010, 37.0503000000000000, 33.221910
0...
## $ geometry  <MULTIPOLYGON [°]> MULTIPOLYGON (((80.0553 15...., MULTIP
O...
```

Manipulating sf Objects

Because spatial dataframes in the `sf` package are dataframes, we can manipulate them using our normal data manipulation tools, such as `dplyr`. Here we'll just select and rename the columns we want.

```
uts <- c("Delhi", "Andaman & Nicobar Islands", "Puducherry",
         "Lakshadweep", "Dadra & Nagar Haveli", "Daman & Diu",
         "Chandigarh")

ind_sf <- ind_sf %>%
     select(name, abbr) %>%
     mutate(
          type = ifelse(name %in% uts, "Union Territory", "State")
     ) %>%
     rename(abb = abbr, state_ut = name)
```

> NOTE:
>
> This doesn't explicitly select the geometry column, but the geometry in `sf` objects is **sticky**. It remains in the object unless explicitly dropped with `ind_sf%>% st_set_geometry(NULL)`.

For those already familiar with the tidyverse, using normal `dplyr` verbs to manipulate `sf` objects is one of the great benefits of using the `sf` package. If we were working with the slots of the earlier `SpatialPolygonsDataFrame`, this wouldn't be possible. Moreover, note that these manipulations haven't affected the class of our object in any way.

```
class(ind_sf)
```

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

Preparing Attribute Data

Now that we have a spatial dataframe, we need to prepare the associated attribute data for each state or union territory.

Since we're focusing on the process rather than the data itself, we'll use population, economic, and region data from Wikipedia. If you are unfamiliar with data import using the **`googlesheets`** package, web scraping with **`rvest`**, and wrangling with

dplyr, please refer to the `prepare_data.R` script in this GitHub repository to see how the `attributes.rds` data set was assembled.

> LEARN MORE:
>
> Unfamiliar with these packages? Check out the `googlesheets` package vignette and the Data Transformation chapter of Hadley Wickham's *R for Data Science*.

Once we've prepared an attributes dataframe, we can join it to the spatial dataframe as with any two dataframes, as well as mutate two new variables.

```
# see prepare_data.R script in github for details of creating attributes
_df
attributes_df <- readRDS("attributes.rds")

ind_sf <- ind_sf %>%
     left_join(attributes_df, by = "state_ut") %>%
     mutate(
          per_capita_gdp_inr = nominal_gdp_inr / pop_2011,
          per_capita_gdp_usd = nominal_gdp_usd / pop_2011
     )
```

> LEARN MORE:
>
> See the Relational Data chapter of *R for Data Science* if `left_join()` is unfamiliar.

If we inspect this object once more, we can see that it has all of the expected attribute columns, and the last column holds each state's geometry in a list. The same spatial attributes regarding the object's bounding box and CRS remain as well.

```
head(ind_sf, 3)
```

```
## Simple feature collection with 3 features and 17 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 72.57426 ymin: 12.62309 xmax: 84.71497 ymax: 3
7.0503
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
##           state_ut abb  type  data_year comparable_economy nominal_gd
p_usd
## 1   Andhra Pradesh  AP State    2018–19            Hungary         1.3
6e+11
## 2  Jammu & Kashmir  JK State    2018–19            Iceland         2.5
0e+10
## 3 Himachal Pradesh  HP State    2018–19            Iceland         2.5
0e+10
##   nominal_gdp_inr pop_2011 decadal_growth rural_pop urban_pop area_km
2
## 1        8.70e+12 49386799          0.111  34776389  14610410    16296
8
## 2        1.57e+12 12548926          0.237   9134820   3414106    22223
6
## 3        1.52e+12  6864602          0.128   6167805    688704     5567
3
##   density_km2 sex_ratio   region per_capita_gdp_inr per_capita_gdp_u
sd
## 1         303       993 Southern           176160.4            2753.7
72
## 2          57       883 Northern           125110.3            1992.2
02
## 3         123       974 Northern           221425.8            3641.8
72
##                           geometry
## 1 MULTIPOLYGON (((80.0553 15....
## 2 MULTIPOLYGON (((77.57808 35...
## 3 MULTIPOLYGON (((75.80153 32...
```

Calculating Area

Our attribute data already has a column for area, pulled from Wikipedia. However, if this wasn't the case or we didn't trust the data, we could calculate the area of each observation in our spatial dataframe using the `st_area()` function.

It's simple enough to do this, but we need to be careful with the units. In this case, we need to convert from square meters to square kilometers.

```
library(units)
# mutate area
ind_sf <- ind_sf %>%
     mutate(my_area = st_area(.))

# convert units
units(ind_sf$my_area) <- with(ud_units, km^2)

# mutate gdp density
ind_sf <- ind_sf %>%
     mutate(gdp_density_usd_km2 = nominal_gdp_usd / my_area)
```

> LEARN MORE:
>
> See this guide on units of measurement for R vectors for more information on unit conversion in R.

In the output below, note the difference between the simple numeric class of `area_km2` and the class of "units" for the area calculation resulting from `st_area()`.

```
class(ind_sf$area_km2)
```

```
## [1] "numeric"
```

```
class(ind_sf$my_area)
```

```
## [1] "units"
```

Moreover, we can see that the two figures are close but not exactly the same.

```
##          state_ut area_km2       my_area
## 1       Rajasthan   342239 342357.2 km^2
## 2  Madhya Pradesh   308245 308203.8 km^2
## 3     Maharashtra   307713 307337.5 km^2
## 4   Uttar Pradesh   240928 241146.5 km^2
## 5 Jammu & Kashmir   222236 219911.8 km^2
## 6       Karnataka   191791 192004.8 km^2
```

Simplifying Geometry

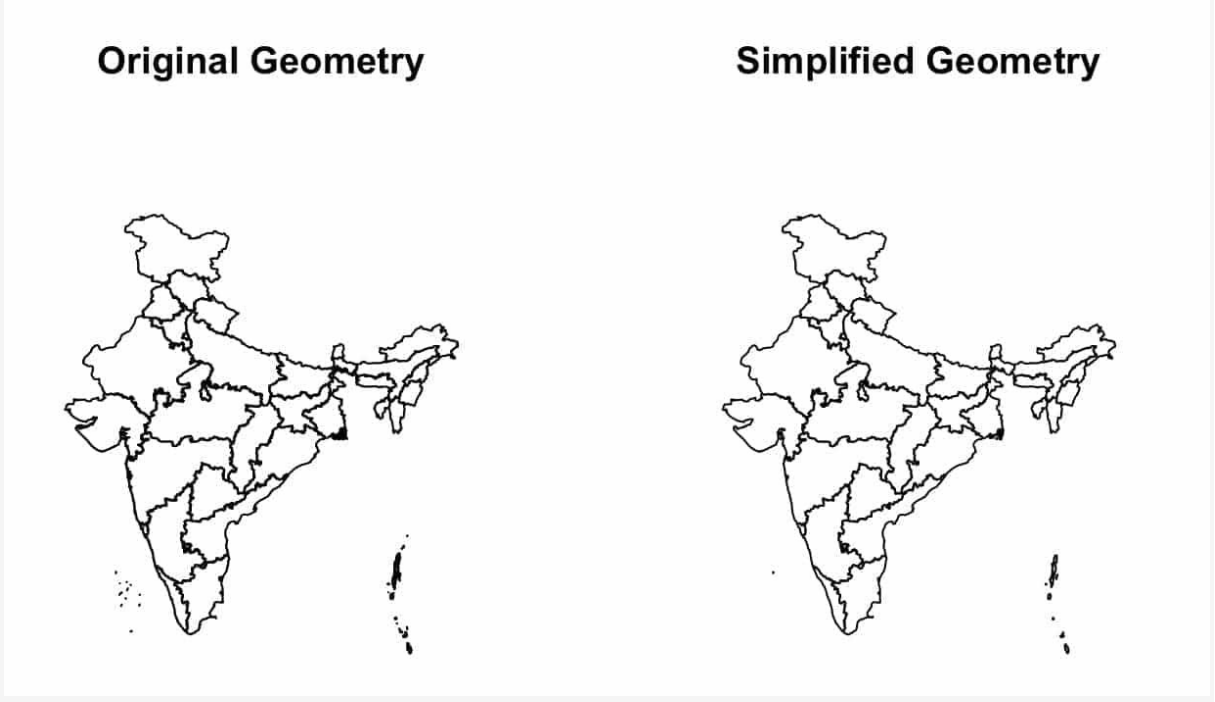
Before plotting `sf` objects (the subject of the next lesson), we should simplify the polygons in the spatial dataframe.

For simple maps, there's no need to have the fine level of detail that comes with the GADM data or many other sources of geospatial data. Simplification can vastly reduce memory requirements while sacrificing very little in terms of visual output. Fortunately, there's an easy process to reduce the number of vertices in a polygon while retaining the same visible shape.

One option is `sf::st_simplify()`, but here we'll use the **`ms_simplify()`** function from the `rmapshaper` package. Below we keep only 1% of the object's vertices while maintaining the same number of shapes.

> NOTE:
>
> Another useful function is `sf::st_geometry()`. When passing it an `sf` object, it will return just the geometry. This allows us to create a quick plot of only the geometry to check if everything looks right.
>
> We also stripped the units class for the area we calculated because it created a problem for `ms_simplify()`. We can always add it after simplification.

```
# strip units class
ind_sf <- ind_sf %>%
     mutate(
          my_area = as.vector(my_area),
          gdp_density_usd_km2 = as.vector(gdp_density_usd_km2)
     )

original_geometry <- st_geometry(ind_sf)

library(rmapshaper)
simp_sf <- ms_simplify(ind_sf, keep = 0.01, keep_shapes = TRUE)
simple_geometry <- st_geometry(simp_sf)

par(mfrow = c(1,2))
plot(original_geometry, main = "Original Geometry")
plot(simple_geometry, main = "Simplified Geometry")
```



The original map is above on the left, and the simplified version is on the right. The simplified version looks no different despite having only 1% of the vertices. In fact, it looks even better because the border lines are cleaner.

Moreover, simplification reduced the geometry size from 9.56 MB to just 150 KB.

```
library(pryr)
object_size(original_geometry)
```

```
## 9.56 MB
```

```
object_size(simple_geometry)
```

```
## 150 kB
```

Finally, let's save the simplified spatial dataframe for the next lesson.

```
saveRDS(simp_sf, "simp_sf.rds")
```

> LEARN MORE:
>
> For more information on simplification, see Section 5.2.1 of *Geocomputation with R*.

Final Thoughts

After briefly introducing the context of using R as a GIS, this lesson showed how the `sf` package creates a class structure for storing geospatial and attribute data together in an object that fits into a tidyverse workflow. We can clearly see the benefits of this structure when it comes to manipulating a spatial dataframe with our familiar `dplyr` verbs.

Now that you know how to manipulate geospatial data, the natural next step is visualization or mapping. Here again, we'll see the benefit of a tidy workflow, now that `ggplot2`'s `geom_sf()` is available to us. Though, as you'll see in the next lesson, `ggplot2` is just one of many excellent package options when it comes to visualizing geospatial data in R.

View this lesson online

LESSON 3

Creating Static Maps in R

This lesson was written by Sean Angiolillo and was last updated on 29 Jan. 2019.

In the previous lesson, we briefly explored the structure of spatial dataframes as defined by the `sf` package. The next step is visualization, or more specifically in the case of geospatial data, mapping. As with any kind of data, visualization is an important step before diving into any kind of statistical analysis.

This lesson introduces how to use some of the most well-known R packages to create static maps, such as `tmap` and `ggplot2`. We'll also explore a few other packages like `cartogram`, `geogrid` and `geofacet` for some more unique spatial visualizations. (We'll tackle creating animated and interactive maps in the next lesson.)

Resources on Visualizing Geospatial Data

Before diving into different R packages for mapping, let's review a few excellent resources that will help you get started.

Below are two excellent open source resources on the principles of data visualization. Both include chapters on geospatial data visualization.

- *Data Visualization: A practical introduction* by Kieran Healy. (This even includes a dedicated chapter on maps.)
- *Fundamentals of Data Visualization* by Claus O. Wilke

Here are two resources focused more narrowly on the mechanics of mapping specifically in R, rather than larger principles of good design:

- "Making Maps with R" chapter of the previously-mentioned *Geocomputation with R*
- Bhaskar V. Karambelkar's tutorial at useR 2017 on "Geospatial Data Visualization in R"

Choosing the Right Visualization

Recent advances in software have made many different types of geospatial data visualizations — such as choropleths, dot density maps and cartograms — easily available. However, the correct visualization often begins with the type of data you have.

Before choosing a visualization, we should be sure about the nature of our data. Is the data numeric? And if so, is it a raw count, such as population, or is it standardized, such as population density? If the data isn't numeric, is it nominal (or categorical), such as linguistic or religion data, or ordinal, such as satisfaction rankings?

One point Healy makes clear in his book is that it's important to consider whether or not a truly geospatial visualization is the best choice for your data. In our case, and in many cases concerning choropleths, the data is only partly geospatial — it really represents counts of some value in an arbitrary unit.

The spatial object we created in the previous lesson has attributes like population, GDP, and sex ratio. It is certainly possible to visualize this data through barplots, ignoring the data's geospatial qualities. Alternatively, instead of ignoring the geospatial elements, we could show some of this information through a proxy — for example, mapping different colors to a variable like region.

In fact, focusing on geospatial elements can sometimes misrepresent the data because of vastly unequal areas between different regions and the populations they hold — as is the case for Indian states. If we were working with district-level (as opposed to state-level) data, then a choropleth might be necessary because we can't show hundreds of bars on a single barplot.

Nevertheless, because of its simplicity, we'll use state-level data to test out different approaches for geospatial visualization. With this goal in mind, hopefully none of the

visualizations below are "bad", but whether or not they are the "best" visualization for this data would depend on the specific objectives at hand.

Static Maps

Although a number of R packages have made it easy to make attractive interactive and animated maps, they haven't removed the need for effective static maps. This section looks at how to create static maps in base R, `tmap`, and `ggplot2`.

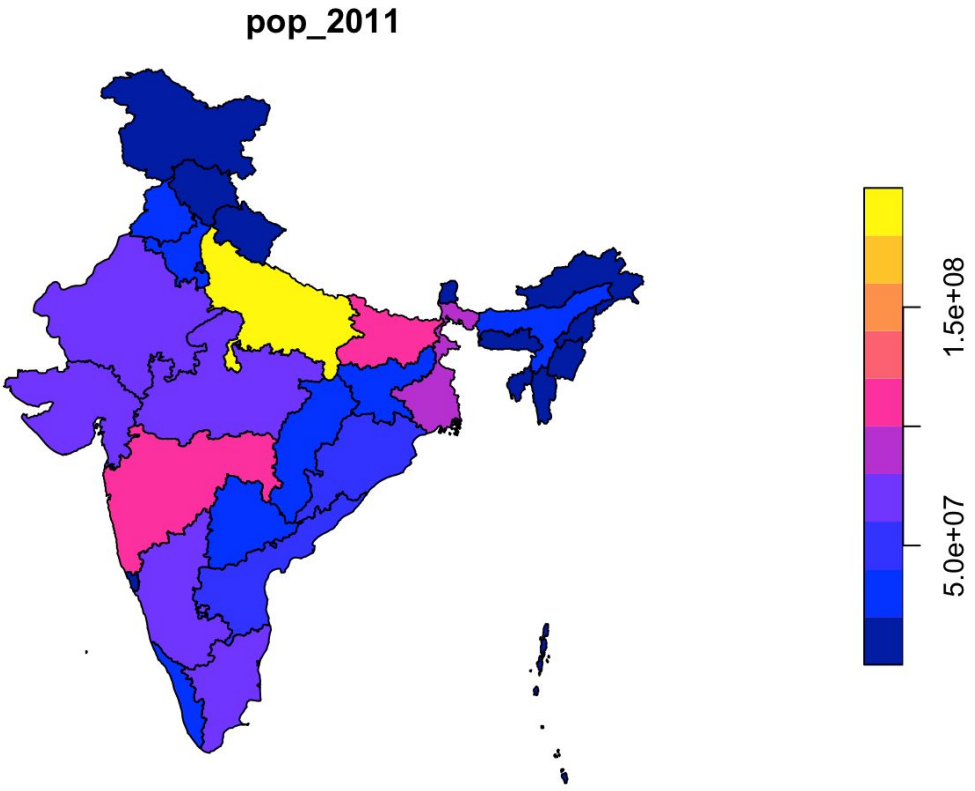
> NOTE:
>
> While `tmap` and `ggplot2` are two of the most popular packages for creating maps, they aren't the only options. The cartography package is another interesting tool, particularly for certain kinds of maps, such as choropleths contained in proportional symbols. See the package vignette and cheat sheet to get started.

Base Plotting

As demonstrated in the previous lesson's plots of geometry, the `sf` package provides a **`plot()`** method for visualizing geographic data. Plotting the object itself will produce a grid of faceted plots, one for each attribute. Choosing a variable produces a single map.

This plot demonstrates how quick and easy plotting base maps can be, but there are reasons why this default choropleth may not be an effective visualization. Hopefully, by the end of this lesson, it will be clear why and what other types of visualization may be more effective in this case.

```
library(tidyverse)
library(sf)
simp_sf <- readRDS("simp_sf.rds")
plot(simp_sf['pop_2011'])
```



Thematic Maps (tmap)

The **`tmap`** package from Martijn Tennekes has been a standard-bearer for mapping in R for some time. As the name "thematic" suggests, it's especially well-suited for choropleths, but it can produce a wide range of geospatial visualizations.

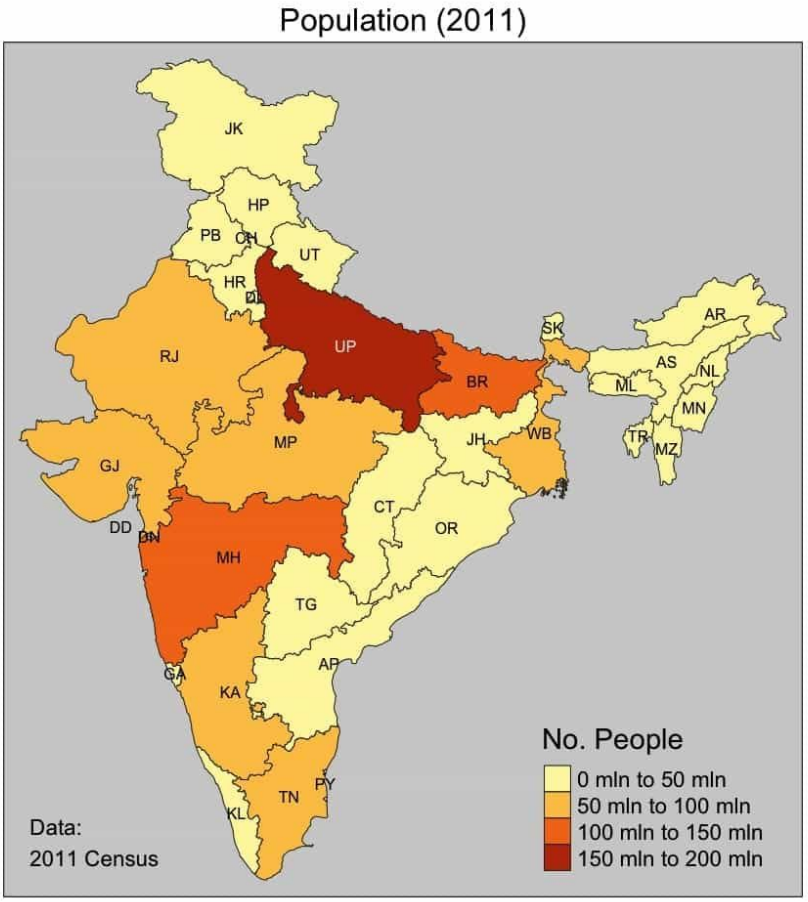
It brings a `ggplot2`-style syntax tailored to geospatial data. Like `ggplot2`, it emphasizes sequentially adding layers to a plot. You can pass a spatial dataframe to the `tm_shape()` function much like you'd pass a dataframe to the `ggplot()` function.

Moreover, because spatial dataframes in the `sf` package are also dataframes, you can filter out any particular features (like "Andaman & Nicobar Islands" below) and directly proceed with piping the object into a `tm_shape()` chain.

> LEARN MORE:
>
> Guides to tweaking other aspects of the map can be found in the documentation, which includes a number of tutorials and vignettes.

After filtering out union territories, the choropleth below maps India's GDP density, a measure of economic activity by area. Measured here in units of nominal GDP per square kilometer, GDP density has no clear midpoint, and so it requires a sequential color scale as opposed to a diverging or categorical color scale.

```
library(tmap)
simp_sf %>%
     filter(!state_ut %in% c("Andaman & Nicobar Islands", "Lakshadweep"
)) %>%
     tm_shape() +
     tm_fill(col = "pop_2011", title = "No. People") +
     tm_borders(lwd = 0.5) +
     tm_text("abb", size = 0.5) +
     tm_style("gray") +
     tm_layout(
          main.title = "Population (2011)",
          main.title.position = c("center"),
          main.title.size = 1,
          legend.position = c("right", "bottom")
     ) +
     tm_credits("Data:\n2011 Census", position = c("left", "bottom"))
```



> LEARN MORE:
>
> There is a great deal of theory and advice about using color in data visualization, including maps. Wilke's book in particular has excellent chapters on color scales and color pitfalls.

Arranging tmap Objects

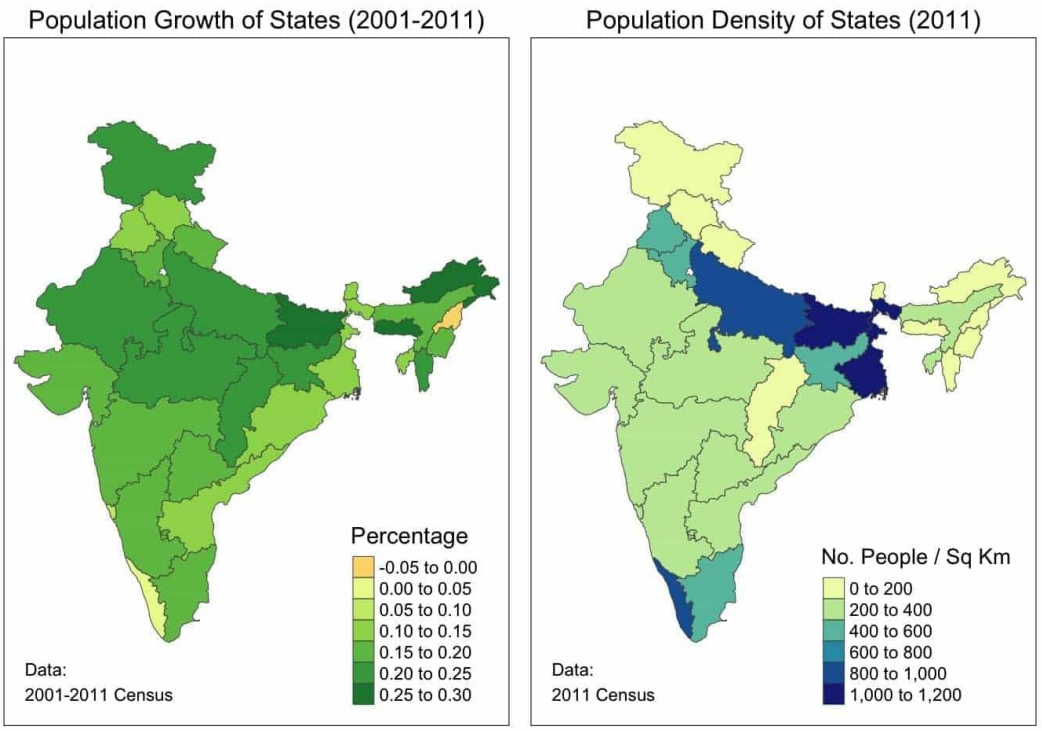
`tmap` also has a helpful function, **`tmap_arrange()`**, for lining up multiple `tmap` objects next to each other.

For example, if we filter out the small union territories to get a fairer distribution, we can separately create `tmap` objects of population growth and density. Then we can arrange them next to each other for comparison.

Like GDP density, values like population growth and population density are standardized data, and so they are well-suited for a choropleth.

```
states_sf <- simp_sf %>%
     filter(!type == "Union Territory")

growth <- tm_shape(states_sf) +
     tm_fill(col = "decadal_growth", title = "Percentage") +
     tm_borders(lwd = 0.5) +
     tm_layout(
          main.title = "Population Growth of States (2001-2011)",
          main.title.position = c("center"),
          main.title.size = 1,
          legend.position = c("right", "bottom")
     ) +
     tm_credits("Data:\n2001-2011 Census", position = c("left", "botto
m"))

density <- tm_shape(states_sf) +
     tm_fill(col = "density_km2", title = "No. People / Sq Km",
             palette = "YlGnBu") +
     tm_borders(lwd = 0.5) +
     tm_layout(
          main.title = "Population Density of States (2011)",
          main.title.position = c("center"),
          main.title.size = 1,
          legend.position = c("right", "bottom")
```

```
    ) +
    tm_credits("Data:\n2011 Census", position = c("left", "bottom"))

tmap_arrange(growth, density)
```



> LEARN MORE:
>
> Axis Maps' cartography guide directly addresses the question of standardizing data with respect to geospatial data.

Inset Maps

`tmap` is also particularly useful for creating **inset maps**, those that include a small window providing the wider geographic context for the main map.

The first step is creating a base or primary map. We have done this for sex ratio in Northeast India.

> NOTE:
>
> One trick from `tmap_tricks()` is reversing the color scale by placing a `-` in front of the palette name. This makes sense because our concern should increase as sex ratio decreases.

```
ne_sex <- simp_sf %>%
     filter(region == "Northeastern") %>%
     tm_shape() +
     tm_fill(col = "sex_ratio", title = "Sex Ratio", palette = "-Reds")
+
     tm_borders(lwd = 0.5) +
     tm_text('state_ut', size = 0.75) +
     tm_layout(
          main.title = "Sex Ratio in India's Northeast",
          main.title.position = c("center"),
          main.title.size = 1
     ) +
     tm_credits("Data Source: Wikipedia", position = c("left", "top"))
```

Next, we created the smaller inset map, which will provide the wider geographic context. For the small map, we wanted to highlight the Northeast region on the larger map of India.

In order to do this, we first grouped the features by region. Using the same `dplyr` syntax, we can reduce the 36 features to 8 regions.

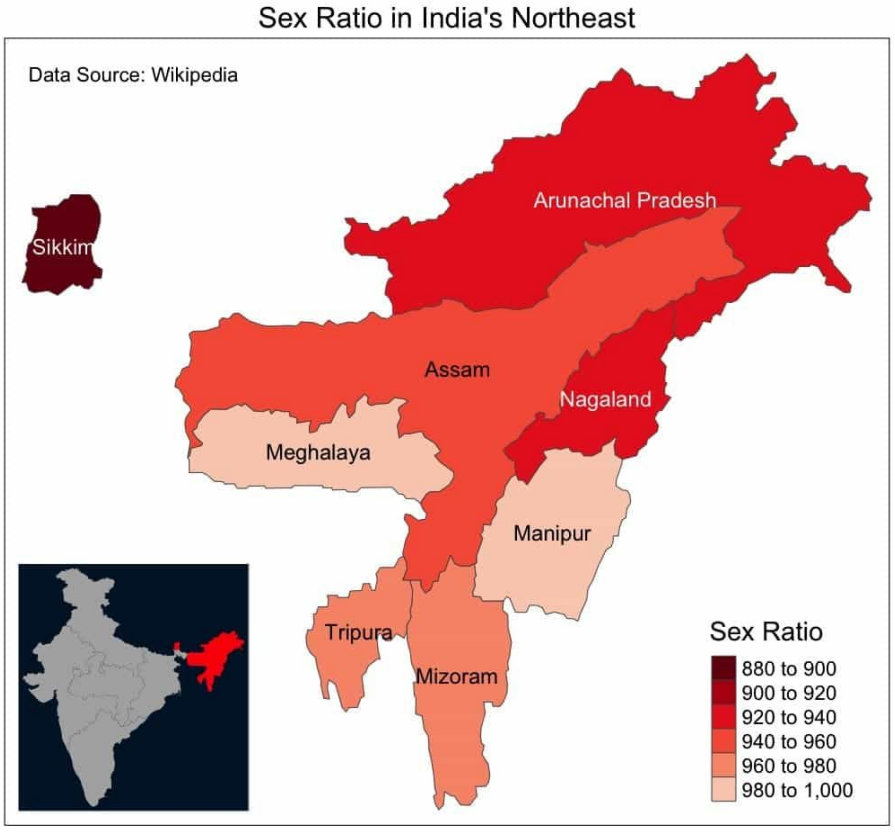
These 8 regional features have a geometry reflecting the "sum" of their individual sub-components. It is quite interesting that this kind of geometric operation can be done so easily by using `st_unify()` behind the scenes.

```
regional_sf <- simp_sf %>%
     group_by(region) %>%
     summarise(pop = sum(pop_2011))

inset <- regional_sf %>%
     filter(!region == "Arabian Sea",
            !region == "Bay of Bengal") %>%
     mutate(northeast = ifelse(region == "Northeastern", TRUE, FALSE))
%>%
```

```
    tm_shape() +
    tm_fill(col = "northeast", palette = c("grey", "red")) +
    tm_style("cobalt") +
    tm_legend(show = FALSE)
```

Once we have a base map and an inset map, we can combine them with the following syntax, using some trial and error to get the placement right.

```
library(grid)
ne_sex
print(inset, vp = viewport(0.24, 0.18, width = 0.2, height = 0.4))
```



Faceted Maps

`tmap` also supports the creation of **faceted maps**, or small multiples. They can be useful for attributes with a fairly small number of levels. For instance, if we have population data for a few years, we could show a progression over time. In this case, region is a useful variable for faceting. Splitting up a map by region can sometimes highlight contrasts better than looking at one image of the entire area.

The `free.coords` argument controls whether to show only the faceted map area or instead highlight the facet's place in the original map.

There is no inherent order to regions, but it's useful to impose one. Below we've ordered the facets in a roughly counter-clockwise order starting from "Northern". To do this, it helps to first make `region` an ordered factor.

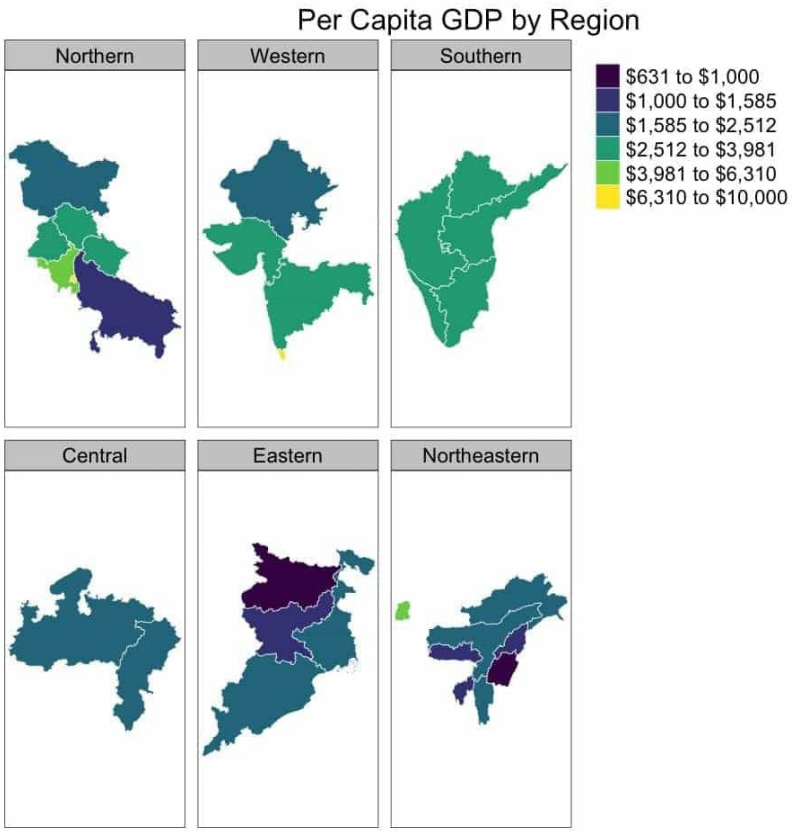
It's also important to pay attention to the nature of the distribution before making any plot. In India, per capita GDP is highly skewed because of outliers like Goa and Delhi. If you map this data on a linear scale, most states may end up the same color. This will conceal important differences in the bulk of the data.

If you have highly skewed data, it may be helpful to perform a logarithmic transformation. This should add greater color differentiation in the map, though the legend may require more careful interpretation because the color bins aren't of equal width.

> LEARN MORE:
>
> For more information about statistical transformations in the context of data visualization, see Section 8.2 of Wilke's book.

```
# create custom labels for log scale
gdp_seq <- 10 ^ (seq(2.8, 4.0, by = 0.2))
gdp_vec <- scales::dollar(round(gdp_seq))

my_labels = vector(mode = "character", length = 6)
for (i in seq_along(1:6)) {
     my_labels[i] = str_c(gdp_vec[i], " to ", gdp_vec[i + 1])
}

simp_sf %>%
     mutate(
          log_pc_usd = log10(per_capita_gdp_usd),
          region_fac = factor(region, levels = c("Northern", "Western",
"Southern",
                                                 "Central", "Eastern", "Nor
theastern",
                                                 "Arabian Sea", "Bay of Ben
gal"))
     ) %>%
```

```
      filter(!state_ut %in% c("Andaman & Nicobar Islands",
                              "Lakshadweep")) %>%
      tm_shape() +
      tm_borders(lwd = 0.5, col = "white") +
      tm_fill(col = 'log_pc_usd', title = '', palette = "viridis",
              labels = my_labels) +
      tm_facets(by = "region_fac", nrow = 2, free.coords = TRUE) +
      tm_layout(
           main.title = "Per Capita GDP by Region",
           main.title.size = 1,
           main.title.position = "center",
           legend.outside.position = "right"
      )
```



In the example above, a `log10()` transformation has hopefully achieved a greater level of differentiation. You can, for instance, differentiate between the small bright yellow dots of Goa and Delhi, the blue of Uttar Pradesh and Jharkhand, the turquoise of Central India and Rajasthan, and the green of South India.

> NOTE:
>
> The `free.coords()` argument of `tm_facets()` is set to `TRUE`. If it was instead set to `FALSE`, the entire map of India would appear in each facet with the given region highlighted.

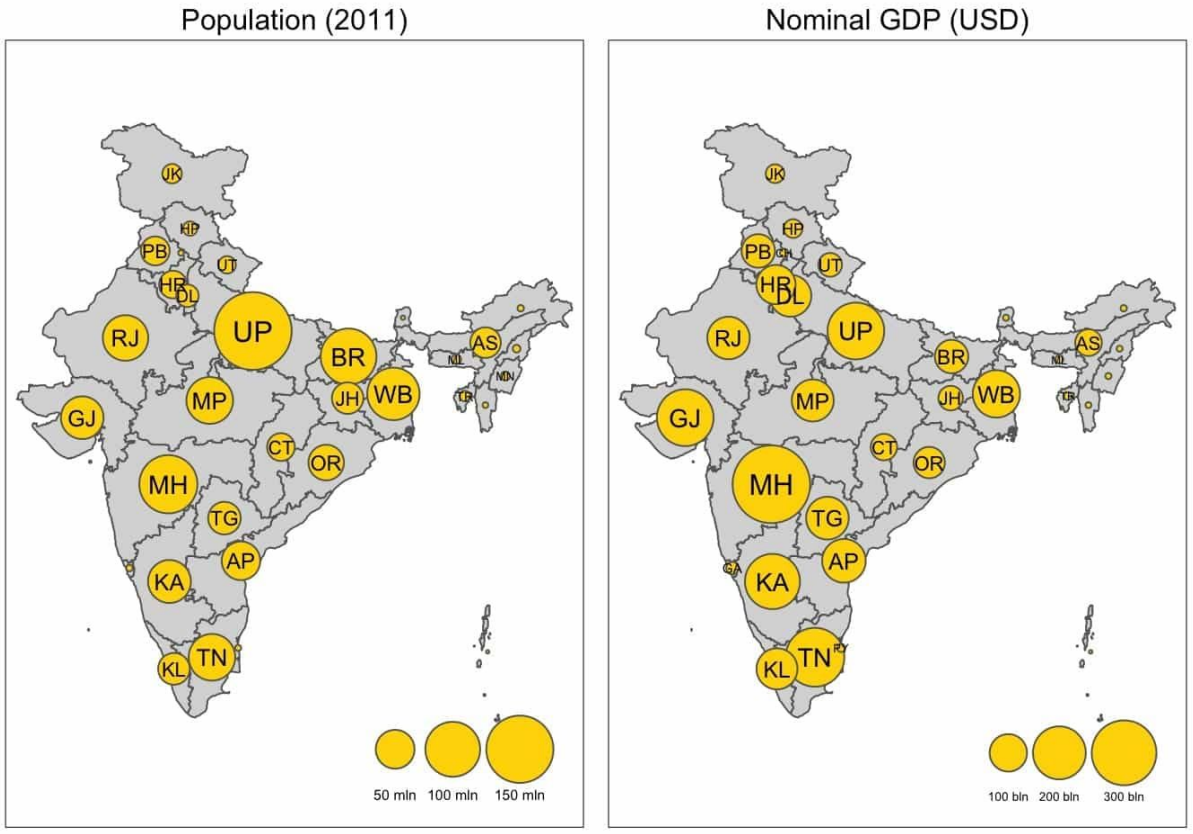
Proportional Symbols Maps

So far, all of our maps have been choropleths. This was convenient because our data was always standardized in some way — a density, percentage or ratio for example. Choropleths, however, are poorly suited to raw count data. When dealing with count data, such as population, a **proportional symbols map** can be more effective.

Luckily, `tmap` is also well-suited to these types of visualizations. Here, a symbol (typically a circle) is drawn in proportion to the depicted variable on top of the original geography. We can use this kind of map to visualize both population and nominal GDP data.

```
pop_bubbles <- simp_sf %>%
      tm_shape() +
      tm_polygons() +
      tm_bubbles(col = "gold", size = "pop_2011",
                 scale = 3, title.size = "") +
      tm_text("abb", size = "pop_2011", root = 5,
              legend.size.show = FALSE) +
      tm_layout(
            main.title = "Population (2011)",
            main.title.position = c("center"),
            main.title.size = 1,
            legend.position = c("right", "bottom")
      )

gdp_bubbles <- simp_sf %>%
      tm_shape() +
      tm_polygons() +
      tm_bubbles(col = "gold", size = "nominal_gdp_usd",
                 scale = 3, title.size = "") +
      tm_text("abb", size = "nominal_gdp_usd", root = 5,
              legend.size.show = FALSE) +
```

```
    tm_layout(
        main.title = "Nominal GDP (USD)",
        main.title.position = c("center"),
        main.title.size = 1,
        legend.position = c("right", "bottom")
    )

tmap_arrange(pop_bubbles, gdp_bubbles)
```



The proportional symbols map retains the original geography, only obscured by the top layer of symbols. The symbols retain the correct spatial arrangement and are easy to interpret in relationship to each other. Still, judging the area of circles is more difficult than compared to a non-spatial representation, such as a barplot.

geom_sf in ggplot2

Hopefully these examples have demonstrated that `tmap` is a robust mapping tool. At the same time, the addition of `geom_sf` has made **`ggplot2`** another attractive option.

`ggplot2` requires tidy data. Since spatial dataframes defined in the `sf` package are dataframes, it makes sense that we could expect to use `ggplot2` to visualize `sf`

objects. Recently, `ggplot2` added support for `sf` objects with `geom_sf()`. The key advantage of `geom_sf()` is that tidyverse users are already familiar with `ggplot2` and its wider ecosystem of add-on packages.

However, as this is a recent addition, you might expect a few bugs. For example, in the above faceted `tmap` object, setting `free.coords = FALSE` allowed for the entire object to be plotted in each facet. At this time, faceting an `sf` geom doesn't seem to allow setting `scales = "free"` to allow a similar outcome.

Nevertheless, there are many benefits and cases where we can visualize `sf` objects with `ggplot2`. For instance, `ggplot2` users will be familiar with the process of mapping data from dataframes to aesthetics, and layering additional dataframes on top of a plot. That same workflow holds for plotting `sf` objects.

In the plot below, we want to add only the state name "Kerala" to the map. We could have done it with the `annotate()` function, but instead we created an `sf` object (also a dataframe) holding only the feature we wanted to annotate (Kerala).

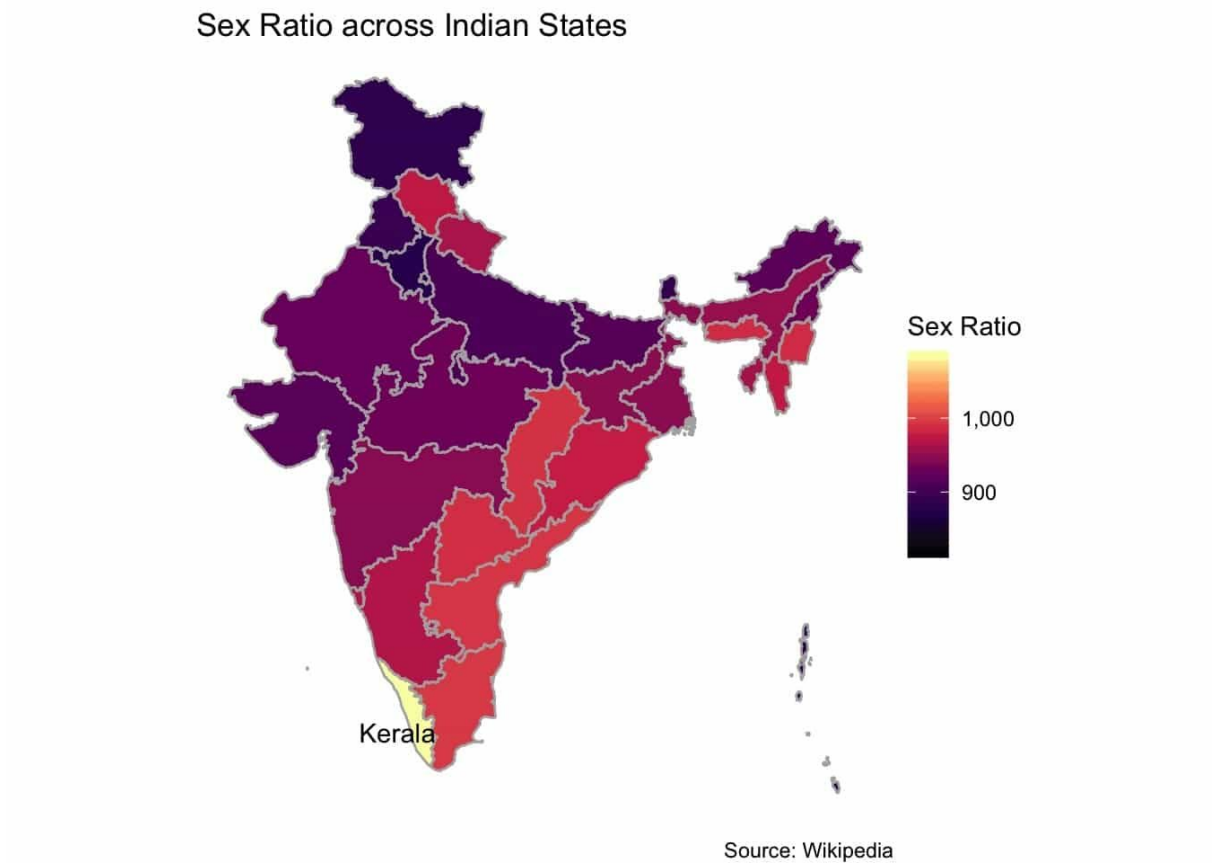
In order to do this successfully, however, we first need to find the geographic center of Kerala to know the point from which to draw the label. Geometric operations like calculating centroids, buffers and distance require a projected CRS as opposed to a geographic CRS, and so we've done so below using `st_transform()`.

With a geographic CRS, `st_centroid()` does produce a result, but it produces a warning that *"`st_centroid` doesn't give correct centroids for longitude/latitude data"* because it assumes attributes are constant over geometries. The distance between longitudes, however, changes based on its given latitude. (Think of the distance between longitudes at the equator vs. at the North Pole.)

The question then becomes choosing an appropriate projected CRS. Viewing `crs_data = rgdal::make_EPSG()` shows thousands of options. We also searched for "India" at EPSG.io. Ultimately we chose a CRS with EPSG code 24343, which notes *"# Kalianpur 1975 / UTM zone 43N"* since UTM zone 43N covers Kerala. (You might also find Projection Wizard useful.)

Using this CRS, we were able to use `st_transform()` to project both `sf` objects onto the same projected CRS. Once that was done, we could add the Kerala label using `geom_text_repel()` like we'd normally do in `ggplot2`.

```
library(ggplot2)
library(ggrepel)

proj_sf <- simp_sf %>%
     st_transform(crs = 24343) %>%
     mutate(
          CENTROID = purrr::map(geometry, st_centroid),
          COORDS = purrr::map(CENTROID, st_coordinates),
          COORDS_X = purrr::map_dbl(COORDS, 1),
          COORDS_Y = purrr::map_dbl(COORDS, 2)
     )

kerala <- proj_sf %>%
     filter(state_ut == "Kerala")

proj_sf %>%
     filter(!state_ut %in% c("Daman & Diu", "Dadra & Nagar Haveli")) %
>%
     ggplot() +
     geom_sf(aes(fill = sex_ratio), lwd = 0) +
     geom_sf(fill = NA, color = "grey", lwd = 0.5) +
     scale_fill_viridis_c("Sex Ratio", labels = scales::comma, option =
"A") +
     labs(
          title = "Sex Ratio across Indian States",
          caption = "Source: Wikipedia"
     ) +
     geom_text_repel(
          data = kerala,
          mapping = aes(x = COORDS_X, y = COORDS_Y, label = state_ut),
          nudge_x = -0.5,
          nudge_y = -1
     ) +
     scale_y_continuous(NULL) +
     scale_x_continuous(NULL) +
     theme(plot.title = element_text(hjust = 0.5)) +
     # remove graticules
     coord_sf(datum = NA) +
     theme_void()
```



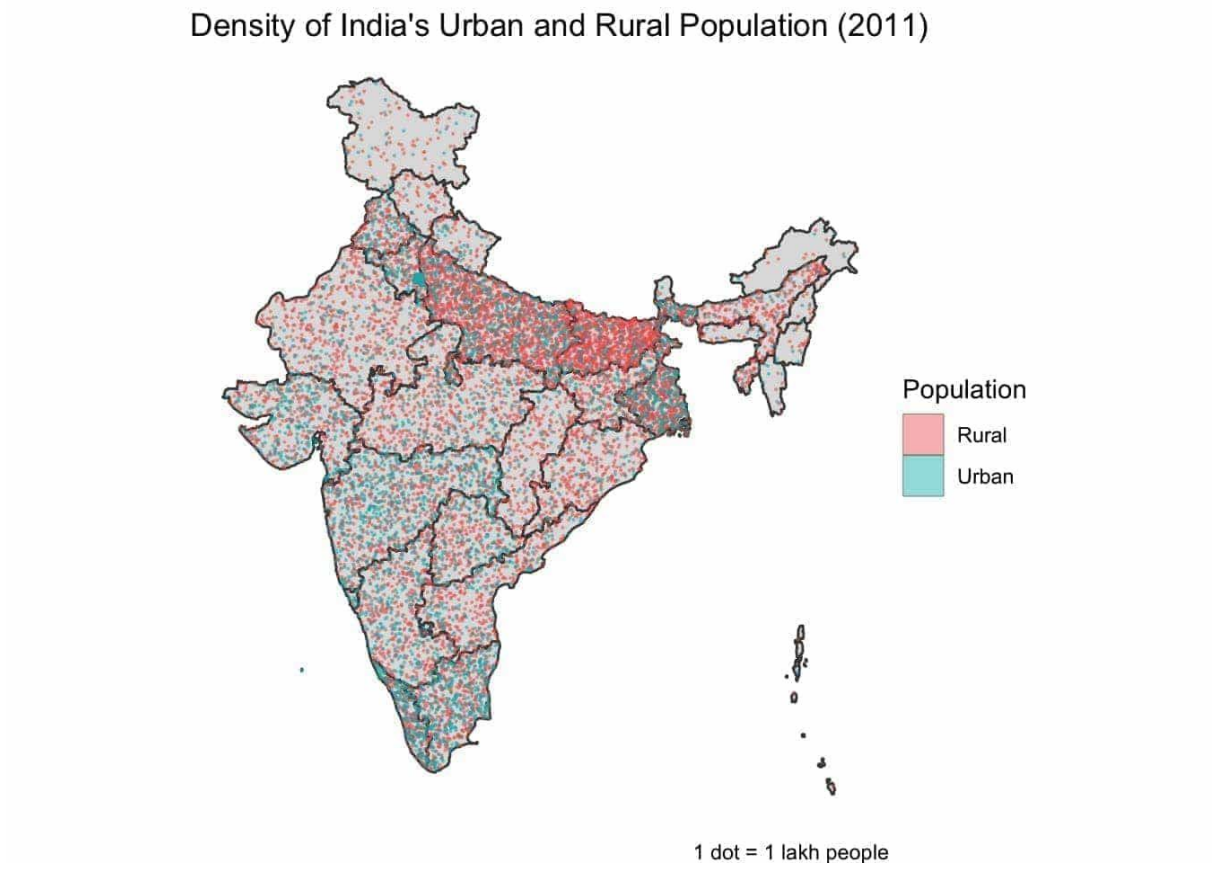
Dot Density Maps

Proportional symbols maps are not the only option for raw count data. A **dot density map** can be an effective tool to spatially visualize count data, particularly when your goal is to find clusters and regional patterns instead of exact data values.

Below we've created a dot density plot comparing rural and urban populations. To do this, first, we depart from a tidy data format and `gather()` urban and rural population data. Then we use the `st_sample()` function to draw sample points based on the respective urban and rural population data for each observation.

This type of visualization would be much more effective if we had data at smaller levels of administration, such as districts. Instead, because we are sampling at the state level, our dots will be placed in locations counter to the actual population density. For example, Maharashtra's urban population will be randomly spread throughout the state instead of clustering in metros like Mumbai. Nevertheless, it's still useful to see how such a map can be created.

```
# save geometry
proj_geometry <- proj_sf %>% select(state_ut)

# gather data and rejoin geometry
pop_gathered <- proj_sf %>%
     st_set_geometry(NULL) %>%
     select(state_ut, rural_pop, urban_pop) %>%
     gather(key = "pop", value = "count", -state_ut) %>%
     arrange(state_ut) %>%
     left_join(proj_geometry) %>%
     st_as_sf()

# create a list of urban and rural populations
pop_split <- pop_gathered %>% split(.$pop)

# draw 1 dot per 1 lakh people
generate_samples <- function(data) {
     st_sample(data, size = round(data$count / 1e5))
}

# generate samples for each and combine
points <- map(pop_split, generate_samples)
points <- imap(points, ~st_sf(tibble(
                    pop = rep(.y, length(.x))),
                    geometry = .x))
points <- do.call(rbind, points)

# group points into multipoints
points <- points %>%
     group_by(pop) %>%
     summarise()

# plot with ggplot
points %>%
     ggplot() +
     geom_sf(data = simp_sf) +
     geom_sf(aes(color = pop, fill = pop),
             size = 0.1, alpha = 0.4) +
     scale_fill_discrete("Population", labels = c("Rural", "Urban")) +
     labs(
          title = "Density of India's Urban and Rural Population (201
1)",
          caption = "1 dot = 1 lakh people"
     ) +
     theme(plot.title = element_text(hjust = 0.5)) +
     coord_sf(datum = NA) +
     theme_void() +
     guides(color = FALSE)
```



While we can't see population clusters around metros, we can still see relatively sparsely populated areas (Jammu & Kashmir and the Northeast), the extremely dense rural belt of Uttar Pradesh and Bihar, and the relatively more urban South India.

Moreover, as we'll see in the next lesson, this is one example of a visualization where adding interactivity — specifically, the ability to separately plot urban and rural data — can be a real benefit.

> LEARN MORE:
>
> For more details on generating dot density plots in R, see these excellent blogs from Tarak and Paul Campbell.

Partially Spatial Static Representations

At the beginning of this lesson, we discussed how a choropleth can misrepresent data if there are large differences between the area of an observational unit (e.g. a

state) and its population. When data is only partially spatial, a map may not always the best visualization, depending on your objectives.

If you don't need a fully spatial representation of your data, there are other visualization options that communicate some spatial aspects of the data, but diverge in some aspect or another. Examples include cartograms, hexbin maps and geofaceted plots.

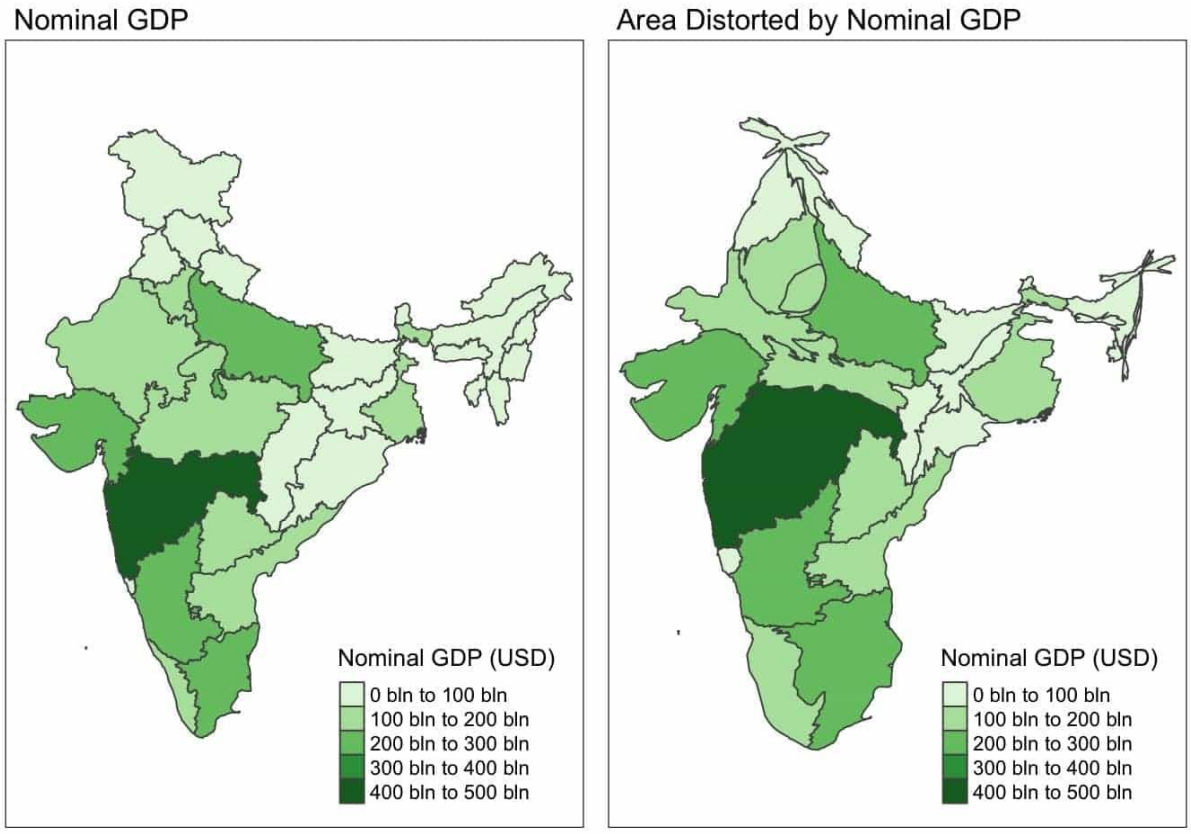
Cartograms

In a **cartogram**, we maintain the overall geospatial nature of an object, but distort the area of each observational unit so that each unit is scaled proportional to some chosen variable.

Nominal GDP is a useful variable to demonstrate this relationship. For example, in relation to its very small geographic area, Delhi's contribution to India's GDP is very high. A traditional choropleth (shown on the left) fails to make this distinction. Using a cartogram (shown on the right), we can distort a state's geographic area to match its contribution to India's GDP.

```
library(cartogram)

ccart_gdp_sf <- cartogram_cont(proj_sf, "nominal_gdp_usd")

gdp_ccart <- ccart_gdp_sf %>%
     filter(!state_ut == "Andaman & Nicobar Islands") %>%
     tm_shape() +
     tm_polygons("nominal_gdp_usd", title = "Nominal GDP (USD)",
                 palette = "Greens") +
     tm_layout(
          main.title = "Area Distorted by Nominal GDP",
          main.title.position = c("left"),
          main.title.size = 1,
          legend.position = c("right", "bottom")
     )

gdp_original <- proj_sf %>%
     filter(!state_ut == "Andaman & Nicobar Islands") %>%
     tm_shape() +
     tm_polygons(col = "nominal_gdp_usd", title = "Nominal GDP (USD)",
                 palette = "Greens") +
     tm_layout(
          main.title = "Nominal GDP",
          main.title.position = c("left"),
          main.title.size = 1,
```

```
            legend.position = c("right", "bottom")
      )

tmap_arrange(gdp_original, gdp_ccart)
```

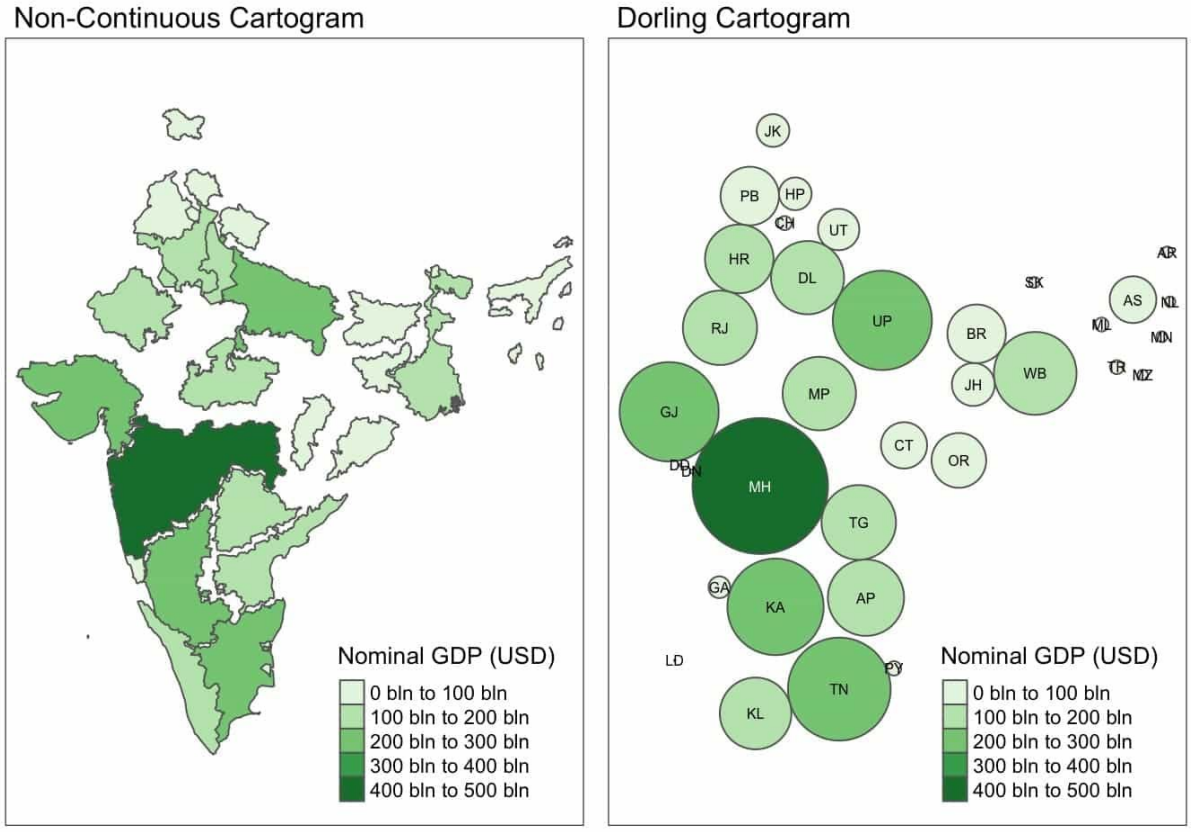


While still maintaining the overall geographic structure of India, we can vividly see which states shrink or expand when distorting geographic area by the size of nominal GDP. (For the same reasons discussed above, we need to use a projected CRS rather than a geographic one.)

Above we created a continuous cartogram with the `cartogram_cont()` function. However, as shown below, we could have also just as easily chosen a non-continuous area cartogram using `cartogram_ncont()`, which would introduce separation between states, or a Dorling cartogram using `cartogram_dorling()`, which would represent each state as a circle.

```
ncart_gdp_sf <- cartogram_ncont(proj_sf, "nominal_gdp_usd")
dorling_gdp_sf <- cartogram_dorling(proj_sf, "nominal_gdp_usd")

gdp_ncart <- ncart_gdp_sf %>%
     filter(!state_ut == "Andaman & Nicobar Islands") %>%
```

```
      tm_shape() +
      tm_polygons("nominal_gdp_usd", title = "Nominal GDP (USD)",
                  palette = "Greens") +
      tm_layout(
            main.title = "Non-Continuous Cartogram",
            main.title.position = c("left"),
            main.title.size = 1,
            legend.position = c("right", "bottom")
      )

gdp_dorling <- dorling_gdp_sf %>%
      filter(!state_ut == "Andaman & Nicobar Islands") %>%
      tm_shape() +
      tm_polygons("nominal_gdp_usd", title = "Nominal GDP (USD)",
                  palette = "Greens") +
      tm_text("abb", size = 0.5) +
      tm_layout(
            main.title = "Dorling Cartogram",
            main.title.position = c("left"),
            main.title.size = 1,
            legend.position = c("right", "bottom")
      )

tmap_arrange(gdp_ncart, gdp_dorling)
```



Given that cartograms are already distorting the actual geographic shapes, it's often not necessary to keep one continuous unit. In such cases, a non-continuous cartogram may be preferable. Alternatively, we can go even more abstract and replace all geographic shapes with a simple circle, scaled to the parameter of interest.

> NOTE:
>
> The Dorling cartogram is essentially the proportional symbols map without the underlying map.

Hexbin Maps

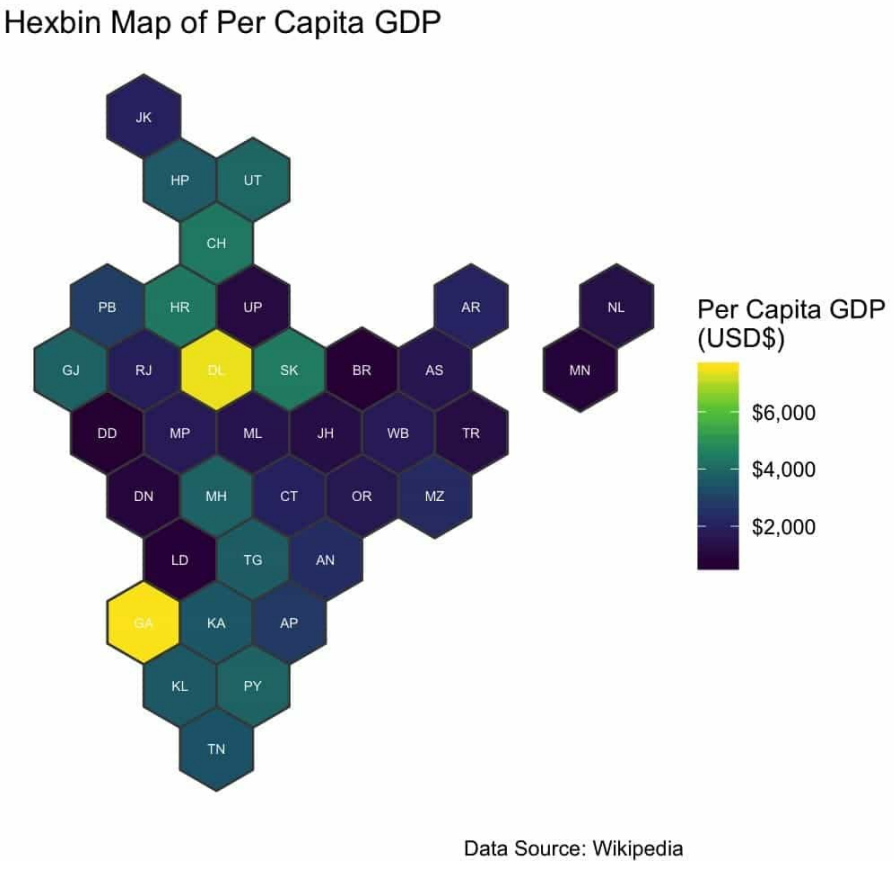
Similar to a Dorling cartogram, a **hexbin map** also replaces exact spatial boundaries with a rough spatial arrangement. However, instead of mapping the variable of interest to size, it's mapped to color.

Hexbin grids for the United States and a few other countries are well established, but `geogrid` is a new package under development that tries to generate automatic hexbin grids given any set of geospatial polygons. Although the package lets you generate a number of possible grids and select the best option, we had trouble generating a map that adequately placed certain states, the Northeast and non-contiguous territories in particular.

Nevertheless, the hexbin map below gives a sense of why reducing geospatial polygons to a hexagon can be more useful, in certain cases, than the original geometry.

```
library(geogrid) # devtools::install_github("jbaileyh/geogrid")

## test possible grids before selecting seed
# par(mfrow = c(3, 3), mar = c(0, 0, 2, 0))
# for (i in 1:9) {
#     new_cells <- calculate_grid(shape = proj_sf,
#                                 grid_type = "hexagonal", seed = i)
#     plot(new_cells, main = paste("Seed", i, sep = " "))
# }
```

```
new_cells_hex <- calculate_grid(shape = proj_sf,
                                grid_type = "hexagonal", seed = 1)
hex_result <- assign_polygons(proj_sf, new_cells_hex)

# assign_polygons generates V1 V2 which are center coordinates of tiles
ggplot(hex_result) +
     geom_sf(aes(fill = per_capita_gdp_usd)) +
     geom_text(aes(x = V1, y = V2,
                   label = abb), size = 2, colour = "white") +
     scale_fill_viridis_c("Per Capita GDP\n(USD$)", labels = scales::do
llar) +
     labs(
          title = "Hexbin Map of Per Capita GDP",
          caption = "Data Source: Wikipedia"
     ) +
     coord_sf(datum = NA) +
     theme_void() +
     guides(size = FALSE)
```



The downside of this visualization is that it gives equal area to all states. Tiny union territories are represented by the same area as Uttar Pradesh. However, if we understand that context in advance, this can be a useful visualization if we only want a distribution of per capita GDP across states, regardless of population or area.

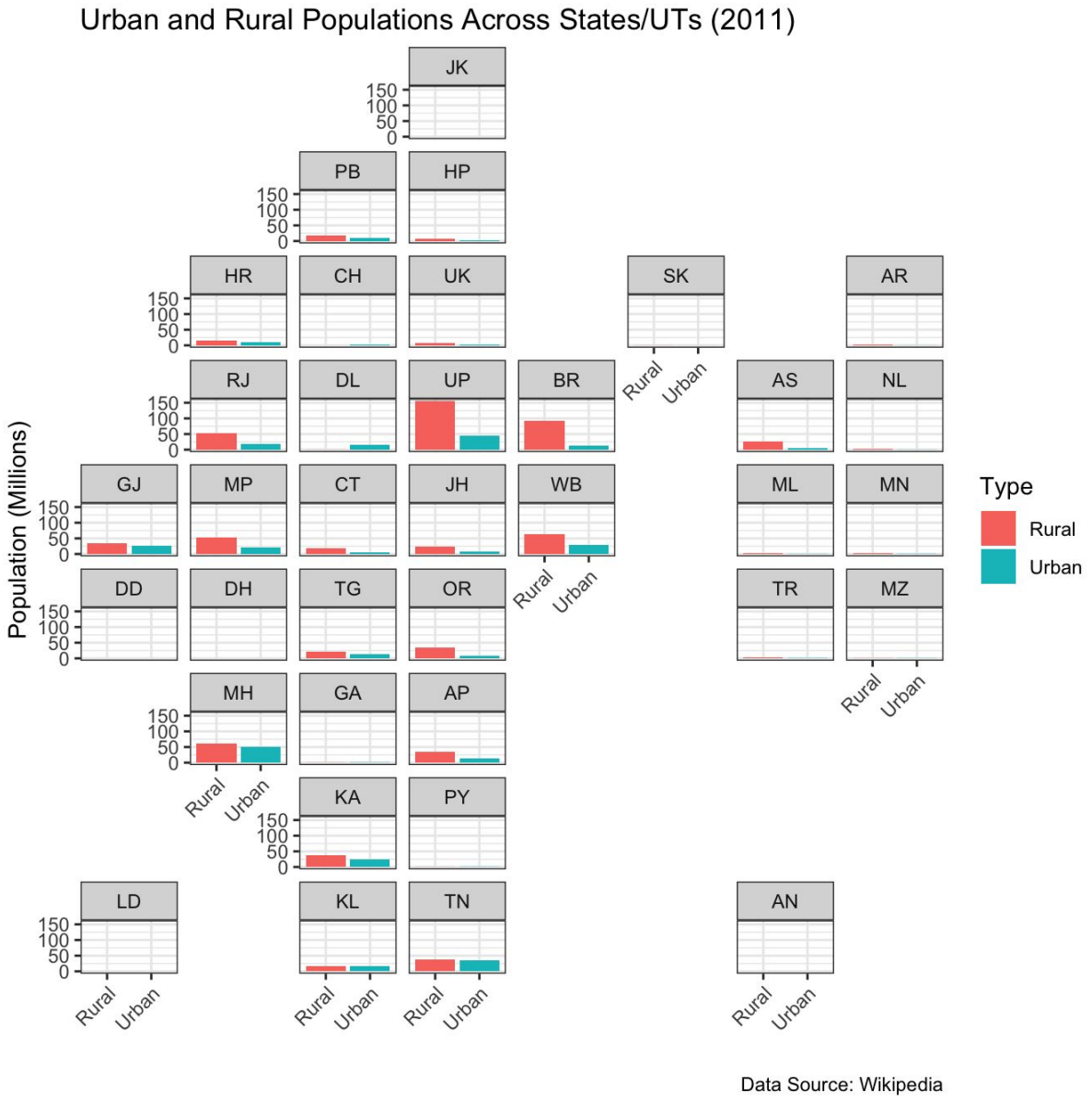
> NOTE:
>
> The package's README documents the process for generating geogrids using an `sp` class, but it's even simpler for an `sf` object (as shown above).

Geofaceted Plots

Similar to a hexbin map, a **geofaceted plot** sacrifices exact spatial characteristics in favor of a loose spatial arrangement. It strongly prioritizes accurate presentation of the attribute data at obvious cost to the geospatial representation.

The **`geofacet`** package makes it easy to design a custom grid and use it to facet data across the grid.

```
library(geofacet)

simp_df <- simp_sf %>%
     st_set_geometry(NULL) %>%
     select(state_ut, urban_pop, rural_pop) %>%
     gather(Type, pop_value, -state_ut) %>%
     mutate(Type = ifelse(Type == "urban_pop", "Urban", "Rural"))

ggplot(simp_df,
       aes(x = Type, y = pop_value / 1e6, fill = Type)) +
     geom_col() +
     facet_geo(~ state_ut, grid = mygrid, label = "code") +
     labs(
          title = "Urban and Rural Populations Across States/UTs (201
1)",
          caption = "Data Source: Wikipedia",
          x = "",
          y = "Population (Millions)"
     ) +
     theme_bw() +
     theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



Although it doesn't look much like an Indian state map anymore, this visualization does vividly communicate the vast differences in urban and rural populations across states. It also highlights how the choice of visualization affects our interpretation.

> NOTE:
>
> The dot density map created above comes from the exact same data.

Final Thoughts

With the help of packages `tmap`, `ggplot2`, `cartogram`, `geogrid` and `geofacet`, this lesson has introduced some of the most common methods for creating various kinds of static geospatial visualizations in R, such as choropleths, dot density maps and cartograms.

Interested in going beyond static maps and exploring the world of animated or interactive maps? Check out the next lesson in this course.

View this lesson online

LESSON 4

Creating Animated and Interactive Maps in R

This lesson was written by Sean Angiolillo and was last updated on 29 Jan. 2019.

Now that we've completed an overview of static mapping in R in the previous lesson, let's explore how to create animated and interactive maps. We'll then conclude by creating a Shiny app to show the potential of visualizing geospatial data through interactive web applications in R.

Adding animation or interaction to a static map creates opportunities for stories and experiences that are not otherwise possible in a static world. Despite this power, it's important to introduce these elements in the right circumstances. Animated and interactive maps both demand a higher level of attention from the user; without this, the visualization won't be as clear or meaningful.

Animated Maps

Animated maps are particularly well-suited for spatio-temporal data as they can show change of a variable over time, but they certainly have other uses as well.

This lesson introduces two methods for making animated maps: animated `tmaps` and the `gganimate` package. We'll also need the packages and objects from the previous lesson.

```
library(tidyverse)
library(sf)
library(tmap)
```

```
simp_sf <- readRDS("simp_sf.rds")

states_sf <- simp_sf %>%
     filter(!type == "Union Territory")

proj_sf <- simp_sf %>%
     st_transform(crs = 24343) %>%
     mutate(
          CENTROID = purrr::map(geometry, st_centroid),
          COORDS = purrr::map(CENTROID, st_coordinates),
          COORDS_X = purrr::map_dbl(COORDS, 1),
          COORDS_Y = purrr::map_dbl(COORDS, 2)
     )
```

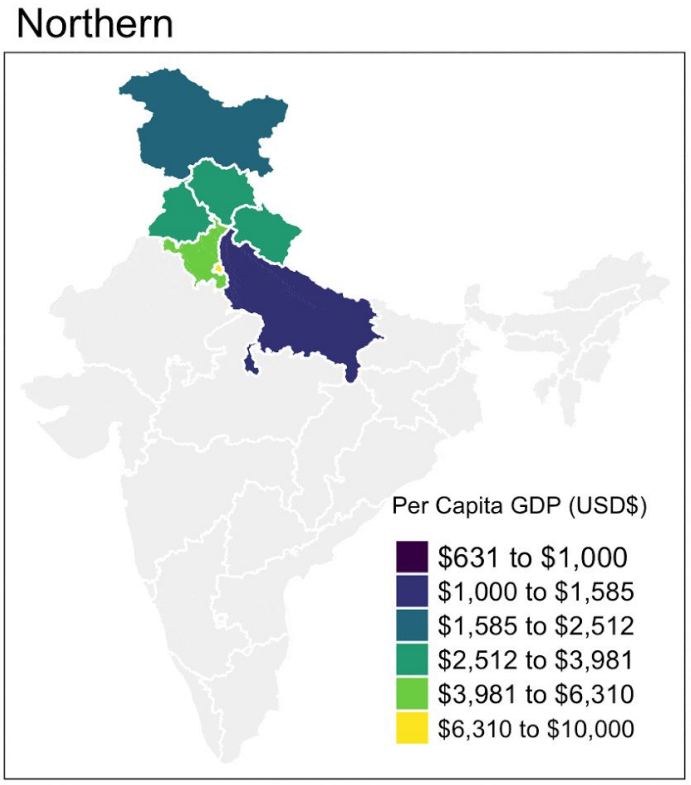
Animation with tmap

Perhaps the simplest way to create basic animated maps is through a slight modification to a faceted **`tmap`** plot. Simply change the `tm_facets()` argument `by` to `along`. (ImageMagick is required.) Then, providing the generated output to the `tmap_animation()` function loops each faceted plot into a gif or mpeg animation. You can specify further details like the delay of each frame and the output dimensions.

We can see an example of how this works by turning the earlier faceted plot of per capita GDP by region into an animated gif. Instead of showing all facets at once in a grid, we've looped each image into a gif, displaying them one at a time.

```
pc_gdp_anim <- simp_sf %>%
     filter(!state_ut %in% c("Lakshadweep", "Andaman & Nicobar Island
s")) %>%
     mutate(
          log_pc_usd = log10(per_capita_gdp_usd),
          region_fac = factor(region, levels = c("Northern", "Western"
, "Southern",
                                                 "Central", "Eastern", "Northeast
ern"))
     ) %>%
     tm_shape() +
     tm_fill(col = 'log_pc_usd', title = 'Per Capita GDP (USD$)',
             palette = "viridis",
             labels = my_labels) +
     tm_borders(col = "white") +
     tm_facets(along = "region_fac", free.coords = FALSE) +
     tm_layout(main.title.size = 1)
```

```
tmap_animation(pc_gdp_anim, filename = "pc_gdp_anim.gif",
               delay = 200, restart.delay = 200)
```



Animation with gganimate

For more robust animations (not only geospatial), look to the recently rewritten **`gganimate`** package from Thomas Lin Pedersen. The package allows not just for looping of facets but also a huge array of possibilities. This leads to some really creative plots, as found in the package's wiki.

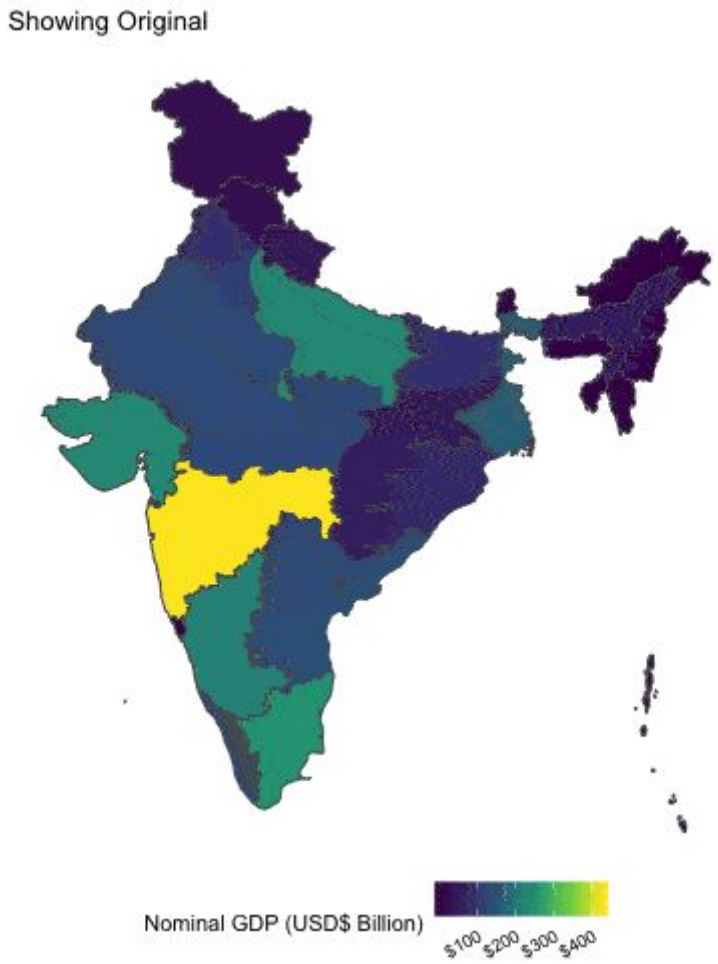
> LEARN MORE:
>
> Check out Pedersen's theory on establishing a grammar of animation in a useR keynote here.

Let's mimic an example that Pederson used in his keynote. After binding together transformed data sets into one object, we can achieve the animation below with just one additional line to our plot: `transition_states()`, which mimics

`ggplot2::facet_wrap()` by splitting the data into multiple panels, tweening between defined states and pausing at each state for a specified period.

Instead of visualizing per capita GDP as the previous animation did, let's visualize nominal GDP. Each frame may not be the most effective visualization, but animating them together is certainly attention-grabbing.

```
library(gganimate)
states <- c(
     'Original',
     'Continuous Cartogram Weighted by Nominal GDP',
     'Dorling Cartogram Weighted by Nominal GDP',
     'Hexagonal Tiling'
)

# cartograms, hexbin created in previous lesson

proj_sf$state <- states[1]
ccart_gdp_sf$state <- states[2]
dorling_gdp_sf$state <- states[3]
hex_result$state <- states[4]

nom_gdp_all <- rbind(proj_sf,
                     ccart_gdp_sf[, names(proj_sf)],
                     dorling_gdp_sf[, names(proj_sf)],
                     hex_result[, names(proj_sf)])
nom_gdp_all$state <- factor(nom_gdp_all$state, levels = states)

ggplot(nom_gdp_all) +
     geom_sf(aes(fill = nominal_gdp_usd / 1e9, group = state_ut)) +
     scale_fill_viridis_c(labels = scales::dollar) +
     coord_sf(datum = NA) +
     theme_void() +
     theme(legend.position = 'bottom',
           legend.text = element_text(angle = 30, hjust = 1)) +
     labs(title = 'Showing {closest_state}',
          fill = 'Nominal GDP (USD$ Billion)') +
     transition_states(state, 2, 1)
anim_save("nom_gdp_anim.gif")
```



Interactive Maps

Interactive maps have perhaps even greater storytelling potential than animations as they allow the user, in some respects, to create their own narrative. Certain features, such as panning and zooming, allow a level of freedom just not possible with a static or even animated map.

Nevertheless, it's important to be careful to only use them when necessary. An interactive map requires a higher level of attention from the user than a static or an animated map. The users must interact with the visualization! If they don't (or not in the way intended), then what the map's designer was trying to communicate is lost. That's why it's important to think about what you're trying to achieve and why a static (or even animated) map isn't sufficient.

If you need interaction, there are many levels of interaction to consider. In the case of choropleths, often a simple tooltip with the exact value the color represents can add value. But, of course, interaction can accomplish much more. For example, we can change base maps to plot different kinds of geography. We can let users select

and filter their own data to be plotted. Users can also modify other aspects of a plot, such as the color scheme or statistical transformations. Interactive maps also give users opportunities for brushing and linking. Based on user input or selection, we can design a reaction in another view. These are just a few of the possibilities, and so it's important to think carefully about what you actually need.

This section introduces the `ggiraph` package, `tmap`'s view mode, `mapview`, `leaflet`, and `plotly`.

Interactivity with ggiraph

Let's start simple. In some cases, we may only want to add a minimal amount of interactivity (such as a hover or tooltip effect). In that case, we could turn to the `plotly` or `ggiraph` packages.

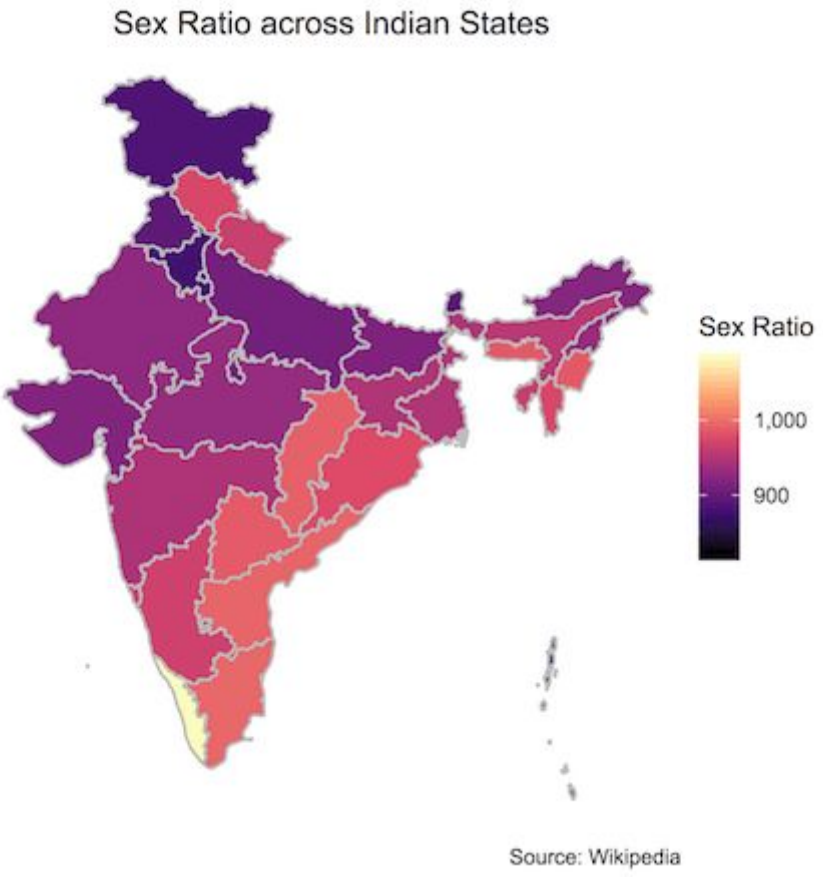
The **`plotly`** package is a popular choice for adding interactivity to `ggplot2` plots. The `ggplotly()` function handles geospatial data in the same way as non-spatial data. You might, however, find this strategy to be a bit of a hassle, often introducing slight distortions, when you're only after a simple tooltip on hover. However, its more advanced features (discussed later) are quite useful.

For simple interactivity, you might try the **`ggiraph`** package from David Gohel. The basic idea is to pass `ggplot()` an interactive geom in place of a traditional geom. In the case of maps, this means using `geom_sf_interactive()` instead of geom_sf(). After specifying additional arguments like `tooltip`, `onclick` and `data_id`, simply call `ggiraph` on the saved `gg` object.

The code below adds a simple tooltip that includes state name and sex ratio when hovering over the previous sex ratio map.

```
library(ggiraph)

tooltip_css <- "background-color:gray;color:white;padding:5px;border-r
adius:5px;font-family:sans-serif;font-size:12px;"

gg_sex <- simp_sf %>%
     mutate(
          tip = str_c(
          "<b>", state_ut, " : ", sex_ratio, "</b>",
          "</span></div>")
     ) %>%
```

```
      filter(!state_ut %in% c("Daman & Diu", "Dadra & Nagar Haveli")) %
>%
      ggplot() +
      geom_sf_interactive(aes(fill = sex_ratio,
                              tooltip = tip, data_id = state_ut)) +
      geom_sf(fill = NA, color = "grey", lwd = 0.5) +
      scale_fill_viridis_c("Sex Ratio", labels = scales::comma, option
= "A") +
      labs(
            title = "Sex Ratio across Indian States",
            caption = "Source: Wikipedia"
      ) +
      coord_sf(datum = NA) +
      theme_void() +
      theme(plot.title = element_text(hjust = 0.5))

ggiraph(ggobj = gg_sex,
        hover_css = "cursor:pointer;stroke-width:5px;fill-opacity:0.8;"
,
        tooltip_extra_css = tooltip_css, tooltip_opacity = 0.75)
```



Click here to view the interactive map in a separate window.

Interactive tmaps

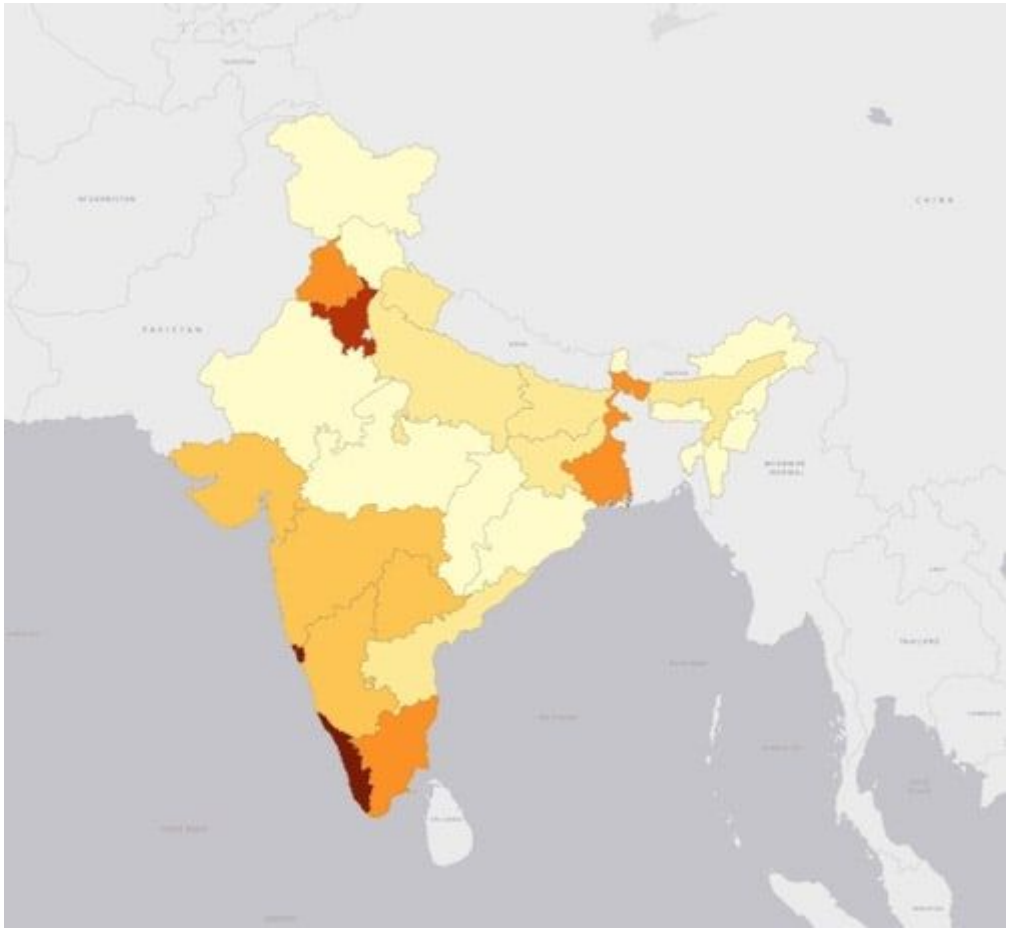
One of the strongest reasons to add interactivity to a static map is the ability to layer geospatial data (such as points or polygons) on top of base maps that depict physical or human geography. Perhaps the easiest way to achieve this advantage is by plotting `tmap` objects in `tmap`'s "view" mode.

By default, `tmap_mode()` is set to "plot", but changing this argument to "view" can make any `tmap` object an interactive plot. This option builds on top of Leaflet, which we'll cover below.

Alternatively, it's possible to convert `tmap` objects to Leaflet objects via the `tmap_leaflet()` function. This interactivity also applies to `tmap_arrange` objects, which is nice for having side-by-side interactive maps.

However, interactive `tmap` objects, particularly arranged `tmap` objects, seem to be difficult to modify after creation. For instance, the tooltip took the first column by default, so we edited the first column to include the tooltip we wanted. Legends also seemed difficult to move from the top right for some reason. Because of small issues like this, you might be better off directly building in Leaflet for more polished projects.

```
tmap_mode("view")
states_sf %>%
     mutate(
          gdp_density = gdp_density_usd_km2 / 1e6,
          label = str_c(state_ut, ": ", gdp_density)
     ) %>%
     select(label, everything()) %>%
     tm_shape() +
     tm_fill(col = 'gdp_density', title = "USD$(mil)/sq.km") +
     tm_borders(lwd = 0.5)
```

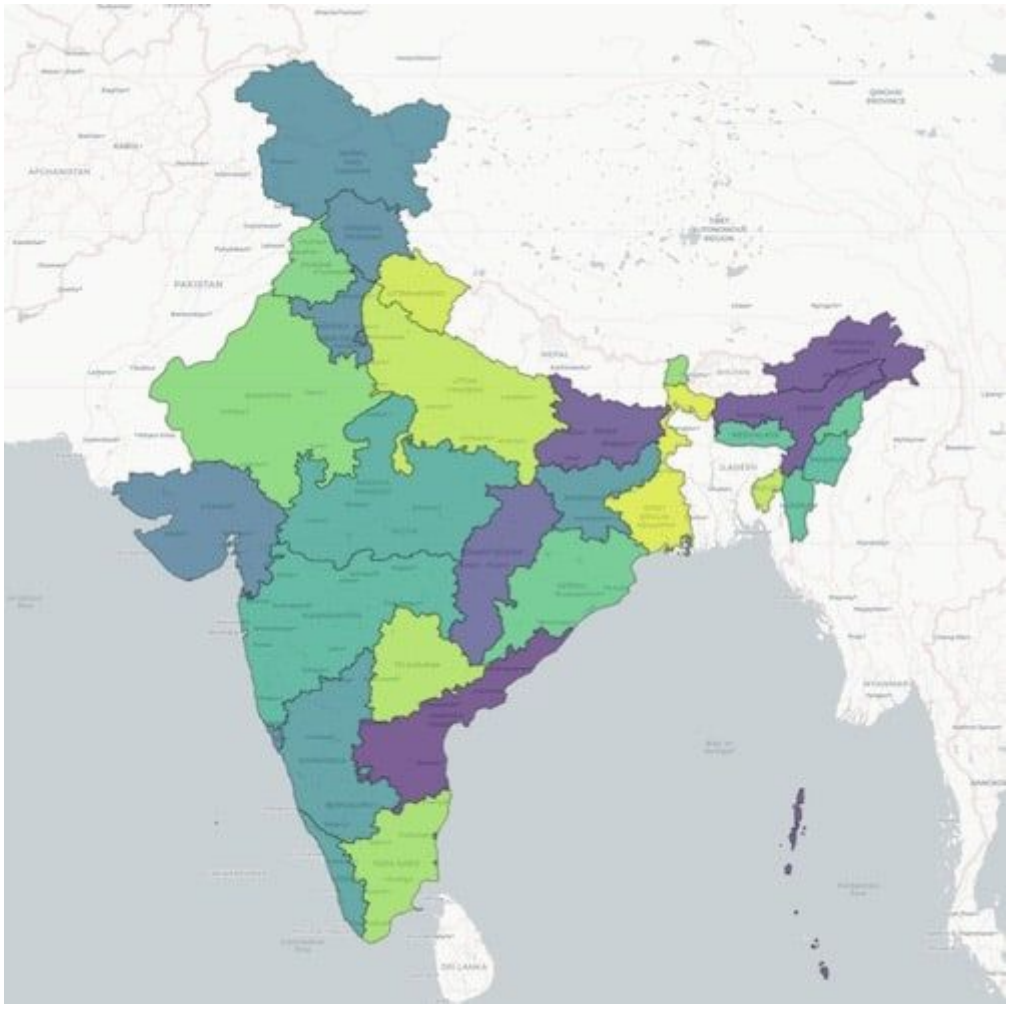


Click here to view the interactive map in a separate window.

Interactivity with mapview

Another package for interactive viewing of spatial data is **mapview**. It too is built on top of Leaflet. Its primary role is to create quick interactive geospatial visualizations, rather than presentation-quality visualizations — but its functionality may be growing. If we wanted to quickly compare a number of choropleths for different variables, it could be a good option.

```
library(mapview)
mapview(
      simp_sf,
      zcol = c("state_ut", "pop_2011", "per_capita_gdp_usd",
               "density_km2", "sex_ratio"),
      legend = FALSE,
      hide = TRUE
)
```



Click here to view the interactive map in a separate window.

> NOTE:
>
> Select one data layer at a time to view the correct choropleth.

Interactivity with Leaflet

Both `mapview` and the interactive mode of `tmap` rely on Leaflet under the hood. Eventually, you'll likely want to build directly in Leaflet.

Leaflet is a Javascript library for interactive maps. An R wrapper package of the same name from RStudio has made it very easy to create Leaflet maps in R. The `leaflet` package has a rich array of features including fully interactive panning and zooming, maps built from customizable tiles, and plotted layers and groups of

markers, polygons and popups. It is also very easy to embed Leaflet map widgets into RMarkdown documents, web pages or Shiny apps.

One of the key advantages of Leaflet is the ability to draw on a huge range of map tiles (which might include roads or natural features) that can lie underneath your data set. Moreover, if you want to represent layers of spatial data, for instance as points on top of polygons, then Leaflet is there for you.

The ability to interact with layers of spatial data comes in handy for enhancing our earlier dot density map of rural and urban population. To address the overplotting of dots, it can be useful to toggle between urban and rural populations, as shown below.

In order to achieve this effect in Leaflet, we first need to use the `st_cast()` function to convert our earlier multi-points into individual points because Leaflet doesn't support multi-point objects at this time. We also need to transform the projected CRS to a geographic CRS (longitude and latitude coordinates) for plotting in Leaflet. Then we establish each respective layer as a "group". Remember, here one dot equals one lakh (100,000) people.

```
library(leaflet)
library(leaflet.extras)

# multipoints are not supported so cast to points
my_points <- st_cast(points, "POINT")

# transform to geo crs for leaflet mapping
geo_points <- my_points %>%
     st_transform(crs = 4326)

# create data for different groups
rural <- geo_points %>% filter(pop == "rural_pop")
urban <- geo_points %>% filter(pop == "urban_pop")
state_lines <- st_geometry(simp_sf)

leaflet() %>%
     addProviderTiles("CartoDB.Positron") %>%
     addResetMapButton() %>%
     addPolylines(
          data = state_lines
          color = "black",
          weight = 1,
          opacity = 1,
     ) %>%
     addCircleMarkers(
          data = rural
          color = "#F8766D",
          radius = 1,
          stroke = FALSE,
```

```
        fillOpacity = 0.5,
        group = "Rural"
    ) %>%
    addCircleMarkers(
        data = urban
        color = "#00BFC4",
        radius = 1,
        stroke = FALSE,
        fillOpacity = 0.5,
        group = "Urban"
    ) %>%
    addLayersControl(
        overlayGroups = c("Rural", "Urban", "Rural Population"),
        options = layersControlOptions(collapsed = FALSE)
    )
```

Click here to view the interactive map in a separate window.

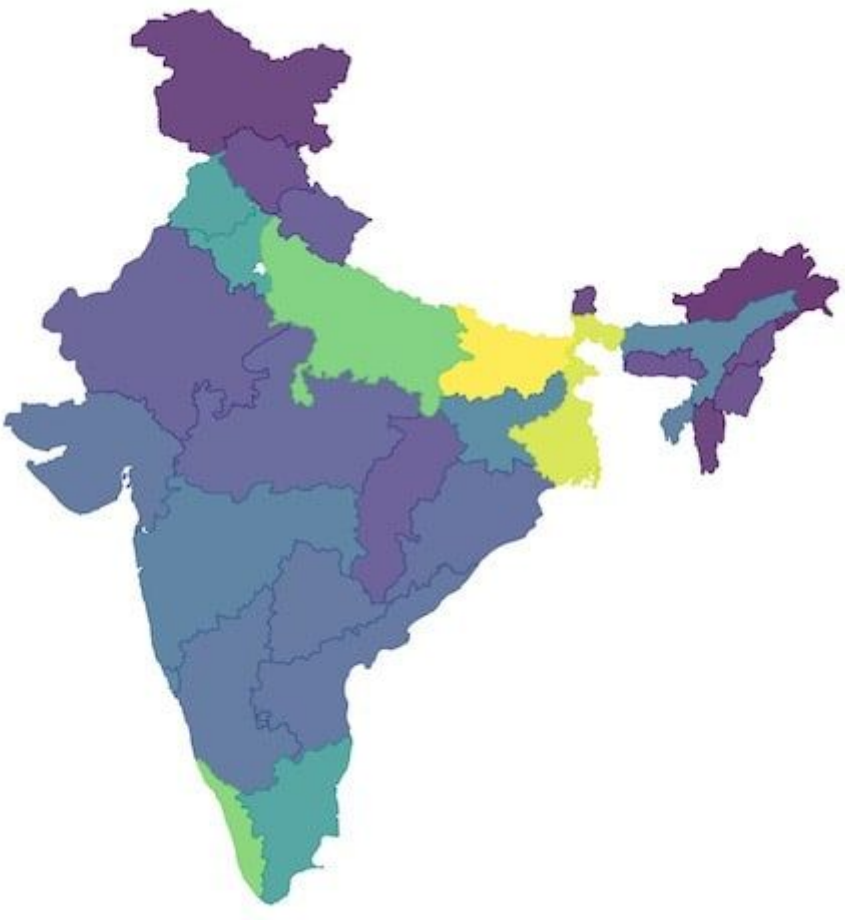
Interactivity with plotly

Like Leaflet, **`plotly`** is an R wrapper package for a JavaScript visualization API — in this case, it's the `plot.ly` JavaScript library. `plotly` is another robust and well-documented option for adding interactivity to any kind of plot.

> LEARN MORE:
>
> In addition to the official documentation, there are a number of excellent resources for plotting geospatial data with `plotly`.
>
> - There is now a cookbook-style handbook, which includes a dedicated section for maps.
> - This blog post explores improving `ggplotly()` conversions for maps.
> - This blog post covers visualizing geospatial data with `sf` and `plotly`.

Using `plotly` syntax, we can easily construct interactive choropleths similar to the one we created with `ggiraph`. Here is one for population density.

```
library(plotly)
plot_ly(states_sf,
        split = ~state_ut,
        color = ~density_km2,
        text = ~paste0(state_ut, ": ", density_km2),
        hoverinfo = "text",
        hoveron = "fill",
        alpha = 0.8,
        showlegend = FALSE) %>%
    layout(title = "Population Density across Indian States")
```



Click here to view the interactive map in a separate window.

Beyond simple choropleths, `plotly` really shines at more complex levels of interaction, implementing effects more commonly reserved for Shiny apps. Like Leaflet, `plotly` can also take advantage of customizable base map tiles via the `plot_mapbox()` function.

> NOTE:
>
> Using the `plot_mapbox()` function will require setting up a public access Mapbox token.

`plotly` can also be used for brushing and linking views. **Brushing** refers to subsetting data based on some kind of user input like a box selection. This user selection is then linked to a part of the visualization, which reacts to the selection.

For example, we can design visualizations with multiple views that interact with each other. The **`crosstalk`** package lets HTML widgets share data and "talk" to each other. Taking advantage of the `crosstalk` package, `plotly` is able to "link views". The `crosstalk::bscols()` function arranges HTML elements or widgets in Bootstrap columns. This allows for persistent or generalized selection across elements.

Below, after creating a `SharedData` object and specifying `region` as a key argument, we can make selections on a map that generate changes in a data table. This is still a fairly simple effect, but it demonstrates a kind of linking framework only possible with interactive maps.

Click anywhere on the map to see the table filter by region.

```
library(listviewer)
library(crosstalk)
library(DT)

simp_sd <- SharedData$new(
     simp_sf %>%
          select(state_ut, pop_2011, decadal_growth, region) %>%
          mutate(pop_2011 = format(simp_sf$pop_2011, big.mark = ",")),

     ~region)

bscols(
     plot_mapbox(
          simp_sd,
          text = ~state_ut, hoverinfo = "text"
          ) %>%
          layout(title = "Filter Table by Region via Point Selection"
),
     DT::datatable(
          simp_sd,
          rownames = FALSE,
          colnames = c('State/UT','Population','Pop. Growth','Regio
n','geometry'),
          options = list(
               autoWidth = FALSE,
               columnDefs = list(
                    list(width = '50px', className = 'dt-left', target
s = 0:4)
                    )
               )
     )
)
```

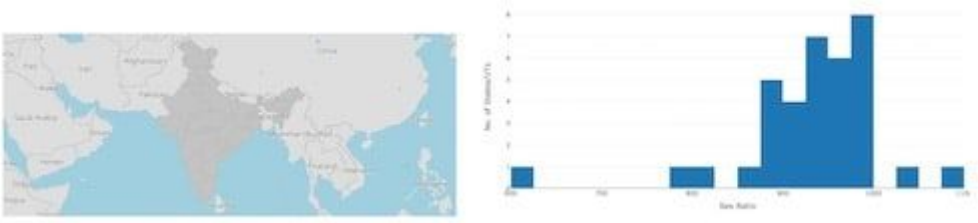
State/UT	Population	Pop. Growth	Region
Andhra Pradesh	49,386,799	0.111	Southern
Jammu & Kashmir	12,548,926	0.237	Northern
Himachal Pradesh	6,864,602	0.129	Northern
Punjab	27,704,236	0.137	Northern
Uttarakhand	10,116,752	0.192	Northern
Haryana	25,353,081	0.189	Northern
Chandigarh	1,055,450	0.171	Northern
Rajasthan	68,621,012	0.214	Western
Arunachal Pradesh	1,382,611	0.259	Northeastern
Delhi	16,787,941	0.21	Northern

Showing 1 to 10 of 36 entries

Click here to view the interactive map in a separate window.

We can even aggregate brush selections into a "**persistent selection**". For instance, we can examine the distribution of a variable like sex ratio in a histogram; then select points at the low or high end of the distribution, and watch the map highlight which states fall into the selected bin(s).

```
simp_sd <- SharedData$new(simp_sf)

bscols(
     plot_mapbox(
          simp_sd,
          split = ~state_ut,
          color = I("grey"),
          text = ~paste0(state_ut, ": ", sex_ratio),
          hoverinfo = "text",
          alpha = 0.8,
          showlegend = FALSE
          ) %>%
          highlight(dynamic = TRUE, persistent = TRUE),
     plot_ly(simp_sd, x = ~sex_ratio) %>%
          add_histogram(xbins = list(start = 600, end = 1100, size = 2
5)) %>%
          layout(
               barmode = "overlay",
               xaxis = list(title = "Sex Ratio"),
               yaxis = list(title = "No. of States/UTs")
          ) %>%
          highlight("plotly_selected", persistent = TRUE)
)
```



Click here to view the interactive map in a separate window.

There may be simpler ways to more effectively communicate this data, but it helped us get an idea of what is possible with `plotly`.

Mapping Applications in Shiny

These approaches for creating interactive maps can be quite powerful, but they do have their limitations. The interactive maps shown above allow some degree of user interactivity, but the code is ultimately static and so the user interface is largely fixed. While packages like `plotly` can be used for linking views, there are better ways to achieve high levels of interaction between maps and plots.

As shown in this gallery, there are now more than 100 registered HTML widgets for R. Communicating the full extent of their capabilities is often best done through a web application. In general, a complete application affords more possibilities that aren't possible with only the techniques above, such as downloading output after some kind of interaction.

Building a complete web application is the final step for interactivity. Within R, **Shiny** is the way to achieve this. Shiny is an R package that makes it easy to build interactive web apps straight from R without any knowledge of web development languages like HTML, CSS or JavaScript.

> LEARN MORE:
>
> The Shiny documentation is the best place to get started. It includes a number of sample apps, articles, and webinars. Shiny also has a gallery of sophisticated and simple apps to learn and seek inspiration from.

The Basics of Shiny Apps

Essentially, Shiny apps have two parts: a front end and a back end. When creating a Shiny app, you can choose to build it as a single file (in which case, the front end and back end are housed in two functions, `ui()` and `server()`) or two files (in which case, the front end and back end are found in separate files, `ui.R` and `server.R`).

`ui.R` controls the app's appearance, while **`server.R`** contains the logic that transforms a list of user inputs, such as dropdown menus or radio buttons, into various kinds of outputs, like plots or tables.

Beyond these minimum two files, larger projects often involve a few other important components. One is a separate data folder that holds all of the data read into the app. Another is a file, perhaps named `global.R`, that reads in data files, sets global variables, and contains functions to be used in `server.R`.

Particularly as complexity increases, it's helpful to pare down the `server.R` file to only the reactive logic of Shiny. Setting variables, reading in data, and functions describing how to build objects can all be handled in a `global.R` file. Removing these elements allows you to better focus on the reactivity in `server.R`.

Lastly, you might add a `styles.css` file for custom styling. I chose to add `includeCSS(styles.css)` inside the header tag within my `ui.R` file. This allowed us to override any of the app's default styling in a separate file without distracting from the structure of `ui.R`.

> LEARN MORE:
>
> After you get a handle on these concepts, it can be helpful to view the code for mapping applications in Shiny. Examples include those found in the Leaflet documentation, *Geocomputation with R*, tmap, and the Shiny gallery itself. The Shiny documentation also includes an example of an interactive choropleth. Finally, this blog post is also aimed at beginners starting to learn Shiny and Leaflet.

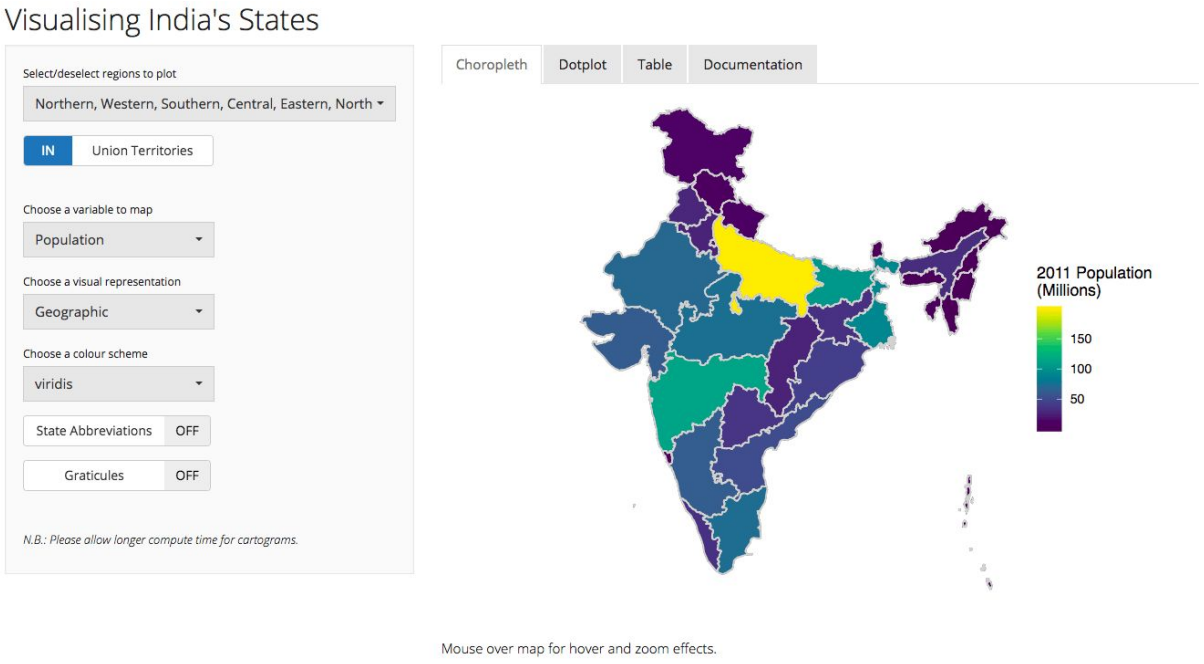
Example of a Geospatial Shiny App

Leaflet is the most robust option for mapping in Shiny, especially when the data is truly geospatial. However, because our data set focuses more on attributes than

geometry, we chose to explore different geospatial representations without the underlying base maps using a `ggiraph` object.

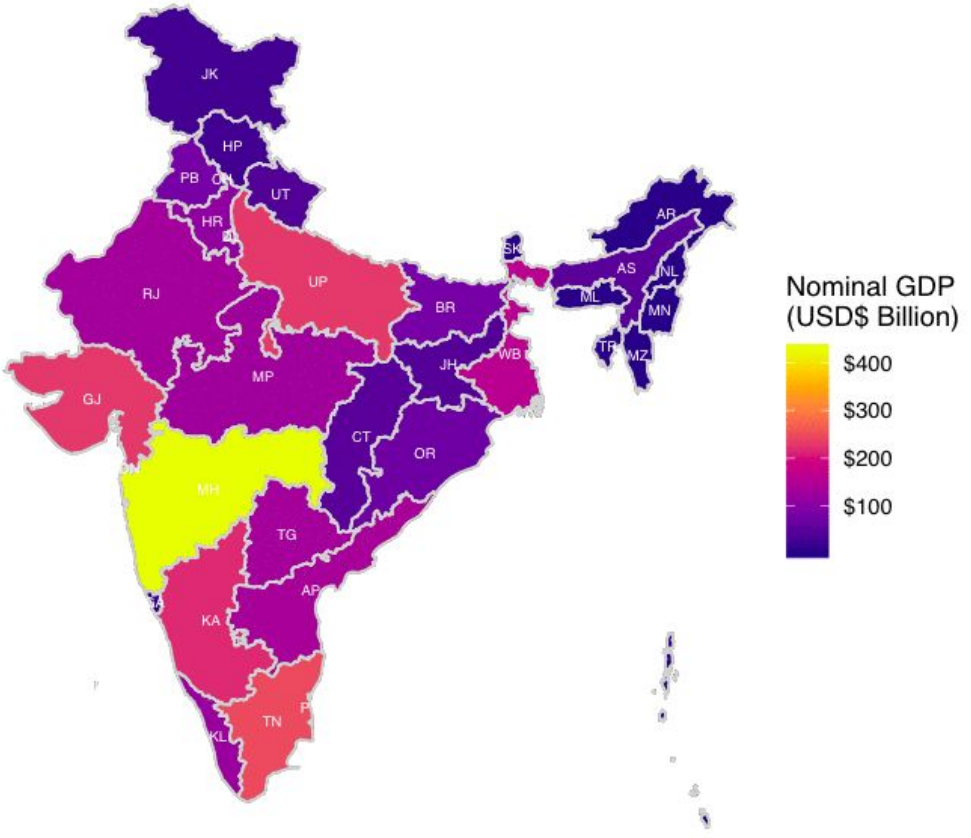
In the Shiny app below, you can construct a choropleth of any variable in the data set for any subset of India's regions. You can also compare how this choropleth changes across a number of geographic representations, such as cartograms and hexbin maps. Further, you can cross-check the data presented in the choropleth with its corresponding dotplot and table in the adjacent tabs.

You can try the app out for yourself here.



> LEARN MORE:
>
> Check out Sean Angiolillo's R User Meetup talk and blog about how he built a much more complex Shiny app — an interactive data visualization of the 1991-2011 Indian Census data depicting "Households Classified by Source and Location of Drinking Water and Availability of Electricity and Latrine".

The gif below also shows some of the visualizations the app can generate.



As far as what Shiny is capable of, this app is still quite simple, but hopefully it demonstrates some of Shiny's potential for communicating interesting geospatial data stories.

Final Thoughts

This lesson and the previous one together have only scratched the surface of R's mapping capabilities. R is well-known for its visualization libraries, and this reputation holds for geospatial data as well. Whether you're creating static, animated or interactive maps, there's an R package ready to help you create high-quality visualizations. Just remember to keep in mind your objectives and goals for any visualization before designing either a static, animated or interactive map.

View this lesson online

LESSON 5

Performing Spatial Subsetting in R

This lesson was written by Sean Angiolillo and was last updated on 29 Jan. 2019.

So far in this GIS course, we've introduced the following ideas:

- use cases for geospatial data
- getting started with the `sf` package
- manipulating attributes of geospatial data in a tidy workflow
- visualizing geospatial data through a wide array of static, interactive and animated maps

However, we've yet to really do anything useful with the actual geometry of our geospatial data. In this lesson, we'll introduce **spatial subsetting**, an important family of operations applicable to geospatial data.

What Is Spatial Subsetting?

We've seen how to manipulate `sf` spatial dataframes through their attributes, as we'd do with normal dataframes. For instance, with `dplyr`, we can filter a spatial dataframe to keep only observations that match certain factor levels or have a numeric variable above or below a certain threshold. Similarly, we can also filter observations using the geometry column of our geospatial data.

As defined in *Geocomputation with R*, “Spatial subsetting is the process of selecting features of a spatial object based on whether or not they in some way relate in space to another object.”

> LEARN MORE:
>
> Chapter 4, "Spatial data operations", of *Geocomputation with R* is the place to start learning about spatial subsetting in R.

To take an example using our previous data set of Indian states, we might wish to filter for only states that share a border with Delhi NCR. Or, rather than filtering by attributes like states or districts, we may only care about states or districts within a certain distance from a particular point. Spatial subsetting operations allow us to perform these kinds of manipulations.

Topological Relations

Many types of spatial subsetting operations are available at our fingertips. Different types of spatial relations are more formally called **topological relations**. The two examples given above describe different topological relations. The former is looking for a common border, or perhaps areas that "touch", whereas the latter is looking for areas "within" another area.

As implemented in the `sf` package, you'll find these operations in functions like `st_intersects()`, `st_disjoint()`, `st_within()`, `st_contains()`, `st_touches()`, `st_crosses()` and more. The documentation for any of these functions includes the complete list.

These functions require a pair of `sf` geometry sets — a target object and a selecting object. Before diving into the specific syntax, let's first get a sense of how these relations are defined.

The simplest is `st_intersects()` and its inverse `st_disjoint()`. Giving two `sf` objects to **`st_intersects()`** will return all observations that intersect with each other in any way. Conversely, **`st_disjoint()`** returns observations with no intersection.

Another pair of operators is **`st_within()`** and **`st_contains()`**. Both of these operations return only observations that lie entirely within one object or another. The designation of "x" and "y" arguments determines whether `st_within()` or `st_contains()` is actually the operation you need. `st_within()` returns observations in "x" that fall entirely within "y". `st_contains()` returns observations in "x" that entirely contain "y".

> LEARN MORE:
>
> These relationships are much easier to understand with diagrams. GITTA has a useful introduction to topological relations, complete with Venn diagrams of each. More diagrams can also be found on Wikipedia. Another helpful resource is S Ogletree's article, which uses toy data within R to look at the differences between relations.

Preparing Data

Before diving in to the syntax of spatial subsetting, we need some sample data. We'll use the **`tidycensus`** and **`tigris`** packages to download median household income data for the Philadelphia metro area at the census tract level.

> NOTE:
>
> In the US Census hierarchy, census tracts are below counties and above block groups. See Kyle Walker's `tigris` slides for more information.

> LEARN MORE:
>
> The documentation is a good way to get started with Kyle Walker's `tidycensus` and `tigris` packages. Note that you'll need to get an API key from the Census Bureau.

```
library(tigris)
library(sf)
library(tidycensus)
library(tidyverse)
library(rvest)
options(tigris_class = "sf")
options(tigris_use_cache = TRUE)

api_key <- "YOUR_API_KEY"
census_api_key(api_key)
```

The `tigris` package does have a `core_based_statistical_areas()` function for downloading shapefiles of metro areas, but instead we'll start with a table of counties covering our area of interest. Then we'll demonstrate how to get a more narrow geographic area through spatial subsetting.

```
# create df of states and counties
counties <- tribble(
     ~state, ~county,
     "PA", "Philadelphia",
     "PA", "Montgomery",
     "PA", "Bucks",
     "PA", "Delaware",
     "NJ", "Burlington",
     "NJ", "Camden",
     "NJ", "Gloucester"
)

# query tidycensus and combine data into one sf object
raw_tracts <- map2(counties$state, counties$county, function(x, y) {
     get_acs(geography = "tract", state = x, county = y,
             variables = c(hhincome = "B19013_001"),
             geometry = TRUE)
}) %>%
     do.call(rbind, .)

glimpse(raw_tracts)
```

```
## Observations: 1,186
## Variables: 6
## $ GEOID    <chr> "42101000100", "42101000200", "42101000300", "4210
10004…
## $ NAME     <chr> "Census Tract 1, Philadelphia County, Pennsylvani
a", "C…
## $ variable <chr> "hhincome", "hhincome", "hhincome", "hhincome", "h
hinco…
## $ estimate <dbl> 103772, 50455, 93036, 57604, 70038, 40568, 71250,
51279…
## $ moe      <dbl> 11761, 22642, 15233, 10786, 14655, 13811, 17493, 5
359, …
## $ geometry <MULTIPOLYGON [°]> MULTIPOLYGON (((-75.15154 3..., MULTIP
OLYG…
```

We now have 1,186 census tracts covering the Philadelphia metropolitan area. This is a larger area than we want to cover so we'll spatially subset this data set to a

smaller area based on distance from a central point of interest, in this case Philadelphia's City Hall.

Before we can do this, however, it's important to pay attention to the coordinate reference system (CRS) of our geospatial data. The commands below show that the data has a geographic CRS with EPSG code 4269.

```
st_crs(raw_tracts)
```

```
## Coordinate Reference System:
##   EPSG: 4269
##   proj4string: "+proj=longlat +datum=NAD83 +no_defs"
```

```
st_is_longlat(raw_tracts)
```

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

In order to use spatial subsetting operations, we need to reproject our data from a geographic CRS to a projected CRS. In this case, we've chosen to use EPSG code 2272.

```
proj_crs <- 2272
proj_tracts <- raw_tracts %>%
     st_transform(crs = proj_crs)
st_crs(proj_tracts)
```

```
## Coordinate Reference System:
##   EPSG: 2272
##   proj4string: "+proj=lcc +lat_1=40.96666666666667 +lat_2=39.933333
33333333 +lat_0=39.33333333333334 +lon_0=-77.75 +x_0=600000 +y_0=0 +el
lps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=us-ft +no_defs"
```

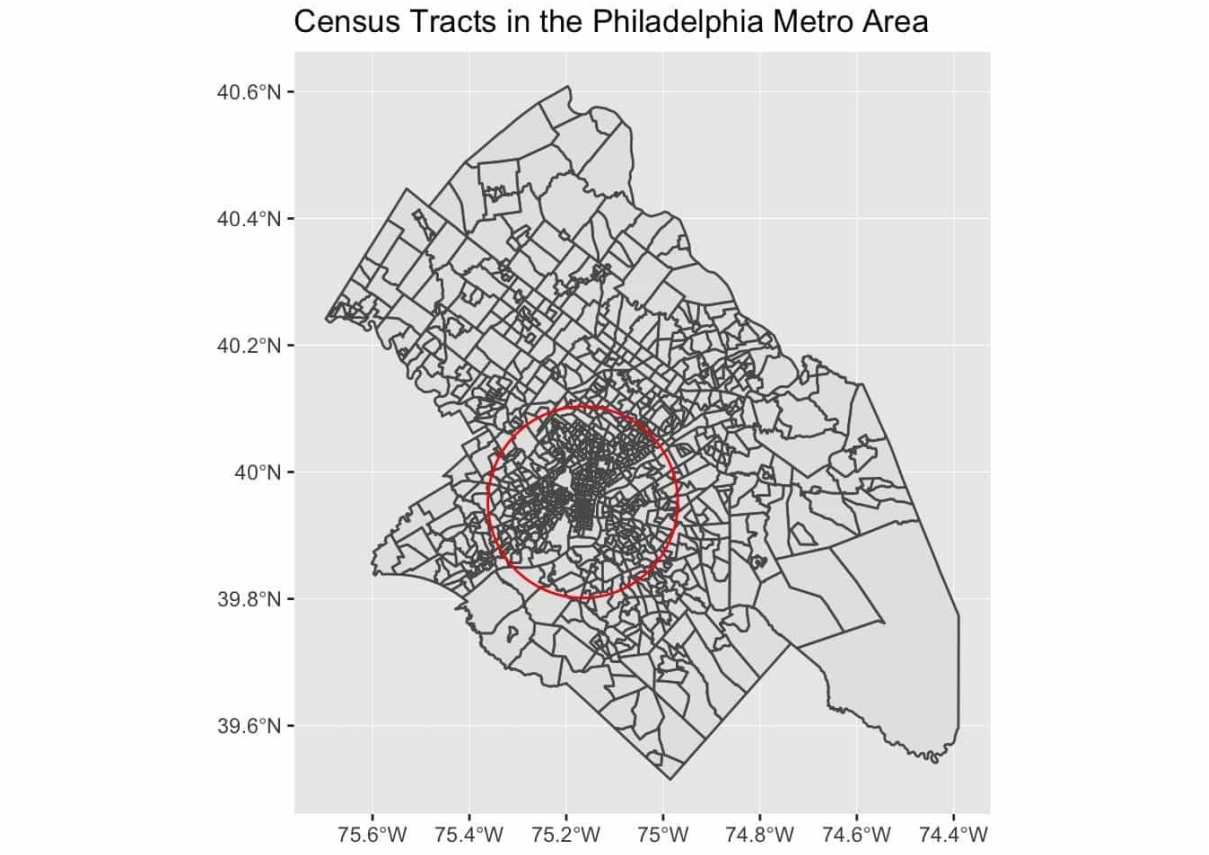
> LEARN MORE:
>
> For more information on coordinate reference systems, see Chapter 2 and Chapter 6 of *Geocomputation with R*.

Now that we have projected census tracts, we'll define a circle and use it as the second geometry feature set by which we'll subset the census tracts.

```
# choose a central long-lat point and radius to define circle
city_hall_lng <- -75.1657936
city_hall_lat <- 39.952383
geo_crs <- 4326
buffer <- 55000 # ft (same units as crs)

circle <- st_sfc(st_point(c(city_hall_lng, city_hall_lat)),
                 crs = geo_crs) %>%
     st_transform(crs = proj_crs) %>%
     st_buffer(dist = buffer)

st_crs(circle)
```

```
## Coordinate Reference System:
##   EPSG: 2272
##   proj4string: "+proj=lcc +lat_1=40.96666666666667 +lat_2=39.933333
33333333 +lat_0=39.33333333333334 +lon_0=-77.75 +x_0=600000 +y_0=0 +el
lps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=us-ft +no_defs"
```

As shown in the map below, with these two simple feature geometry sets in the same projected CRS, we are ready to spatially subset.



Spatial Subsetting Syntax

In R, there are often multiple ways to achieve the same result. For subsetting, we have a base R method using the square bracket `[` and a tidyverse method using `filter()`. Spatial subsetting is no exception — both options are available within the `sf` package.

The syntax is remarkably simple with the **square bracket method**. It's very similar to bracket subsetting of a dataframe. But inside the square bracket, where a logical expression would filter rows, you just need to place the selecting simple feature geometry (i.e. a spatial dataframe, `sfc_POLYGON`, etc).

In the example below, we subset the original 1,186 census tracts by those that intersect the circle we defined. The result is a new `sf` spatial dataframe with 617 observations.

```
philly <- proj_tracts[circle,]
glimpse(philly)
```

```
## Observations: 617
## Variables: 6
## $ GEOID    <chr> "42101000100", "42101000200", "42101000300", "4210
10004…
## $ NAME     <chr> "Census Tract 1, Philadelphia County, Pennsylvani
a", "C…
## $ variable <chr> "hhincome", "hhincome", "hhincome", "hhincome", "h
hinco…
## $ estimate <dbl> 103772, 50455, 93036, 57604, 70038, 40568, 71250,
51279…
## $ moe      <chr> 11761, 22642, 15233, 10786, 14655, 13811, 17493, 5
359, …
## $ geometry <MULTIPOLYGON [US_survey_foot]> MULTIPOLYGON (((2696885
236.…
```

By default, `st_intersects()` is the unspoken topological operator when using the square bracket for spatial subsetting. Setting the "op" argument allows us to choose any topological relation instead of the default `st_intersects`. In the example below, we've chosen `st_disjoint()`.

> NOTE:
>
> The 617 observations returned from the intersection plus the 569 observations returned from `st_disjoint()` sum to the original 1,186 tracts.

```
philly_dj <- proj_tracts[circle, , op = st_disjoint]
nrow(philly_dj)
```

```
## [1] 569
```

A second method of spatial subsetting involves creating an intermediary object of the class "**sparse geometry binary predicate**" (sgbp), which is essentially a list of matching indices we can use to subset the target object. Under this method, rather than setting an "op" argument, we use a different topological operator beginning with `st_*`.

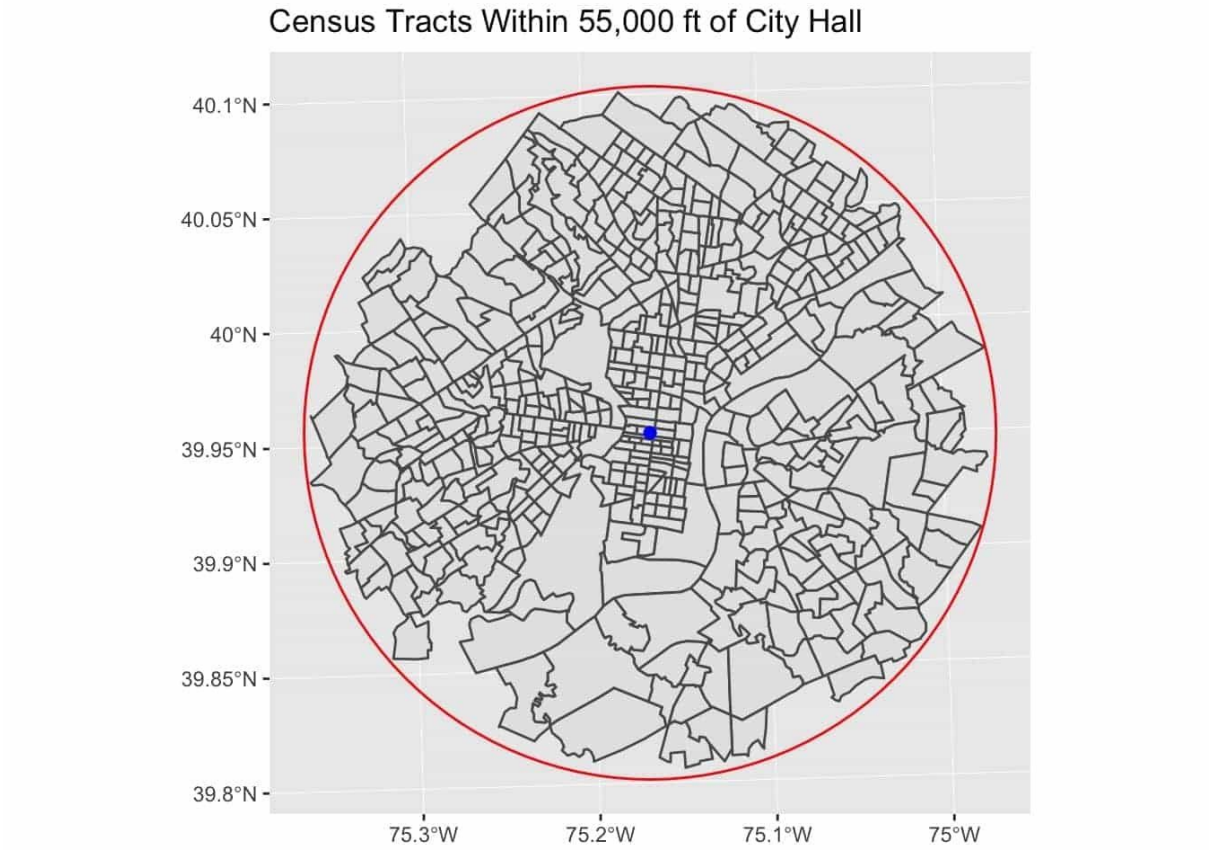
Moreover, we have the option of returning a **sparse** or a **dense matrix**, which slightly affects the syntax as shown below. This method fits more easily into a tidy workflow, as evidenced by the use of `dplyr`.

```
# sgbp, sparse matrix
philly_sparse <- proj_tracts %>%
     filter(lengths(st_intersects(x = ., y = circle)) > 0)

# sgbp, dense matrix
philly_dense <- proj_tracts %>%
     filter(st_intersects(x = ., y = circle, sparse = FALSE))
```

Regardless of which method you choose, all three methods should return the same number of observations in a spatial dataframe of the same CRS.

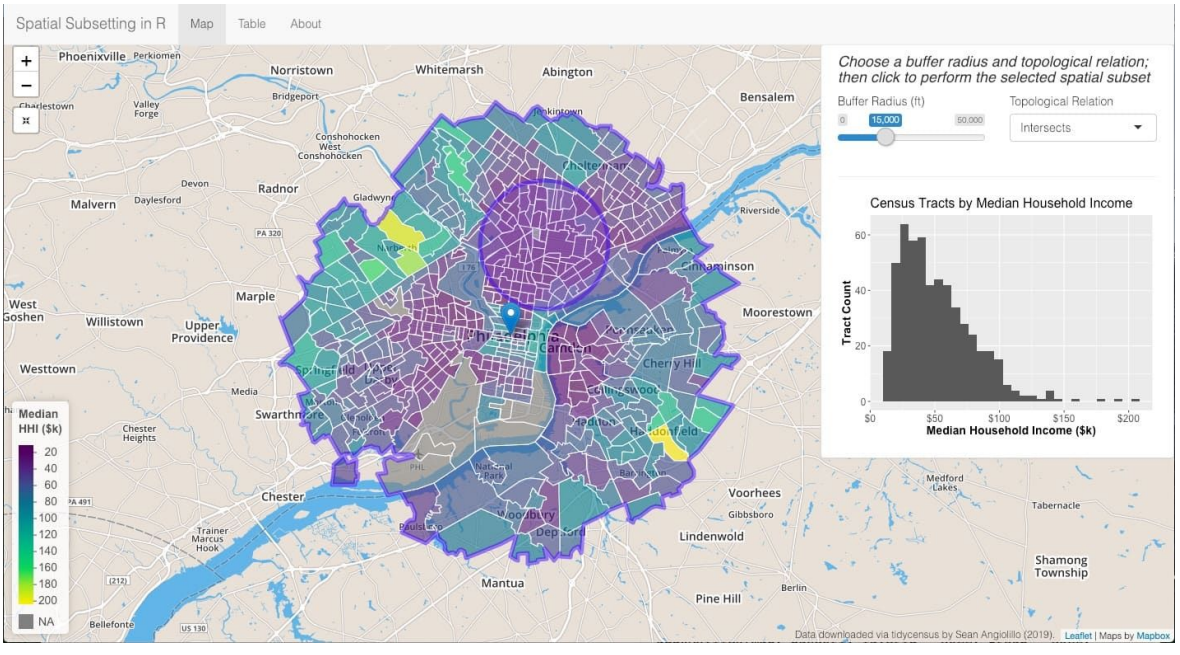
To give one example of this tidy workflow, note below how we can start with our original spatial dataframe, perform a spatial subset (in this case `st_within`), and directly pipe the result into `ggplot2`. As we'd expect, the result is a much smaller and more circular shape, fitting just inside the boundaries of our circle.

```
city_hall <- st_sfc(
     st_point(c(city_hall_lng, city_hall_lat)),
     crs = geo_crs) %>%
     st_transform(crs = proj_crs)

proj_tracts %>%
     filter(st_within(x = ., y = circle, sparse = FALSE)) %>%
     ggplot() +
     geom_sf() +
     geom_sf(data = circle, color = "red", fill = NA) +
     geom_sf(data = city_hall, color = "blue", size = 2) +
     labs(title = "Census Tracts Within 55,000 ft of City Hall")
```



Explore Spatial Subsetting via Shiny

While seeing the syntax and a few diagrams can be useful, it's even better to practice with different operations and quickly see the results. **This Shiny app** will let you quickly explore spatial subsetting through different topological relations.



It shows the same map of census tracts for the Philadelphia metropolitan area. After choosing a topological relation, you can position a circle of any size over the map of census tracts and click to perform a spatial subset.

After a click, you'll see several results:

- The syntax of the given spatial subset, in both square bracket and dense matrix methods, appears in the right-hand panel.
- The histogram for the selection is plotted against the original distribution of census tracts.
- The choropleth color scale and legend adjusts to the selection's domain. This can be used to reveal more detailed variation within a region. For example, income levels vary widely between wealthier suburbs and core urban areas in the Philadelphia metro region. Spatially subsetting a smaller, more homogenous geographic area can show new patterns.

Final Thoughts

With this lesson and the Shiny apps as tools, you've hopefully learned:

- the concept of spatial subsetting and when it may be useful
- differences in topological relations
- multiple ways to spatially subset your own data

Hopefully, now you are well on your way to becoming as comfortable spatially subsetting your data as if it were simply attribute data.

Keep reading for one final lesson on how to explore satellite images, one of the best forms of geospatial data today!

View this lesson online

LESSON 6

Exploring Raster Images in R

This lesson was written by Himanshu Sikaria and was last updated on 29 Jan. 2019.

In the previous lessons, we talked about how to handle basic geospatial data — any data with a geographic component. Our previous examples used data sets with several economic indicators for each Indian state.

However, there's a more complex form of geospatial data — raster images, which is data captured by satellites orbiting the Earth. Raster images can be far more difficult to find and process, but their high level of detail and frequent updates make them incredibly valuable for analysis.

This lesson focuses on the basics of raster images — what they are, where to get them, how to extract and process them, and what basic operations and analysis you can do on them. We'll illustrate all of this by examining satellite data for a rural region of Karnataka, a state in south India.

The Basics of Rasters

Imagine if you had the power to click images of any location on Earth from space. Today, thanks to satellites, we do.

A **raster file** is an image of the Earth, which is geotagged. (That means that we can find the exact location of any of raster image on the world map.) Like any other image, raster images are made up of **cells** (pixels), and each cell has a value associated with it.

> LEARN MORE:
>
> Looking for more information about raster images? This blog by ArcGIS explains raster images perfectly, and these flashcards explain the jargon of raster imagery.

Raster Attributes

Every raster scene has various attributes, or parameters. These can be accessed by `@` — for example, `rastername@extent`.

Here are some attributes you should know:

1. **class**: There are three options — RasterLayer, RasterStack or RasterBrick. A **RasterLayer** object represents single-layer (variable) raster data. A **RasterStack** is a collection of RasterLayer objects with the same spatial extent and resolution. A **RasterBrick** is truly a multilayered object, and processing a RasterBrick can be more efficient than processing a RasterStack.

2. **resolution**: The size of each cell (or pixel) that makes up the entire image. This value is in degrees for the example below (1 degree ~ 110 kms).

3. **extent**: The latitude and longitude of the image's top right point and bottom left point.

4. **coord. ref.**: This is the current raster file's projection. The Earth is a sphere, and it needs to be projected to be converted to 2D.

5. **values**: The minimum and maximum values among all the cells in the raster.

> LEARN MORE:
>
> Learn more about geographic projections with this video from Vox.

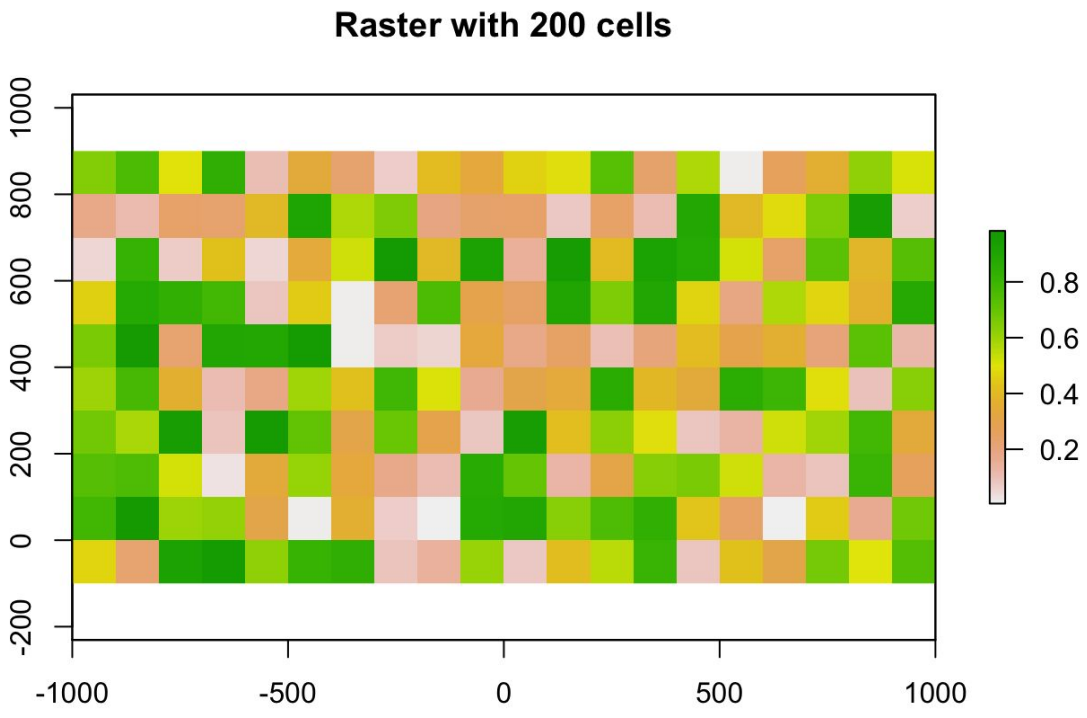
The "Hello World" of Rasters

```
library(raster)
```

```
# creating with default parameters
first_raster <- raster()
first_raster
```

```
## class       : RasterLayer
## dimensions  : 180, 360, 64800 (nrow, ncol, ncell)
## resolution  : 1, 1 (x, y)
## extent      : -180, 180, -90, 90 (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,
0
```

```
# creating with other parameters
r <- raster(ncol=20, nrow=10, xmn=-1000, xmx=1000, ymn=-100, ymx=900)
values(r) <- runif(ncell(r))

# Plotting the raster
plot(r, main = "Raster with 200 cells")
```



Exploring Karnataka and the Impact of Droughts

Let's get started with raster data in R by exploring Landsat 8 data for Karnataka, a state in south India.

What Is Landsat 8?

Landsat is without a doubt one of the best sources of free satellite data today. Managed by NASA and the United States Geological Survey, the Landsat satellites have been capturing multi-spectral imagery for over 40 years.

The latest satellite, **Landsat 8**, orbits the Earth every 16 days and captures more than 700 satellite images per day across 9 spectral bands and 2 thermal bands. Its imagery has been used for everything from finding drought-prone areas and monitoring coastal erosion to analyzing an area's fire probability and setting the best routes for electricity lines.

Landsat 8 Operational Land Imager (OLI) and Thermal Infrared Sensor (TIRS) images consist of nine spectral bands. Bands 1 to 7 and 9 have a spatial resolution of 30 meters, Band 8 (panchromatic) is 15 meters, and Bands 10 and 11 are 100 meters. The ultra blue **Band 1** is useful for coastal and aerosol studies. **Band 9** is useful for cirrus cloud detection. Thermal **Bands 10 and 11** are useful for providing more accurate surface temperatures.

	Bands	Wavelength (micrometers)	Resolution (meters)
Landsat 8 Operational Land Imager (OLI) and Thermal Infrared Sensor (TIRS) **Launched February 11, 2013**	Band 1 - Coastal aerosol	0.43 - 0.45	30
	Band 2 - Blue	0.45 - 0.51	30
	Band 3 - Green	0.53 - 0.59	30
	Band 4 - Red	0.64 - 0.67	30
	Band 5 - Near Infrared (NIR)	0.85 - 0.88	30
	Band 6 - SWIR 1	1.57 - 1.65	30
	Band 7 - SWIR 2	2.11 - 2.29	30
	Band 8 - Panchromatic	0.50 - 0.68	15
	Band 9 - Cirrus	1.36 - 1.38	30
	Band 10 - Thermal Infrared (TIRS) 1	10.60 - 11.19	100
	Band 11 - Thermal Infrared (TIRS) 2	11.50 - 12.51	100

Table from USGS

USGS gives free, public access to both its raw and processed satellite images. Raw images are available on AWS S3 and Google Cloud Storage, where they can be downloaded immediately. Processed images are available with the EROS Science Processing Architecture (ESPA). Images are also available through a variety of data products, such as SR (Surface Reflectance), TOA (Top of Atmosphere) and BR (Brightness Temperature).

Accessing the processed Landsat 8 data can be tricky. There are two different APIs — one by Development Seed for searching (called *sat-api*) and one by USGS for downloading (called *espa-api*). Download requests have to include the product ID, projection, and format of the data, then they must be approved by USGS, which can take anywhere from a couple minutes to a couple days. To make matters worse, the APIs input and output data with different structures.

> LEARN MORE:
>
> New to Landsat 8 data? Here's lots more information to get you started:
>
> - Read about the Landsat Collection (Pre-Collection and Collection 1) here.
> - Watch this video to understand the difference between the data on ESPA and AWS S3/Google Cloud Storage, and why using ESPA is preferred over AWS's Digital Numbers (DN).
> - Watch a video on how Landsat data is captured here.
> - Read about over 120 applications of Landsat 8 data here.

Downloading Raster Images from the Landsat Satellite

Our open-source package **rLandsat** can be used to easily download the latest Landsat raster images — no Python or API knowledge needed! (You can read more about `rLandsat` here.)

Landsat divides the entire earth into grids, each with a unique row and path. This tool from USGS can be used to convert a latitude and longitude to a path and row.

A part of Karnataka that had a major drought lies in path 145 and row 49. Let's download the latest imagery for this grid using `rLandsat` functions.

```
# load the library
library(rLandsat)

# input the credentials. Can be obtained from https://ers.cr.usgs.gov/
register
espa_creds("username", "password")

# search for the available scenes for the specified data and row/path
result = landsat_search(min_date = "2016-01-01", max_date = Sys.Date
(), path_master = 145,
         row_master = 49)

# Placing an order to download the raster files for the tiles, one for
each year
product_id = c("LC08_L1TP_145049_20180301_20180308_01_T1", "LC08_L1TP_
145049_20170330_20170414_01_
               T1", "LC08_L1TP_145049_20140407_20170424_01_T1")
result_oreder = espa_order(product_id, product = "sr")
order_id = result_oreder$order_details$orderid
download_url = espa_status(order_id)

# Downloading the zipped files once the order is processed
landsat_download(download_url$order_details$product_dload_url, dest_fi
le = "karnataka_landsat/")
```

> NOTE:
>
> To run any of the functions starting with `espa_`, you need valid login credentials from ESPA-LSRD, and you need to input them in your environment with `espa_creds(username, password)` (as above) for the functions to work properly.

Reading the Files in R

Once the download is complete and the TAR files extracted, there will be a GeoTIFF file for each Landsat band. Each band represents the light reflected at different frequencies. Bands 2, 3 and 4 represent the light visible to the human eye.

The next step is to load the rasters to R using the **`raster`** library. The file size of rasters are generally huge (one Landsat tile has about 60 million pixels), so the `raster` library doesn't load the entire data to memory; only when required, the functions call the values and processes them in chunks. As a result, the `raster` library saves every intermediate variable in the temp folder.

> LEARN MORE:
>
> Read detailed information about the `raster` library in its CRAN documentation.

First, let's try to load the data for different bands in R using the raster library, and then we'll plot any one of them. To read a single raster image, we can use the `raster()` function. To read a stack (multiple) of rasters at once, we can use the `stack()` function. Printing the raster/stack file will give brief information about the raster.

```
# listing all the files for the 3 years and saving them in stacks
karnataka_2018_files <- list.files("LC08145049201803010 1T1-SC20180920033522/", pattern =
                                   "*band.*tif")
karnataka_2018 <- stack(karnataka_2018_files)
karnataka_2017_files <- list.files("LC08145049201703300 1T1-SC20180920035626/", pattern =
                                   "*band.*tif")
karnataka_2017 <- stack(karnataka_2017_files)
karnataka_2014_files <- list.files("LC08145049201404070 1T1-SC20180920031728/", pattern =
                                   "*band.*tif")
karnataka_2014 <- stack(karnataka_2014_files)
```

```
# printing the summary of a raster
karnataka_2018[[1]]
```

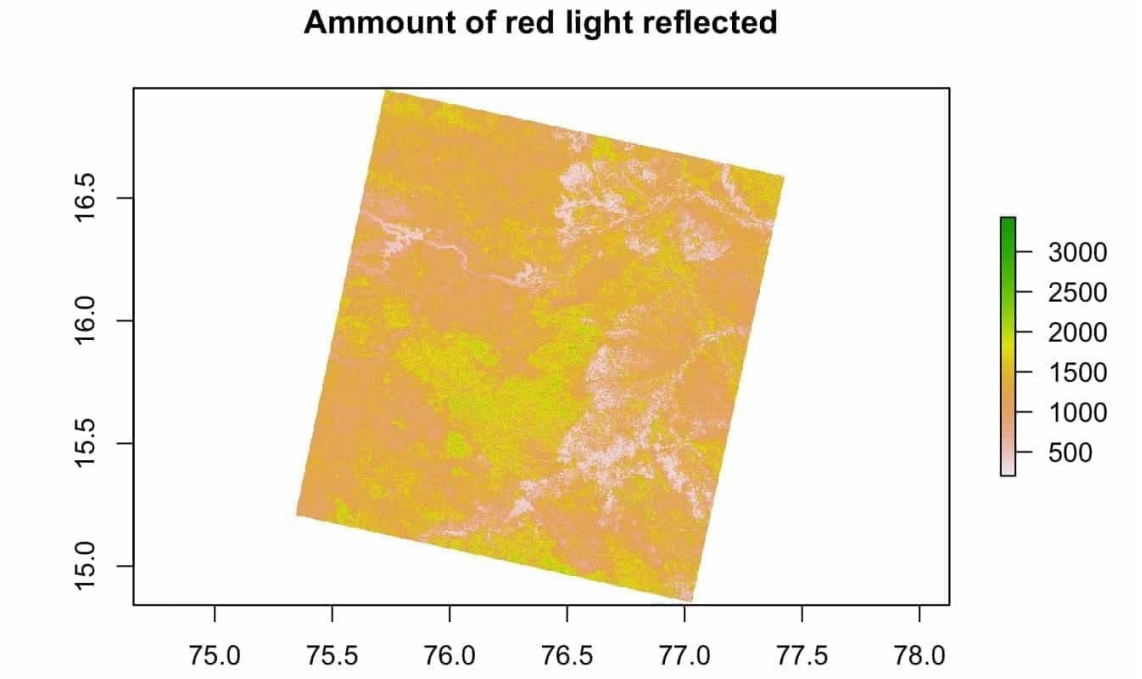
```
## class       : RasterLayer
## dimensions  : 7817, 7914, 61863738 (nrow, ncol, ncell)
## resolution  : 0.0002695, 0.0002695 (x, y)
## extent      : 75.32427, 77.45709, 14.84117, 16.94785 (xmin, xmax, y
min, ymax)
```

```
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +tow
gs84=0,0,0
## data source : /Users/himanshu/Downloads/LC08145049/LC08145049201803
0101T1-SC20180920033522/LC08_L1TP_145049_20180301_20180308_01_T1_sr_ba
nd1.tif
## names       : LC08_L1TP_145049_20180301_20180308_01_T1_sr_band1
## values      : -32768, 32767 (min, max)
```

Plotting a Raster Image

Plotting a single band's raster is simple — just use the `plot` function. We can also modify the color range of the plot using the `col` parameter.

```
# we can plot the Landsat8 band using the plot function
plot(karnataka_2018[[4]], main = "Ammount of red light reflected")
```

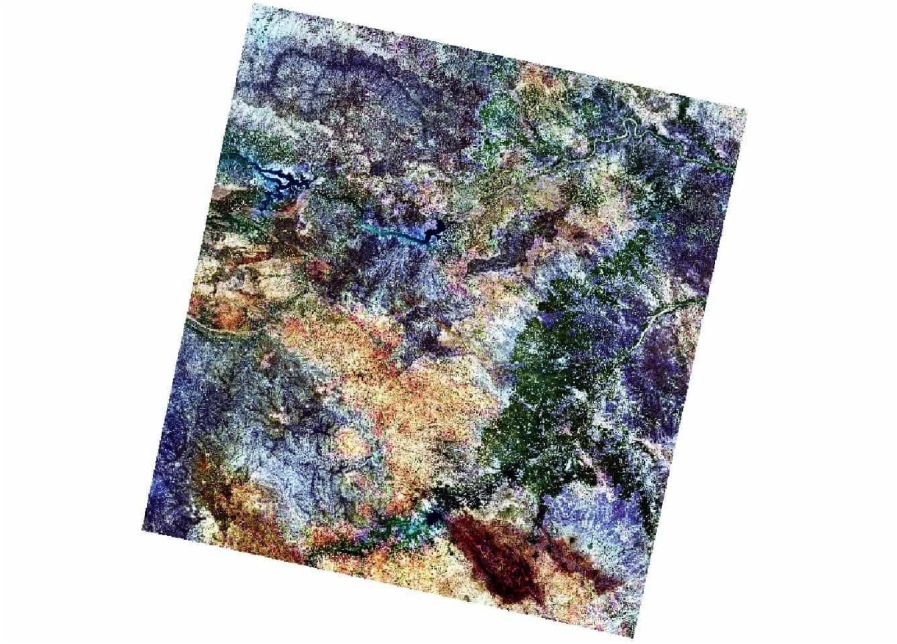


Plotting a Stack of Raster Images

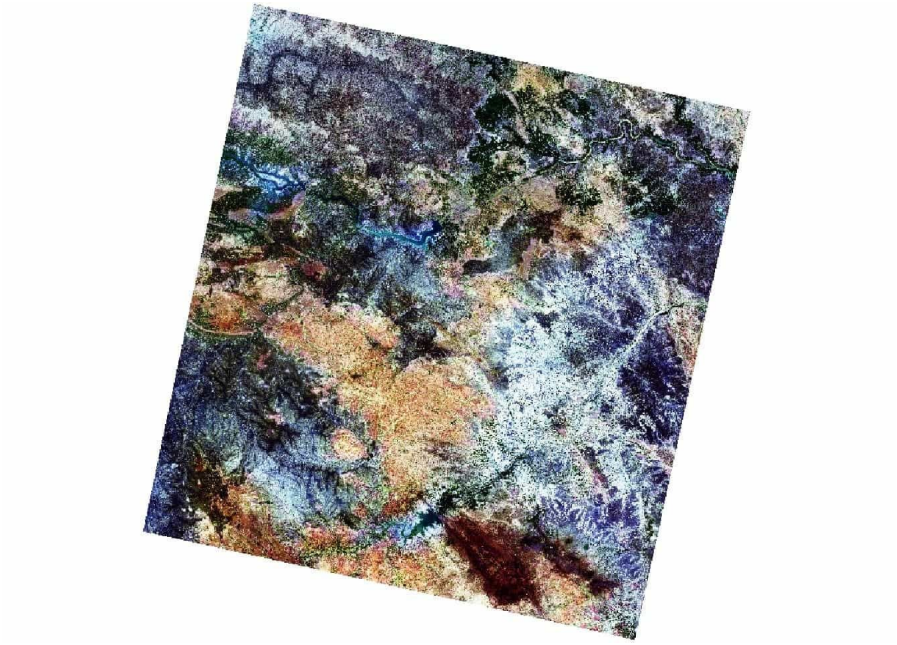
A **RasterLayer** is a single band raster file, which means that each pixel in the raster has a single value attached to it. We can create a RasterStack by combining different RasterLayers. This would assign multiple values to a single pixel.

Plotting a RasterStack with the visible bands gives an image of exactly how the human eye would see this piece of Earth from space.

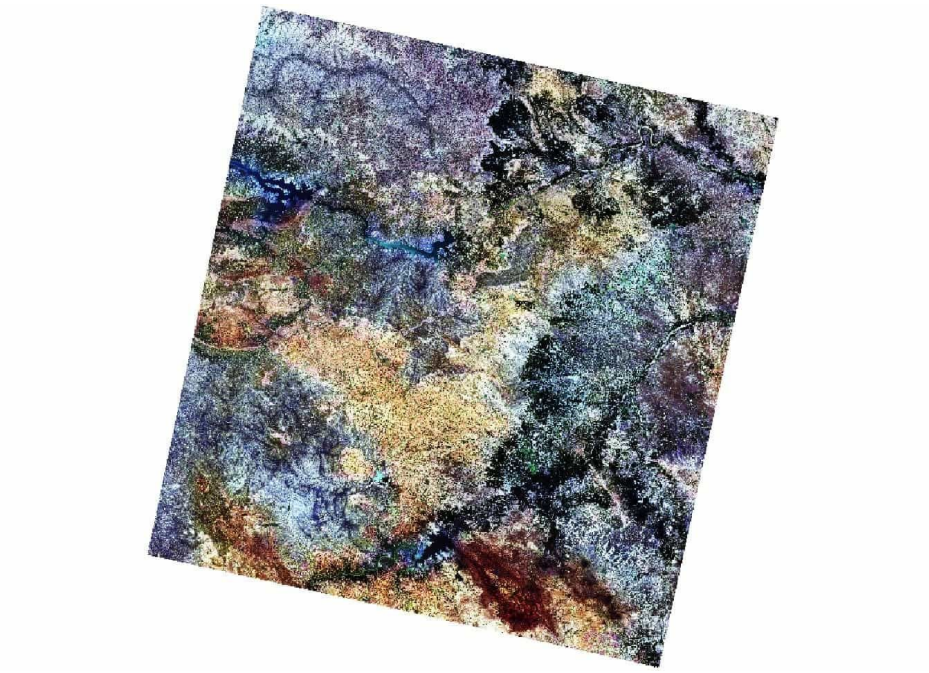
```
# we can also mention the bands as parameters in the function
plotRGB(karnataka_2014,4,3,2, stretch='hist')
```



```
plotRGB(karnataka_2017,4,3,2, stretch='hist')
```



```
rgb_stack = karnataka_2018[[c(4,3,2)]]
plotRGB(rgb_stack, stretch='hist')
```



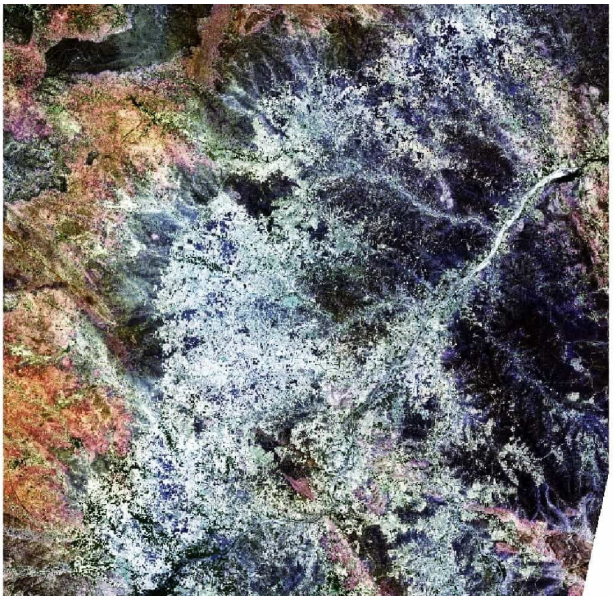
Cropping Raster Images

As you can see, this Landsat scene covers a lot of area, and it's difficult to actually get insights from it. Let's try to crop the scene to only the east region for more clarity.

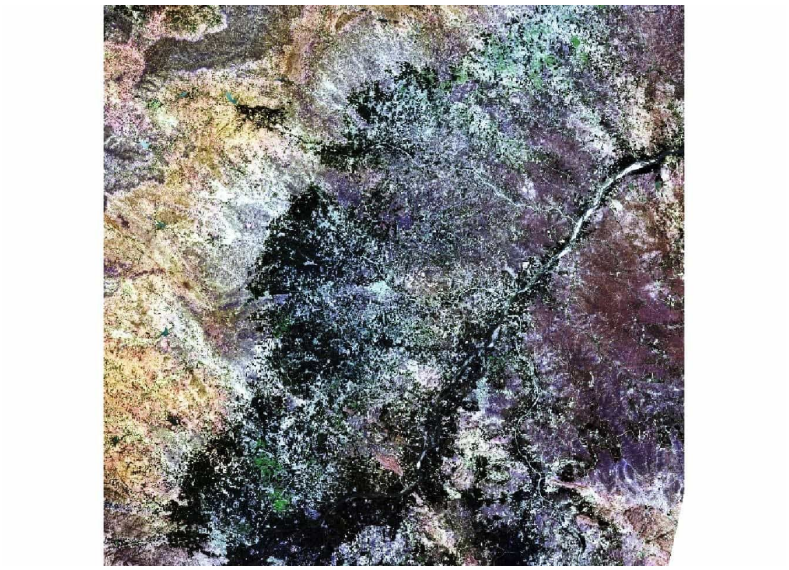
The upper-left and bottom-right coordinates are specified in the `extent()` function. The RasterStack is passed to the `crop()` function with the specified extent to be cropped.

```
east_extent <- extent(76.371,77.176,15.398,16.145)
karnataka_2017_east = karnataka_2017 %>% crop(east_extent)
karnataka_2014_east = karnataka_2014 %>% crop(east_extent)
karnataka_2018_east = karnataka_2018 %>% crop(east_extent)

plotRGB(karnataka_2014_east, 4,3,2, stretch='hist')
```

```
plotRGB(karnataka_2017_east, 4,3,2, stretch='hist')
```



```
plotRGB(karnataka_2018_east, 4,3,2, stretch='hist')
```



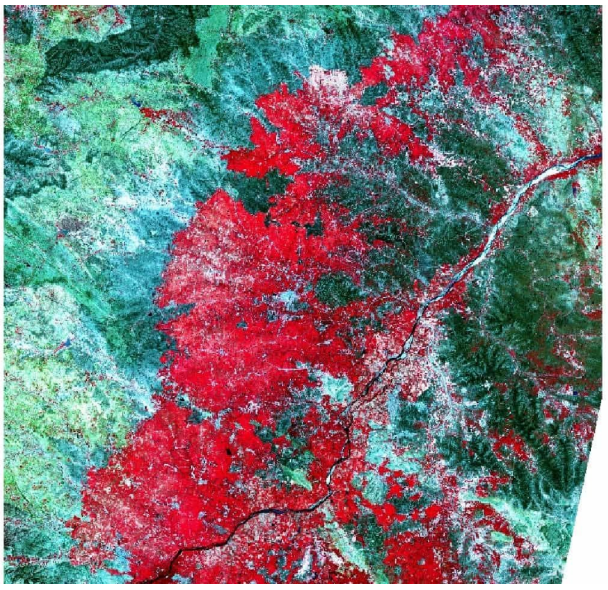
In these images, we can clearly see the change in greenery over the years — 2014 is the most green, and 2017 is the least. In fact, in 2017, Karnataka faced the worst drought in 42 years.

In the next section, let's see how we can quantify this change and check which regions were worst affected by the drought.

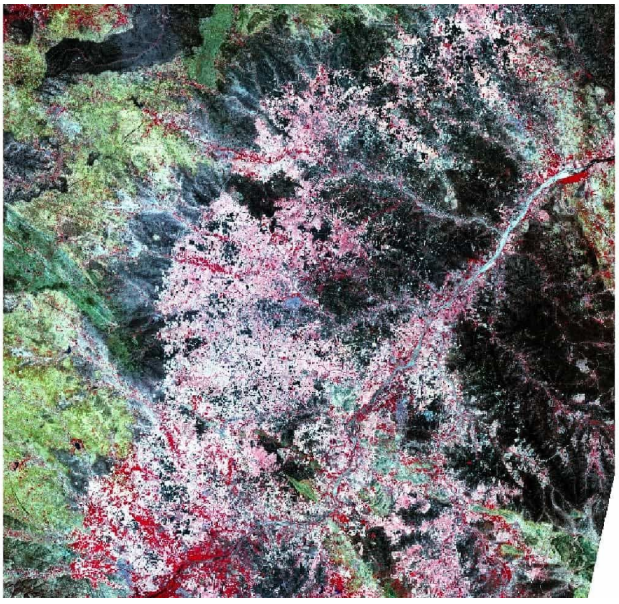
Vegetation Quality In and Around the Region

A combination of other bands can be super helpful too. If we combine infrared, red and green (Bands 5, 4 and 3), we can create a plot where the vegetation is red. The more vibrant the color, the healthier the crop.

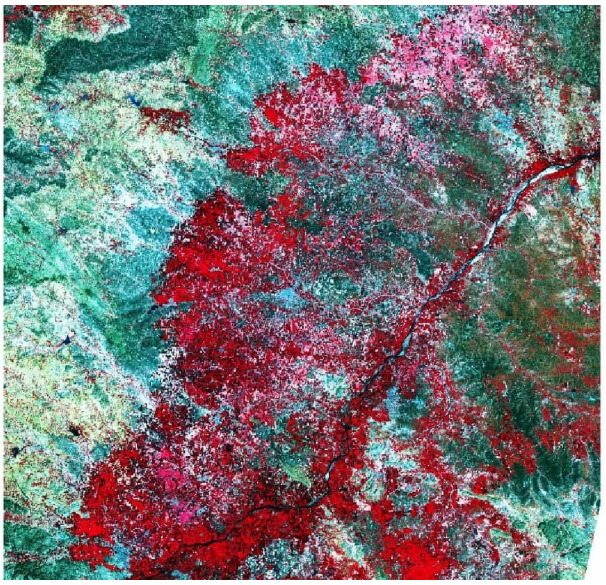
```
plotRGB(karnataka_2014_east, 5,4,3, stretch='hist')
```



```
plotRGB(karnataka_2017_east, 5,4,3, stretch='hist')
```



```
plotRGB(karnataka_2018_east, 5,4,3, stretch='hist')
```



Basic Operations

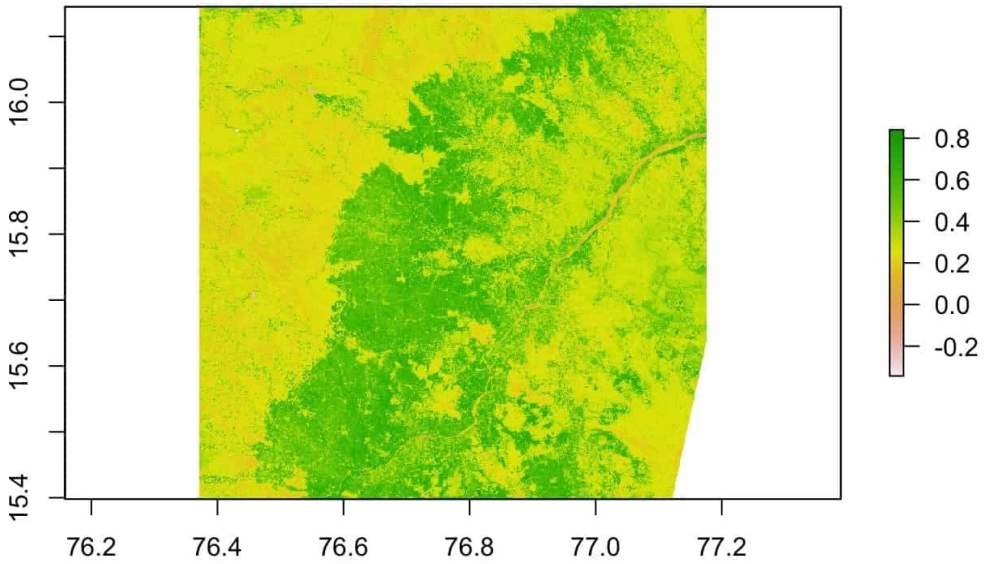
Doing raster operations is easy — most of the time, it can be treated as a numeric vector. By doing basic algebraic operations on different bands, we can create indices that better explain the characteristics of the region.

Let's try to create one of the most extensively used indices from Landsat — **NDVI** (Normalized Difference Vegetation Index). NDVI is defined as `(Band 5 - Band 4)/(Band 5 + Band 4)`.

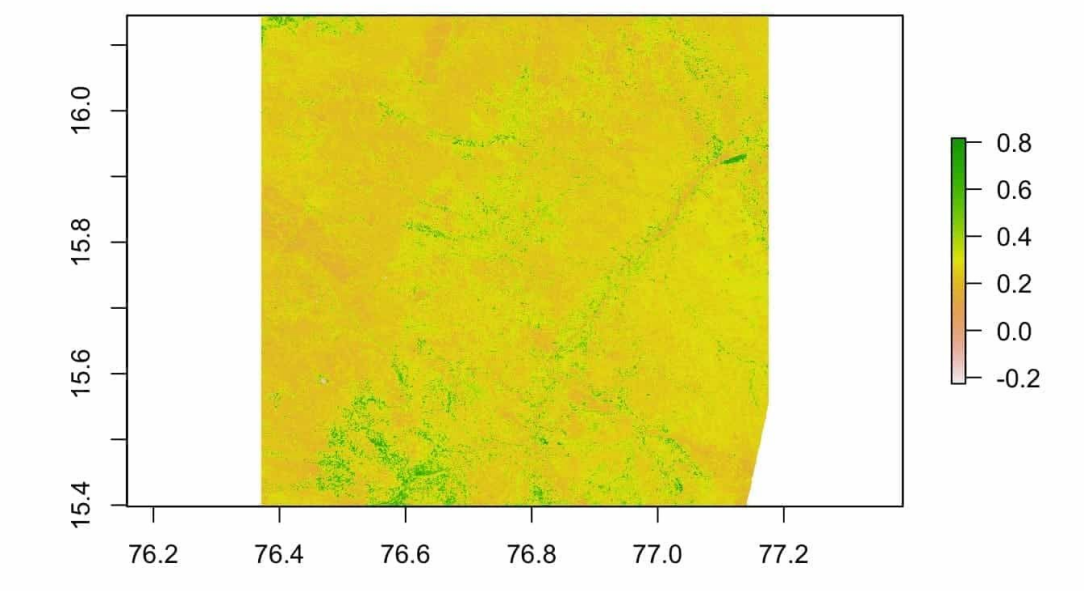
Negative values of NDVI (values approaching -1) correspond to water. Values close to zero (-0.1 to 0.1) generally correspond to barren areas of rock, sand or snow. Low, positive values (approximately 0.2 to 0.4) represent shrub and grassland, while high positive values (values approaching 1) indicate temperate and tropical rainforests.

> LEARN MORE:
>
> This document from Landsat gives more information about NVDI and all the other spectral indices you can create using Landsat data.

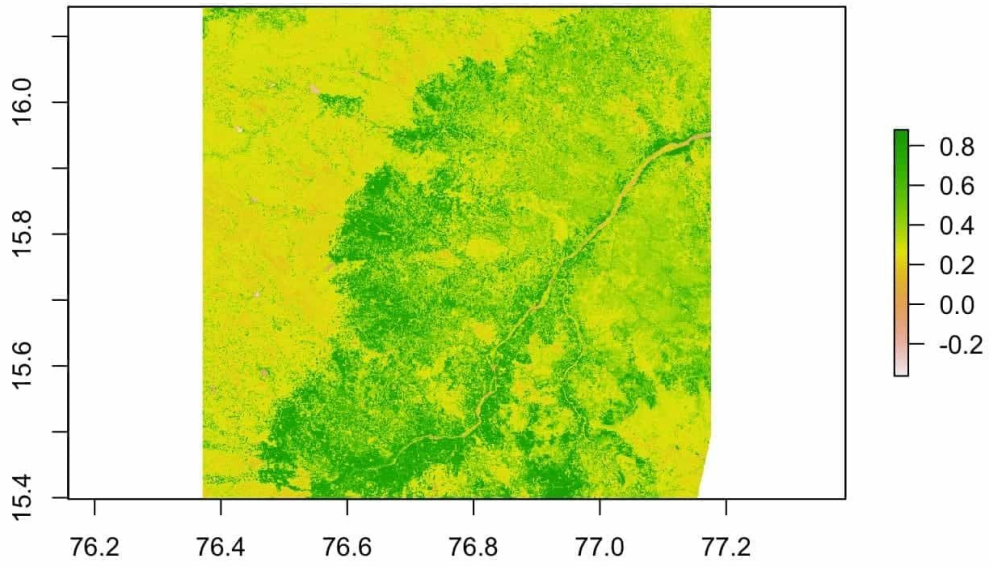
```
# treating raster as vectors work
karnataka_2014_ndvi = (karnataka_2014_east[[5]] - karnataka_2014_east
[[4]])/
                      (karnataka_2014_east[[5]] + karnataka_2014_east
[[4]])
karnataka_2017_ndvi = (karnataka_2017_east[[5]] - karnataka_2017_east
[[4]])/
                      (karnataka_2017_east[[5]] + karnataka_2017_east
[[4]])

# using getValues and setValues functions
karnataka_2018_ndvi = (getValues(karnataka_2018_east[[5]]) - getValues
(karnataka_2018_east[[4]]))/
                      (getValues(karnataka_2018_east[[5]]) + getValues
(karnataka_2018_east[[4]]))
karnataka_2018_ndvi = setValues(karnataka_2018_east[[5]], karnataka_20
18_ndvi)

# plot the NDVI for Karnataka
plot(karnataka_2014_ndvi)
```




```
plot(karnataka_2017_ndvi)
```



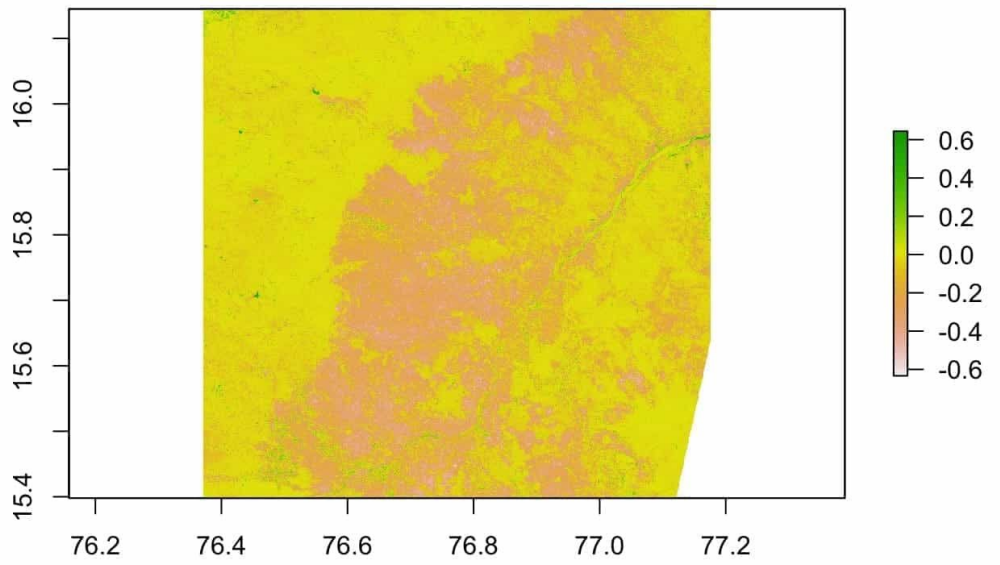

```
plot(karnataka_2018_ndvi)
```



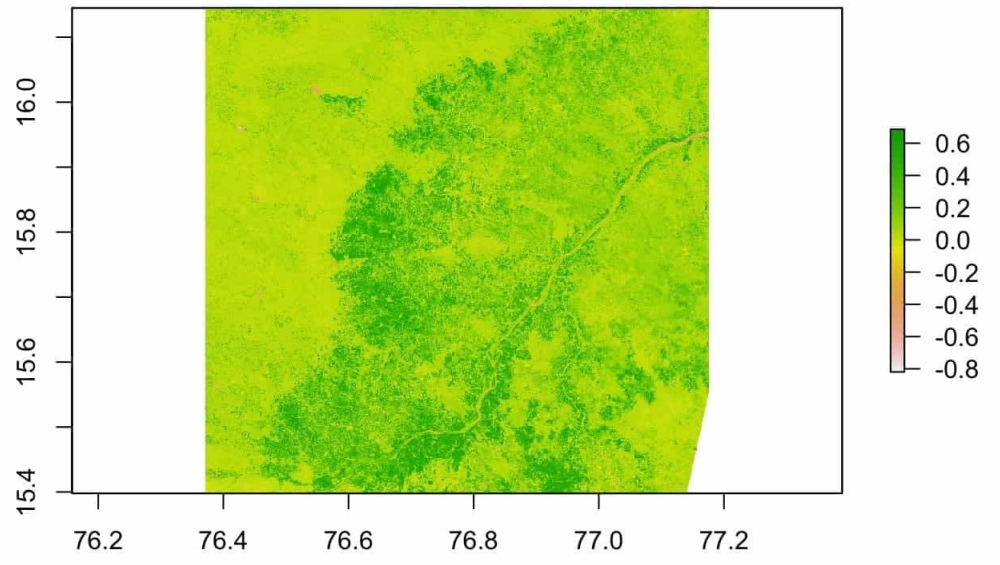
Which Regions Are Most Affected?

With this data, we can quantify the exact change in NDVI and look for the most affected regions in this rural part of Karnataka.

```
origin(karnataka_2017_ndvi) = origin(karnataka_2014_ndvi) = origin(kar
nataka_2018_ndvi)
ndvi_change_2017_2014 = karnataka_2017_ndvi - karnataka_2014_ndvi
ndvi_change_2018_2017 = karnataka_2018_ndvi - karnataka_2017_ndvi

plot(ndvi_change_2017_2014)
```




```
plot(ndvi_change_2018_2017)
```

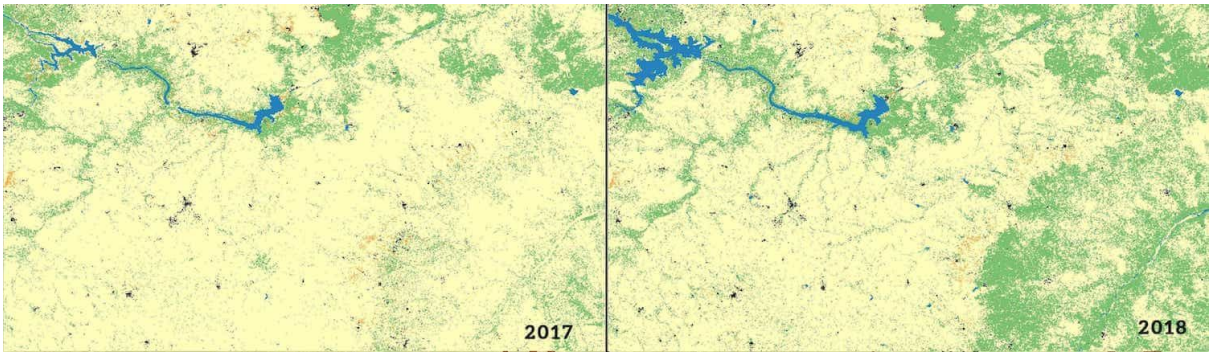


2017 was definitely a bad year with a massive decrease in NDVI, though 2018 seems to be better with a positive NDVI change.

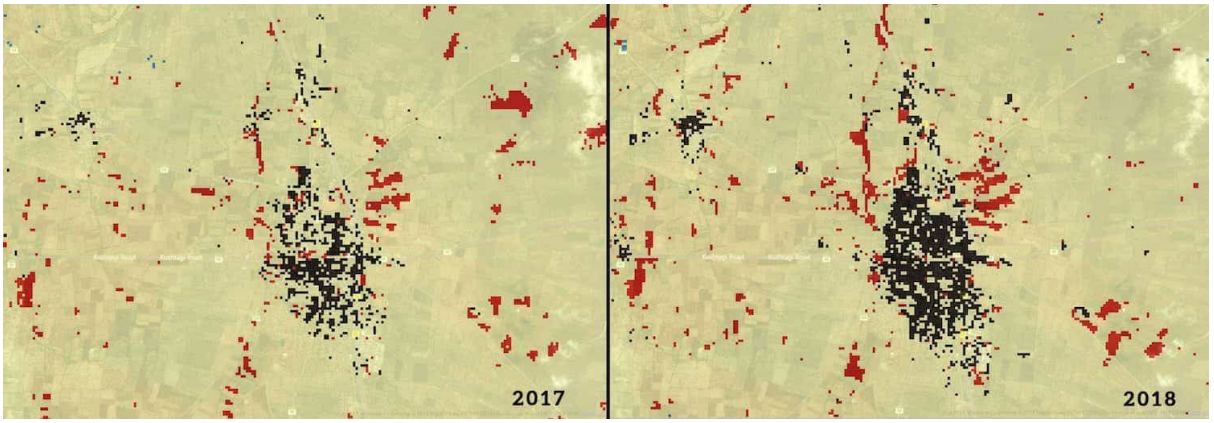
Final Thoughts

The power of spatial data is immense, and this is just the beginning of the sort of work that you can do with satellite imagery. In this lesson, with a few basic operations and visualizations on freely available data, we analyzed how vegetation in Karnataka changed over time. With the same data and techniques, we can do more complex analysis and apply machine learning techniques to further classify land into different types.

For example, at Atlan, we worked with Landsat 8 data in R to classify every piece of land in India into one of four categories: water, barren, green, or built-up regions. The results were really interesting — we could detect where and when new buildings and houses were being built, green regions turned into barren ones, and rivers dried up.



Land classification for Karnataka. (Black is built-up, yellow is barren, blue is water, and green is green land.)



Development in Karnataka. (Black is built-up, yellow is barren, and red is agricultural land.)

View this lesson online