Data Analysis and Visualization with R: Spatial

Department of Geography and Environmental Science Hunter College 695 Park Ave New York, NY 10

Last updated: March 24, 2022

Contents

Prerequisites and Preparations

To get the most out of this spatial section of **Data Analysis and Visualization with R**, you should have:

- **basic knowledge** of R/RStudio, generic data processing, and R plots covered in the first two sections.
- a recent version of R and RStudio on your computer.

Recommended Setting-up Steps:

- Create a new RStudio project ***R-spatial*** in a new folder ***R-spatial***.
- Create a new folder under ***R-spatial*** and name it ***data***.
- If you have your working directory set to ***R-spatial*** and it contains a folder called ***data***, you can copy and run the following lines in R:

```
download.file("http://www.geo.hunter.cuny.edu/~ssun/R-Spatial/data/R-spatial-data.zip",
              "R-spatial-data.zip");
unzip("R-spatial-data.zip", exdir = "data")
```

You can also download the data manually here R-spatial-data.zip and extract the files inside.

- Install and load the following libraries for spatial data handling:
  - `sf`
  - `sp`
  - `rgdal`
  - `raster`
  - `rgeos`
  - `dplyr`
- For spatial data mapping and visualization, install and load these additional libraries:
  - `classInt`
  - `RColorBrewer`
  - `ggplot2`
  - `ggmap`
  - `tmap`
  - `mapview`
  - `leaflet`(On Mac installing binary version is ok)
- You can use the standard methods of `install.packages` and `require` or `library` to install and load these R packages.

Most efficiently, you run the following script in RStudio to install all the packages that will be used for the spatial section:

```
# Load a list of packages. Install them first if they are not available.
# The list of packages to be installed
list.of.packages <- c("sf", "sp", "spatial", "maptools", "rgeos","rgdal",
                      "raster", "grid", "rasterVis",
                      "tidyverse", "magrittr", "ggpubr", "lubridate",
                      "devtools", "htmlwidgets", "mapview",
                      "classInt", "RColorBrewer", "ggmap", "tmap", "leaflet", "mapview",
                      "ggrepel", "ggsn",
                      "spdep","spatialreg","GWmodel");

# Check out the packages that have not been installed yet.
new.packages <- list.of.packages[!(list.of.packages %in% installed.packages()[,"Package"])]

# Install those missing packages first. It could take a long time for the first time.
if(length(new.packages)>0) install.packages(new.packages)

# Load all packages.
lapply(list.of.packages,function(x) {
  require(x,character.only = TRUE,quietly = TRUE)
})

# Or load specific packages when needed
require(sp); require(sf); library(raster)
```

References

Bivand, RS., Pebesma, E., Gómez-Rubio, V. (2013): Applied Spatial Data Analysis with R

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

Lovelace, R., Nowosad, J., Muenchow. J. (2019): Geocomputation with R

Spatial Data Analysis and Modeling with R

CRAN Task View: Analysis of Spatial Data

Engel, C. (2019). R for Geospatial Analysis and Mapping. The Geographic Information Science & Technology Body of Knowledge (1st Quarter 2019 Edition), John P. Wilson (Ed.). DOI:10.22224/gistbok/2019.1.3.

Acknowledgements

The materials for this section are adapted from https://cengel.github.io/R-spatial/ developed by Dr Claudia A Engel.

```
## Warning: package 'rgdal' was built under R version 4.1.3
```

```
## Warning: package 'sf' was built under R version 4.1.3
```

Chapter 1

Introduction to spatial data in R

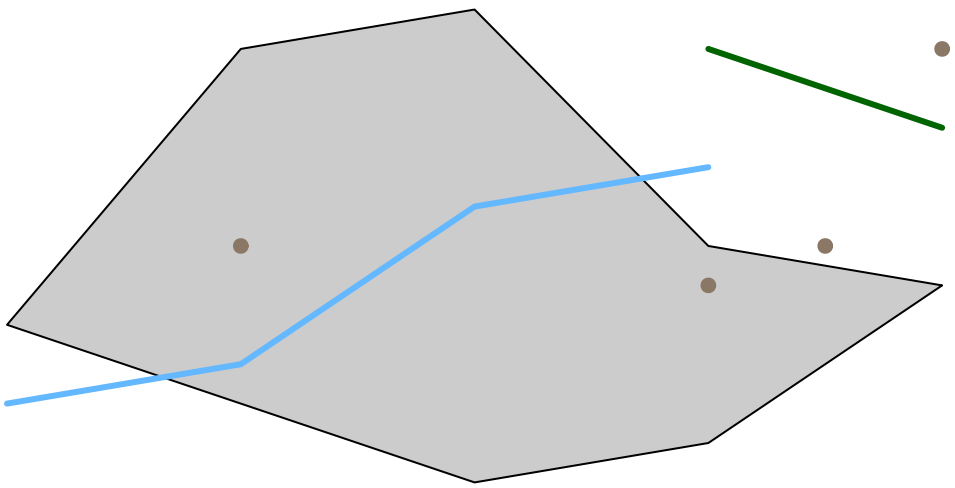
Learning Objectives

- Understand the structure of ***sf*** objects
- Read spatial data into ***sf*** objects
- Create ***sf*** objects from coordinate columns
- Simple static and interactive mapping with ***sf***

- ~~Read GeoTiff single and multiband into a ***raster*** object~~.
- ~~Examine ***raster*** objects~~

1.1 Spatial vector data model in R

In the vector model of spatial data, we have types of points (Points, MultiPoints), lines (LinesStrings, MultiLineStrings), polygons (Polygons, MultiPolygons), and a combination of them (GeometryCollection).



But these are pure geometries defined in the Euclidean space with Cartesian coordinates. we need to refer such data to physical locations on the Earth surface, that is geographically referenced. In order to do that, the software need to know extra information about the shape of the 3D Earth surface and how locations on the 3D surface can be ***projected*** or ***transformed*** to 2D surfaces like our paper maps and computer screens. The spatial reference systems or coordinate reference systems (CRS) are the solution, which have been fully developed in Geographic Information Science or Geographic Information Systems.

- *In the next session, we will continue to examine some of the details of CRS. In this session, we start to learn how it is defined and used.*
- *Geographic data, by definition, are spatial data with reference to the Earth surface. Spatial data, by contrast, could refer to any surfaces or spaces. In other words, all geographic data are spatial data but some spatial data are not geographic data. While mostly interchangeable, they could usefully differentiate them in some contexts.*

1.1.1 Handling spatial data in R

Spatial data analysis has always been an important component in the R ecosystem. For vector spatial data in R, there are two families of data structures based on two packages.

- ***sp* family**: used to be *de facto* standard and still popular
  - *sp* means *spatial* and defines a family of spatial data classes.
  - some of the top level classes are SpaitalPointsDataFrame, SpatialMultiPointsDataFrame, SpatialLinesDataFrame, and SpatialPolygonsDataFrame.
  - The ESRI Shapefile is closest to *sp* in terms of data organization.
- ***sf* family**: the newer and "modern" standard
  - *sf* means *Simple Feature* as defined by Open Geospatial Consortium or OGC.

- It is faster, more efficient, and consistent with other software like PostGIS.
- *sf* is supported by the R Consortium and therefore "official."
- It is almost the same as the table format in spatial databases like PostGIS and SpatiaLite.
- *sf* is tidyverse compatible and works well with tidy-packages like **dplyr**, **ggplot**, and **tidycensus**

One must understand that *sp* and *sf* are not two rivals in the R ecosystem. Instead, *sf* is more like a natural evolution of *sp*. First, there is demand to **unify** the data models across the entire geospatial industry so that GIS software and spatial databases are compatible and interoperable. OGC is leading the development of such geospatial standards. Most organizations and products like QGIS, PostGIS, and ArcGIS are increasingly adopting them. And *sf* was a response from the R spatial community to these standards.

Second, R now has more advanced data processing features and better fundamental packages such as those in the *tidyverse*. For geospatial data, more efficient open source spatial libraries also become available based on libraries like RGEOS, GDAL, PROJ, boost geometry etc. The new *sf* package can better take advantage of these developments.

Practically, it is very easy to convert between *sp* and *sf* structures as their underlying spatial models are compatible. And some of the key developers are actually working on both packages. Many R spatial packages can use both. But in case a package can only use one of them, we must make explicit conversions, mostly from *sf* to *sp* because most classic R spatial packages are based on *sp*.

That being said, when and where possible, we should always give the *sf*-family packages a higher priority as most packages are reworking towards being *tidyverse* compatible. Some packages related to the *sp* family packages are retiring. For this course, we focus on the *sf* package.

1.1.2 The *sf* package

The *sf*[1] package is first released on CRAN in late October 2016 . It implements a formal standard called "Simple Features" that specifies a storage and access model of spatial geometries such as points, lines, and polygons. A feature geometry is called simple when it meets certain criteria. For example, simple polygons cannot self-intersecting and they cannot have spikes or dangling vetexes. Simple Features are independent and have no explicit information about their neighbors or other spatially connected features. This standard has been adopted widely, not only by spatial databases such as PostGIS, but also web standards such as GeoJSON.

If you work with PostGIS or GeoJSON you may have come across the WKT (well-known text) format (Fig 1.1 and 1.2). Note that these texts are not what are being actually saved or encoded in the data. They are just for users' convenience as they are readable as opposed to the binary code.

sf implements the Simple Features standard natively in R. *sf* stores spatial objects as a simple data frame with a special column that contains the information for the geometry coordinates. That special column is a list with the same length as the number of rows in the data frame. Each of the individual list elements then can be of any length needed to hold the coordinates that correspond to an individual feature.

How simple features in R are organized

The three classes used to represent simple features are:

- ***sf***, the table (***data.frame***) with feature attributes and feature geometries, which contains

[1] E. Pebesma & R. Bivand (2016)Spatial data in R: simple features and future perspectives

Geometry primitives (2D)

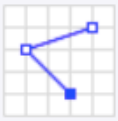
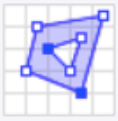
Type	Examples	
Point		`POINT (30 10)`
LineString		`LINESTRING (30 10, 10 30, 40 40)`
Polygon		`POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))`
		`POLYGON ((35 10, 45 45, 15 40, 10 20, 35 10), (20 30, 35 35, 30 20, 20 30))`

Figure 1.1: Well-Known-Text Geometry primitives (wikipedia)

Multipart geometries (2D)

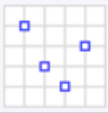
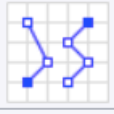
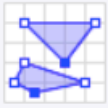
Type	Examples	
MultiPoint		`MULTIPOINT ((10 40), (40 30), (20 20), (30 10))`
		`MULTIPOINT (10 40, 40 30, 20 20, 30 10)`
MultiLineString		`MULTILINESTRING ((10 10, 20 20, 10 40), (40 40, 30 30, 40 20, 30 10))`
MultiPolygon		`MULTIPOLYGON (((30 20, 45 40, 10 40, 30 20)), ((15 5, 40 10, 10 20, 5 10, 15 5)))`
		`MULTIPOLYGON (((40 40, 20 45, 45 30, 40 40)), ((20 35, 10 30, 10 10, 30 5, 45 20, 20 35), (30 20, 20 15, 20 25, 30 20)))`

Figure 1.2: Well-Known-Text Multipart geometries (wikipedia)

- ***sfc***, the list-column with the geometries for each feature (record), which is composed of
- ***sfg***, the feature geometry of an individual simple feature.

```
Simple feature collection with 3883 features and 4 fields
geometry type:  POINT
dimension:      XY
bbox:           xmin: -75.26809 ymin: 39.87503 xmax: -74.95874 ymax: 40.13086
epsg (SRID):    NA
proj4string:    NA
First 10 features:
   DC_DIST SECTOR DISPATCH_DATE    TEXT_GENERAL_CODE                  geometry
1       22      1    2014-09-14  Homicide - Criminal POINT (-75.1568 39.98804)
2        1      B    2006-01-14  Homicide - Criminal POINT (-75.17873 39.92801)
3        1      B    2006-04-01  Homicide - Criminal POINT (-75.18275 39.92607)
4        1      B    2006-05-10  Homicide - Criminal POINT (-75.18092 39.92704)
5        1      E    2006-07-01  Homicide - Criminal POINT (-75.17204 39.92463)
6        1      F    2006-07-09  Homicide - Criminal POINT (-75.17612 39.92517)
7        1      B    2006-07-10  Homicide - Criminal POINT (-75.1785 39.92927)
8        1      J    2006-07-16  Homicide - Criminal POINT (-75.18419 39.92493)
9        1      J    2006-10-03  Homicide - Criminal POINT (-75.1843 39.92424)
10       1      A    2006-10-16  Homicide - Criminal POINT (-75.17702 39.92861)
```


Figure 1.3: Relationship of sf, sfc, and sfg

1.1.3 Create a spatial *sf* object manually

The basic steps of creating *sf* objects are bottom-up, that is from *sfg*, to *sfc*, then to *sf*.

I. Create geometric objects

Geometric objects (simple features) can be created from a numeric vector, matrix or a list with the coordinates. They are called *sfg* objects for Simple Feature Geometry. In *sf* package, there are functions that help create simple feature geometries, like `st_point()`, `st_linestring()`, `st_polygon()` and more.

II. Combine all individual single feature objects for the special column.

The feature geometries are then combined into a Simple Feature Collection with `st_sfc()`. which is nothing other than a simple feature geometry list-column. The *sfc* object also holds the bounding box and the projection information.

III. Add attributes.

Lastly, we add the attributes to the the simple feature collection with the `st_sf()` function. This function extends the well known data frame in R with a column that holds the simple feature collection.

Here we try to creat some highway objects. First, we would generate LINESTRINGs as simple feature geometries *sfg* out of a matrix with coordinates:

```
set.seed(1006) # for replicability
# runif(6) generates 6 random numbers in uniform distribution
lnstr_sfg1 <- st_linestring(matrix(runif(6), ncol=2))
set.seed(1043)
lnstr_sfg2 <- st_linestring(matrix(runif(6), ncol=2))
class(lnstr_sfg1)
```

```
#> [1] "XY"         "LINESTRING" "sfg"
```

We would then combine this into a simple feature collection *sfc*:

```
lnstr_sfc <- st_sfc(lnstr_sfg1, lnstr_sfg2) # just one feature here
class(lnstr_sfc)
```

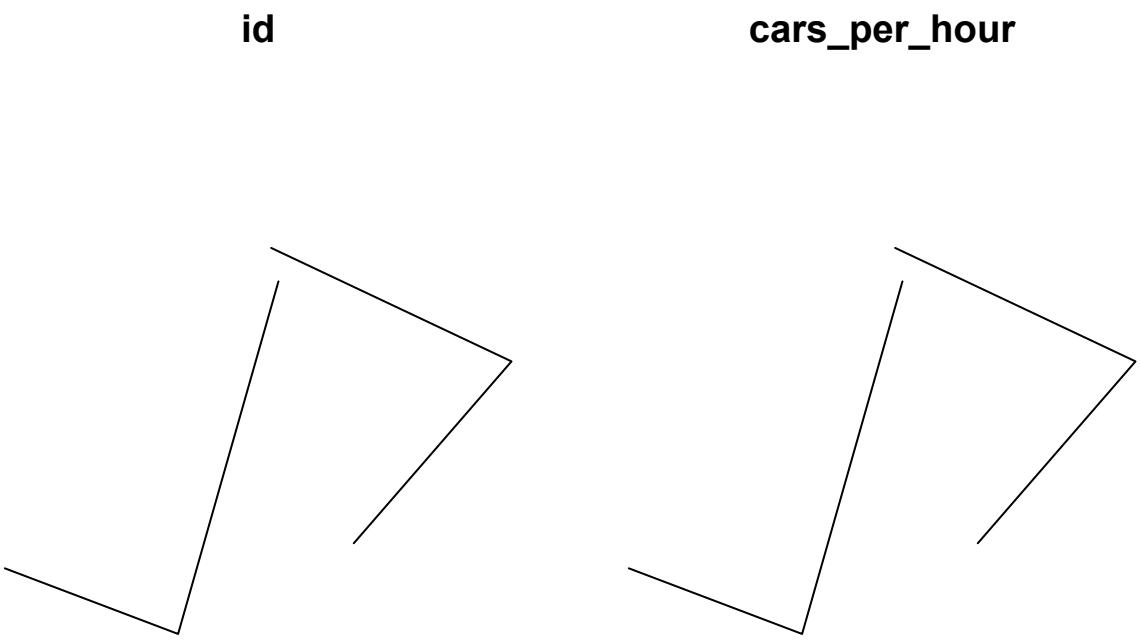
```
#> [1] "sfc_LINESTRING" "sfc"
```

And lastly use our data frame from above to generate the *sf* object:

```
dfr <- data.frame(id = c("hwy1", "hwy2"), cars_per_hour = c(78, 22))
lnstr_sf <- st_sf(dfr , lnstr_sfc)
class(lnstr_sf)
```

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

```
plot(lnstr_sf, col='black')
```



1.1.4 *sf* methods

There are many methods available in the *sf* package. Most function names, particularly those related to spatial data processing and analysis, start with **st_**, which means ***S***patial ***T***ype. To find out more, use

```
methods(class="sf")
```

```
#>   [1] $<-                  [                    [[<-
#>   [4] aggregate            anti_join            arrange
```

```
#>   [7] as.data.frame         cbind                coerce
#>  [10] dbDataType            dbWriteTable         distinct
#>  [13] dplyr_reconstruct     extent               extract
#>  [16] filter                full_join            gather
#>  [19] group_by              group_split          identify
#>  [22] initialize            inner_join           left_join
#>  [25] mapView               mask                 merge
#>  [28] mutate                nest                 pivot_longer
#>  [31] plot                  print                raster
#>  [34] rasterize             rbind                rename
#>  [37] right_join            rowwise              sample_frac
#>  [40] sample_n              select               semi_join
#>  [43] separate              separate_rows        show
#>  [46] slice                 slotsFromS3          spread
#>  [49] st_agr                st_agr<-             st_area
#>  [52] st_as_s2              st_as_sf             st_as_sfc
#>  [55] st_bbox               st_boundary          st_buffer
#>  [58] st_cast               st_centroid          st_collection_extract
#>  [61] st_convex_hull        st_coordinates       st_crop
#>  [64] st_crs                st_crs<-             st_difference
#>  [67] st_drop_geometry      st_filter            st_geometry
#>  [70] st_geometry<-         st_inscribed_circle  st_interpolate_aw
#>  [73] st_intersection       st_intersects        st_is
#>  [76] st_is_valid           st_join              st_line_merge
#>  [79] st_m_range            st_make_valid        st_nearest_points
#>  [82] st_node               st_normalize         st_point_on_surface
#>  [85] st_polygonize         st_precision         st_reverse
#>  [88] st_sample             st_segmentize        st_set_precision
#>  [91] st_shift_longitude    st_simplify          st_snap
#>  [94] st_sym_difference     st_transform         st_triangulate
#>  [97] st_union              st_voronoi           st_wrap_dateline
#> [100] st_write              st_z_range           st_zm
#> [103] summarise             transform            transmute
#> [106] ungroup               unite                unnest
#> see '?methods' for accessing help and source code
```

Here are some of the other highlights of *sf* that you might be interested in:

- provides **fast** I/O, particularly relevant for large files
- spatial functions that rely on GEOS and GDAL and PROJ external libraries are directly linked into the package, so no need to load additional external packages (like in *sp*)
- *sf* objects can be plotted directly with `ggplot`
- *sf* directly reads from and writes to spatial **databases** such as PostGIS
- *sf* is compatible with the `tidyverse`, (but see some pitfalls here)

Note that *sp* and *sf* are not the only way spatial objects are conceptualized in R. Other spatial packages may use their own class definitions for spatial data (for example `spatstat`).

There are packages specifically for the GeoJSON and for that reason are more lightweight, for

example:

- `geojson` and
- `geoops` - (demo)

Usually you can find functions that convert objects to and from these formats.

Self-Exercise

Similarly to the example above generate a Point object in R.

1. Create a matrix ***pts*** of random numbers with two columns and as many rows as you like. These are your points.
2. Create a dataframe ***attrib_df*** with the same number of rows as your ***pts*** matrix and a column that holds an attribute. You can make up any attribute.
3. Use the appropriate commands and ***pts*** to create an ***sf*** object with a geometry column of class ***sfc_POINT***.
4. Try to subset/filter your spatial object using the attribute you have added and the way you are used to from regular data frames.
5. How do you determine the bounding box of your spatial object?

1.2 Load and Display Data with *sf*

With good understanding of the *sf* data structure, we are ready to load and display some real data.

1.2.1 Reading Shapefiles into R

sf utilizes the powerful GDAL library to conduct data I/O (input/output), which is automatically linked in when loading *sf*. GDAL can handle numerous vector and raster data format. The `st_read()` method in *sf* is the interface for reading vector spatial data. In this section, we mostly use the Shapefile format.

```
# read in a Shapefile. With the 'shp' file extension, sf knows it is a Shapefile.
nyc_sf <- st_read("data/nyc/nyc_acs_tracts.shp")
```

```
#> Reading layer `nyc_acs_tracts' from data source
#>   `D:\Cloud_Drive\Dropbox (Hunter College)\Workspace\RSpace\R-Spatial_Book\data\nyc\nyc_acs_tr
#>   using driver `ESRI Shapefile'
#> Simple feature collection with 2166 features and 113 fields
#> Geometry type: MULTIPOLYGON
#> Dimension:     XY
#> Bounding box:  xmin: -74.25559 ymin: 40.49612 xmax: -73.70001 ymax: 40.91553
#> Geodetic CRS:  NAD83
```

```
# take a look at what we've got
str(nyc_sf) # note again the geometry column at the end of the output
```

```
#> Classes 'sf' and 'data.frame':   2166 obs. of  114 variables:
#>  $ UNEMP_RATE: num  0 0.0817 0.1706 0 0.088 ...
#>  $ cartodb_id: num  1 2 3 4 5 6 7 8 9 10 ...
#>  $ withssi   : int  0 228 658 0 736 0 261 0 39 638 ...
#>  $ withsocial: int  0 353 1577 0 1382 99 1122 10 264 895 ...
#>  $ withpubass: int  0 47 198 0 194 0 145 0 5 334 ...
```

```
#>  $ struggling: int  0 694 2589 0 2953 337 3085 17 131 1938 ...
#>  $ profession: int  0 0 36 0 19 745 18 49 35 0 ...
#>  $ popunemplo: int  0 92 549 0 379 321 432 26 106 494 ...
#>  $ poptot    : int  0 2773 8339 0 10760 7024 10955 849 1701 5923 ...
#>  $ popover18 : int  0 2351 6878 0 8867 6637 8932 711 1241 4755 ...
#>  $ popinlabou: int  0 1126 3218 0 4305 5702 5114 657 735 2283 ...
#>  $ poororstru: int  0 2026 4833 0 7044 1041 6376 112 232 4115 ...
#>  $ poor      : int  0 1332 2244 0 4091 704 3291 95 101 2177 ...
#>  $ pacificune: int  0 0 0 0 13 0 0 0 0 0 ...
#>  $ pacificinl: int  0 0 0 0 13 0 0 0 0 0 ...
#>  $ pacific   : int  0 0 0 0 13 0 0 0 0 0 ...
#>  $ otherunemp: int  0 0 103 0 46 0 0 0 17 151 ...
#>  $ otherinlab: int  0 144 348 0 609 112 184 48 90 525 ...
#>  $ otherethni: int  0 598 1175 0 1799 112 377 48 183 1664 ...
#>  $ onlyprofes: int  0 0 102 0 20 890 116 78 77 0 ...
#>  $ onlymaster: int  0 77 412 0 292 2162 408 233 328 16 ...
#>  $ onlylessth: int  0 878 2164 0 3793 121 4384 34 115 1846 ...
#>  $ onlyhighsc: int  0 1088 4068 0 3868 5508 3882 631 1093 2343 ...
#>  $ onlydoctor: int  0 0 66 0 1 145 98 29 42 0 ...
#>  $ onlycolleg: int  0 471 2355 0 2290 5371 2223 585 884 1111 ...
#>  $ onlybachel: int  0 271 1269 0 1322 4685 1262 481 687 200 ...
#>  $ okay      : int  0 742 3474 0 3499 5982 4579 733 1469 1798 ...
#>  $ mixedunemp: int  0 16 21 0 14 24 0 0 51 31 ...
#>  $ mixedinlab: int  0 72 134 0 100 136 91 10 62 140 ...
#>  $ mixed     : int  0 175 234 0 251 224 115 16 102 334 ...
#>  $ master    : int  0 77 310 0 272 1272 292 155 251 16 ...
#>  $ maleunempl: int  0 76 349 0 179 204 197 8 72 277 ...
#>  $ maleover18: int  0 1101 3134 0 4068 3557 3992 454 613 1935 ...
#>  $ malepro   : int  0 0 36 0 19 473 48 51 61 0 ...
#>  $ malemastr : int  0 10 143 0 126 1034 181 123 139 0 ...
#>  $ male_lesHS: int  0 302 1063 0 1693 29 2013 17 103 714 ...
#>  $ male_HS   : int  0 607 1893 0 1651 2985 1702 414 485 985 ...
#>  $ male_doctr: int  0 0 24 0 0 77 30 29 38 0 ...
#>  $ male_collg: int  0 227 1225 0 997 2924 871 381 368 462 ...
#>  $ male_BA   : int  0 132 668 0 562 2411 499 288 292 43 ...
#>  $ maleinlabo: int  0 612 1640 0 2059 3299 2466 423 392 991 ...
#>  $ maledrop  : int  0 16 8 0 0 0 0 0 0 0 ...
#>  $ male16to19: int  0 65 253 0 162 37 236 20 32 187 ...
#>  $ male      : int  0 1318 3850 0 4796 3840 5244 533 897 2514 ...
#>  $ lessthanhi: int  0 878 2164 0 3793 121 4384 34 115 1846 ...
#>  $ lessthan10: int  0 212 760 0 1100 289 625 2 51 463 ...
#>  $ households: int  0 986 3382 0 3838 4104 3950 367 739 2224 ...
#>  $ hispanicun: int  0 30 269 0 115 27 0 0 68 335 ...
#>  $ hispanicin: int  0 357 1171 0 819 297 236 77 201 1145 ...
#>  $ hispanic  : int  0 1187 3503 0 2608 374 318 108 360 3478 ...
#>  $ highschool: int  0 617 1713 0 1578 137 1659 46 209 1232 ...
#>  $ geo_state : int  36 36 36 36 36 36 36 36 36 36 ...
#>  $ geo_place : int  51000 51000 51000 51000 51000 51000 51000 51000 51000 51000 ...
#>  $ geo_county: int  61 61 61 61 61 61 61 61 61 61 ...
#>  $ field_1   : int  1 2 3 4 5 6 7 8 9 10 ...
```

```
#>  $ femaleunem: int  0 16 200 0 200 117 235 18 34 217 ...
#>  $ femaleover: int  0 1250 3744 0 4799 3080 4940 257 628 2820 ...
#>  $ fem_profes: int  0 0 66 0 1 417 68 27 16 0 ...
#>  $ fem_master: int  0 67 269 0 166 1128 227 110 189 16 ...
#>  $ fem_lessHS: int  0 576 1101 0 2100 92 2371 17 12 1132 ...
#>  $ fem_HS    : int  0 481 2175 0 2217 2523 2180 217 608 1358 ...
#>  $ fem_doctor: int  0 0 42 0 1 68 68 0 4 0 ...
#>  $ fem_colleg: int  0 244 1130 0 1293 2447 1352 204 516 649 ...
#>  $ fem_BA    : int  0 139 601 0 760 2274 763 193 395 157 ...
#>  $ femaleinla: int  0 514 1578 0 2246 2403 2648 234 343 1292 ...
#>  $ femaledrop: int  0 0 1 0 0 0 0 0 0 14 ...
#>  $ femal16_19: int  0 84 124 0 271 2 126 2 32 242 ...
#>  $ female    : int  0 1455 4489 0 5964 3184 5711 316 804 3409 ...
#>  $ europeanun: int  0 14 328 0 56 188 71 26 22 132 ...
#>  $ europeanin: int  0 303 1641 0 525 4058 492 540 543 510 ...
#>  $ european  : int  0 540 4091 0 1181 4975 929 717 1327 1645 ...
#>  $ doctorate : int  0 0 66 0 1 145 98 29 42 0 ...
#>  $ com_90plus: int  0 49 101 0 36 61 82 17 27 178 ...
#>  $ comm_5less: int  0 99 0 0 1 220 46 26 0 0 ...
#>  $ comm_60_89: int  0 40 215 0 370 136 687 32 41 331 ...
#>  $ comm_5_14 : int  0 121 226 0 331 913 718 87 58 99 ...
#>  $ comm_45_59: int  0 142 171 0 284 287 409 46 84 351 ...
#>  $ comm_30_44: int  0 217 923 0 1442 1514 1118 198 210 493 ...
#>  $ comm_15_29: int  0 352 970 0 1250 1840 1363 192 134 304 ...
#>  $ college   : int  0 200 1086 0 968 686 961 104 197 911 ...
#>  $ bachelor  : int  0 194 857 0 1030 2523 854 248 359 184 ...
#>  $ asianunemp: int  0 62 38 0 207 108 352 0 16 61 ...
#>  $ asianinlab: int  0 559 615 0 2736 1286 4283 42 35 491 ...
#>  $ asian     : int  0 1249 1724 0 6549 1598 9448 51 75 922 ...
#>  $ americanun: int  0 0 0 0 0 1 0 0 0 24 ...
#>  $ americanin: int  0 0 0 0 57 1 0 0 0 24 ...
#>  $ american  : int  0 43 0 0 57 1 0 0 0 51 ...
#>  $ africanune: int  0 0 59 0 43 0 9 0 0 95 ...
#>  $ africaninl: int  0 48 434 0 265 109 64 17 5 593 ...
#>  $ african   : int  0 168 1115 0 910 114 86 17 14 1307 ...
#>  $ puma      : chr  "3810" "3809" "3809" "3810" ...
#>  $ ntaname   : chr  "park-cemetery-etc-Manhattan" "Lower East Side" "Lower East Side" "park-ce
#>  $ ntacode   : chr  "MN99" "MN28" "MN28" "MN99" ...
#>  $ ctlabel   : chr  "1" "2.01" "2.02" "5" ...
#>  $ cdeligibil: chr  "I" "E" "E" "I" ...
#>  $ boroname  : chr  "Manhattan" "Manhattan" "Manhattan" "Manhattan" ...
#>  $ medianinco: num  NA 17282 24371 NA 18832 ...
#>  $ medianagem: num  NA 39.3 44.9 NA 43.4 29.7 42 33.2 41.5 31.2 ...
#>  $ medianagef: num  NA 43.8 47.9 NA 43 28.3 43 31.9 45.7 47.5 ...
#>   [list output truncated]
#>  - attr(*, "sf_column")= chr "geometry"
#>  - attr(*, "agr")= Factor w/ 3 levels "constant","aggregate",..: NA NA NA NA NA NA NA NA NA NA
#>   ..- attr(*, "names")= chr [1:113] "UNEMP_RATE" "cartodb_id" "withssi" "withsocial" ...
```

Two more words about the geometry column: The geometry column does not have to be named

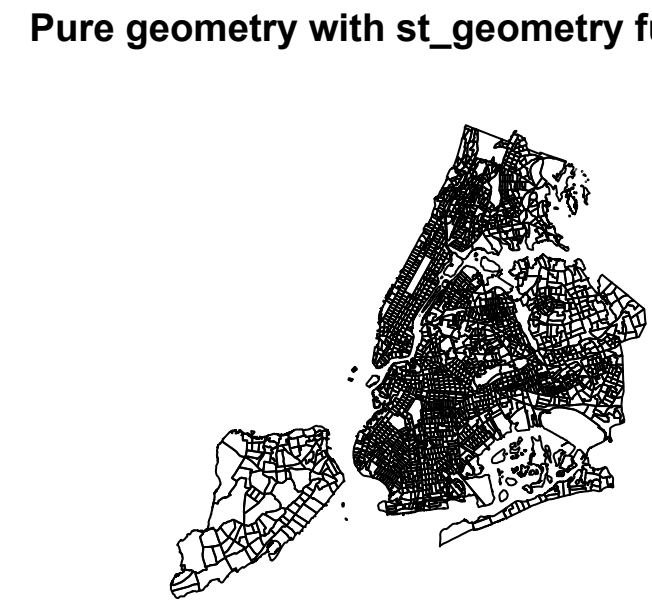
"geometry". You can name this column any way you wish. "geom", for example, is another popular name for the geometry column. Secondly, you can remove this column and revert to a regular, non-spatial dataframe at any time with `st_drop_geometry()`.

1.2.2 Simple *sf* plotting

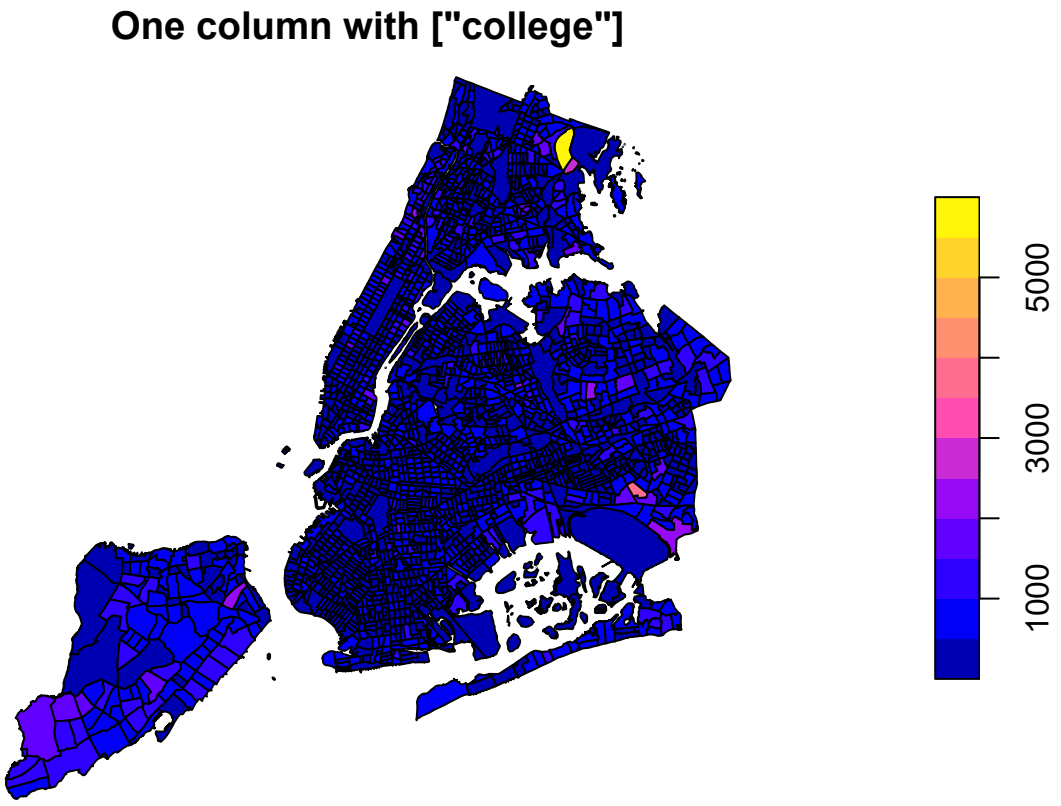
The default `plot` of an *sf* object is a multi-plot of the first attributes, with a warning if not all can be plotted. By default, it plots the first 9 columns:

In order to only plot the polygon boundaries we need to directly use the geometry column. We use the `st_geometry()` function to extract it. We can also select a single column or a few columns to draw.

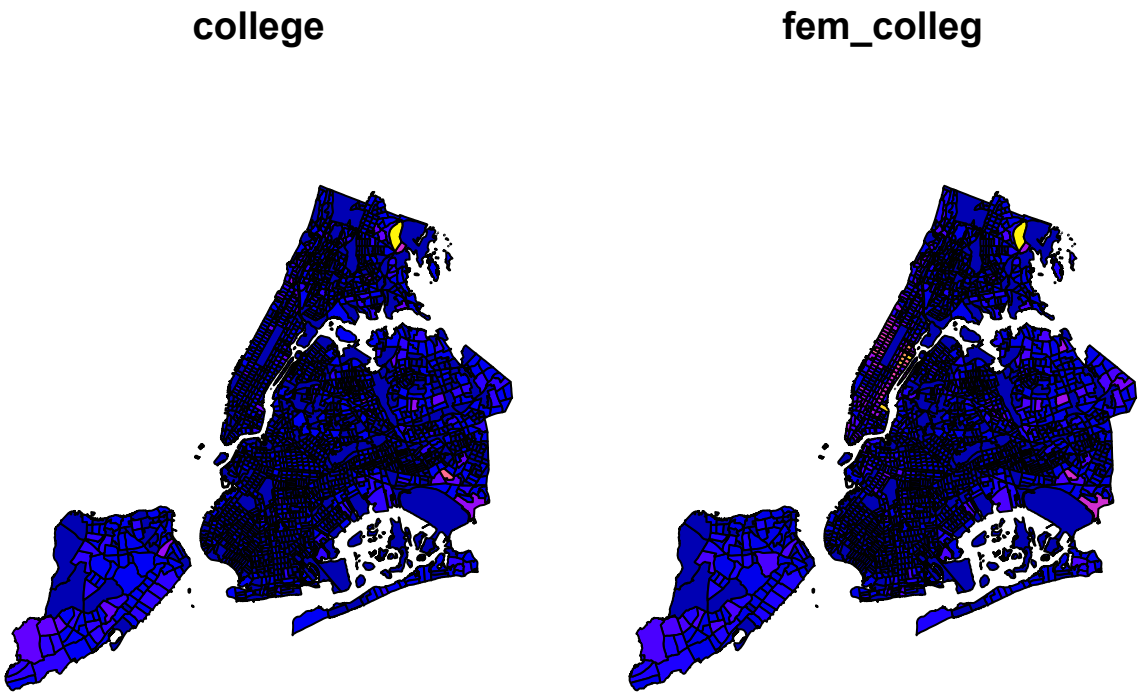
```
# pure geometry
plot(st_geometry(nyc_sf), main='Pure geometry with st_geometry function')
```




```
# Only the college column, with geometry
plot(nyc_sf['college'], main='One column with ["college"]')
```




```
# Two columns, with geometry
plot(nyc_sf[c('college', 'fem_colleg')], main='Two Columns with ["college", "fem_colleg"]')
```



Let's add a subset of polygons with only the census tracts where the median household income is more than $80,000. We can extract elements from an *sf* object based on attributes using your preferred method of filtering dataframes.

```
# Make a plot using all tracts, then add the "rich" in red
plot(st_geometry(nyc_sf))
plot(st_geometry(nyc_sf_rich), add=T, col="red")
```

tidyverse and piping work as well!

```
plot(nyc_sf %>% st_geometry(), col = 'NA', border='grey', lwd=0.5)

nyc_sf %>%
  filter(medianinco > 80000) %>%
  st_geometry() %>%
  plot(col="red", add=T)
```

1.2.3 Simple interactive mapping with *mapview*

One of most useful, yet simplest methods of visualizing a *sf* object or other spatial data is to utilize the *mapview* package for interactive mapping. *mapview* utilizes the *leaflet* package in R and the original JavaScript *leaflet* library for web mapping. While we will learn the details of *leaflet* later, here is a quick application of the *mapview* package.

```
# We use all the default options, but select a few columns.
# If we use all columns, the popup window will be too big.
nyc_sf %>% dplyr::select(boroname, poptot, female, doctorate) %>%
  mapview::mapview()
```

```
#> PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed,
```

Further configuration could be applied.

```
nyc_sf %>% dplyr::select(boroname, poptot, female, doctorate) -> tmp_sf

mapview(tmp_sf, zcol='boroname', layer.name = 'Borough', stroke=FALSE) +
```

```
  mapview(tmp_sf, zcol='poptot',
          layer.name = 'Total Population',
          homebutton=FALSE,
          legend=FALSE) +
  mapview(tmp_sf, zcol="doctorate",
          layer.name = 'Population with Doctorate',
          homebutton=FALSE,
          legend=FALSE,
          label="doctorate",
          popup = 'doctorate')
```

mapview is particularly suitable for quick and simple interactive mapping. However, the cost of convenience and simplicity is the loss of controls to many details. For example, it is very difficult, if ever possible, to keep the first layer while turning off all other layers using code. With *leaflet*, by contrast, this is an easy task.

1.3 Creating a spatial object from a lat/lon table

Increasingly, we are using GPS or GPS-compatible devices to collect spatial data or geographically referenced data in the field. Such data are often organized in a spreadsheet that contains latitude, longitude and some attribute values. We have learned many different ways of reading such spreadsheet into a dataframe. We can then very easily convert the table into a spatial object in R.

1.3.1 With *sf*

An *sf* object can be created from a data frame in the following way. We take advantage of the `st_as_sf()` function which converts any foreign object into an *sf* object. Similarly to above, it requires an argument `coords`, which in the case of point data needs to be a vector that specifies the dataframe's columns for the longitude and latitude (x,y) coordinates.

```
`my_sf_object <- st_as_sf(myDataframe, coords)`
```

`st_as_sf()` creates a new object and leaves the original data frame untouched.

We use `read_csv()` in the `readr` package to read `manhattan_noise.csv` into a dataframe in R and name it `manhattan_noise_df`. Alternatively, we can use the `read.csv()` in the base package.

```
# read_csv has better capabilities for data types like datetime.
manhattan_noise_df <- read_csv("data/nyc/manhattan_noise.csv",
                               show_col_types = FALSE,
                               lazy = FALSE) %>%
  dplyr::mutate(datetime = lubridate::force_tz(datetime, 'America/New_York'))

#manhattan_noise_df <- read.csv("data/nyc/manhattan_noise.csv")
str(manhattan_noise_df)
```

```
#> tibble [68,582 x 7] (S3: tbl_df/tbl/data.frame)
#>  $ datetime              : POSIXct[1:68582], format: "2020-07-25 15:02:08" "2020-07-25 02
#>  $ descriptor            : chr [1:68582] "Loud Music/Party" "Loud Music/Party" "Loud Musi
#>  $ incident_zip          : num [1:68582] 10040 10031 10034 10034 10003 ...
#>  $ incident_address      : chr [1:68582] "281 WADSWORTH AVENUE" "600 WEST  142 STREET" "3
#>  $ open_data_channel_type: chr [1:68582] "MOBILE" "ONLINE" "ONLINE" "ONLINE" ...
```

```
#>  $ latitude              : num [1:68582] 40.9 40.8 40.9 40.9 40.7 ...
#>  $ longitude             : num [1:68582] -73.9 -74 -73.9 -73.9 -74 ...
```

In the real world, things would not be so smooth as many do not have appropriate knowledge on geography and GPS. Quite often, the coordinate columns are not ready for direct conversion. Commonly problems include:

- coordinates are in degree, minute, and second format instead of decimal degrees.
- The +/- signs in coordinates are replaced by N(orth), S(outh), W(est), E(ast).
- Longitude and latitude are switched or labeled the wrong way.
- Missing the negative signs in coordinates when there should be.
- Coordinates are mixed with text values.

With general data processing power in R, these issues can be fixed relatively easily.

We convert the `manhattan_noise_df` data frame into an *sf* object with `st_as_sf()`

```
# Note the two columns for coords must exist in the dataframe
manhattan_noise_sf <- st_as_sf(manhattan_noise_df,
                               coords = c("longitude", "latitude"))
str(manhattan_noise_sf)
```

```
#> sf [68,582 x 6] (S3: sf/tbl_df/tbl/data.frame)
#>  $ datetime              : POSIXct[1:68582], format: "2020-07-25 15:02:08" "2020-07-25 02:58:2
#>  $ descriptor            : chr [1:68582] "Loud Music/Party" "Loud Music/Party" "Loud Music/Par
#>  $ incident_zip          : num [1:68582] 10040 10031 10034 10034 10003 ...
#>  $ incident_address      : chr [1:68582] "281 WADSWORTH AVENUE" "600 WEST  142 STREET" "33 IND
#>  $ open_data_channel_type: chr [1:68582] "MOBILE" "ONLINE" "ONLINE" "ONLINE" ...
#>  $ geometry              :sfc_POINT of length 68582; first list element:  'XY' num [1:2] -73.9
#>  - attr(*, "sf_column")= chr "geometry"
#>  - attr(*, "agr")= Factor w/ 3 levels "constant","aggregate",..: NA NA NA NA NA
#>   ..- attr(*, "names")= chr [1:5] "datetime" "descriptor" "incident_zip" "incident_address" ..
```

Note the additional **geometry** list-column which now holds the simple feature collection with the coordinates of all the points.

To make it a complete geographical object we assign the WGS84 projection, which has the EPSG code 4326:

```
st_crs(manhattan_noise_sf)
```

```
#> Coordinate Reference System: NA
```

```
st_crs(manhattan_noise_sf) <- 4326 # we can use EPSG as numeric here
# Alternatively, we can use the sf::st_set_crs method
# st_set_crs(philly_homicides_sf, 4326)
st_crs(manhattan_noise_sf)
```

```
#> Coordinate Reference System:
#>   User input: EPSG:4326
#>   wkt:
#> GEOGCRS["WGS 84",
#>     DATUM["World Geodetic System 1984",
#>         ELLIPSOID["WGS 84",6378137,298.257223563,
#>             LENGTHUNIT["metre",1]]],
```

```
#>     PRIMEM["Greenwich",0,
#>         ANGLEUNIT["degree",0.0174532925199433]],
#>     CS[ellipsoidal,2],
#>         AXIS["geodetic latitude (Lat)",north,
#>             ORDER[1],
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>         AXIS["geodetic longitude (Lon)",east,
#>             ORDER[2],
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>     USAGE[
#>         SCOPE["Horizontal component of 3D system."],
#>         AREA["World."],
#>         BBOX[-90,-180,90,180]],
#>     ID["EPSG",4326]]
```

- Geographic coordinates from GPS are always using the WGS84 projection, which has a EPSG code of 4326.
- EPSG stands for European Petroleum Survey Group. The map projections are too complicated for many engineers in the petroleum industry. So they started using numeric codes, SRID, to denote spatial reference systems (SRS). Such simplicity has also been favored by software engineers, so they also use EPSG extensively. By contrast, GIS professionals and users are supposedly to have good knowledge on map projection and reference systems. As a result, classic GIS software like ArcGIS have just started using EPSG codes.

- There are many websites that allow us to query EPSG codes. The best one, in my opinion, is Spatial Reference, which provides the projection and spatial reference information in many different formats and for many applications.

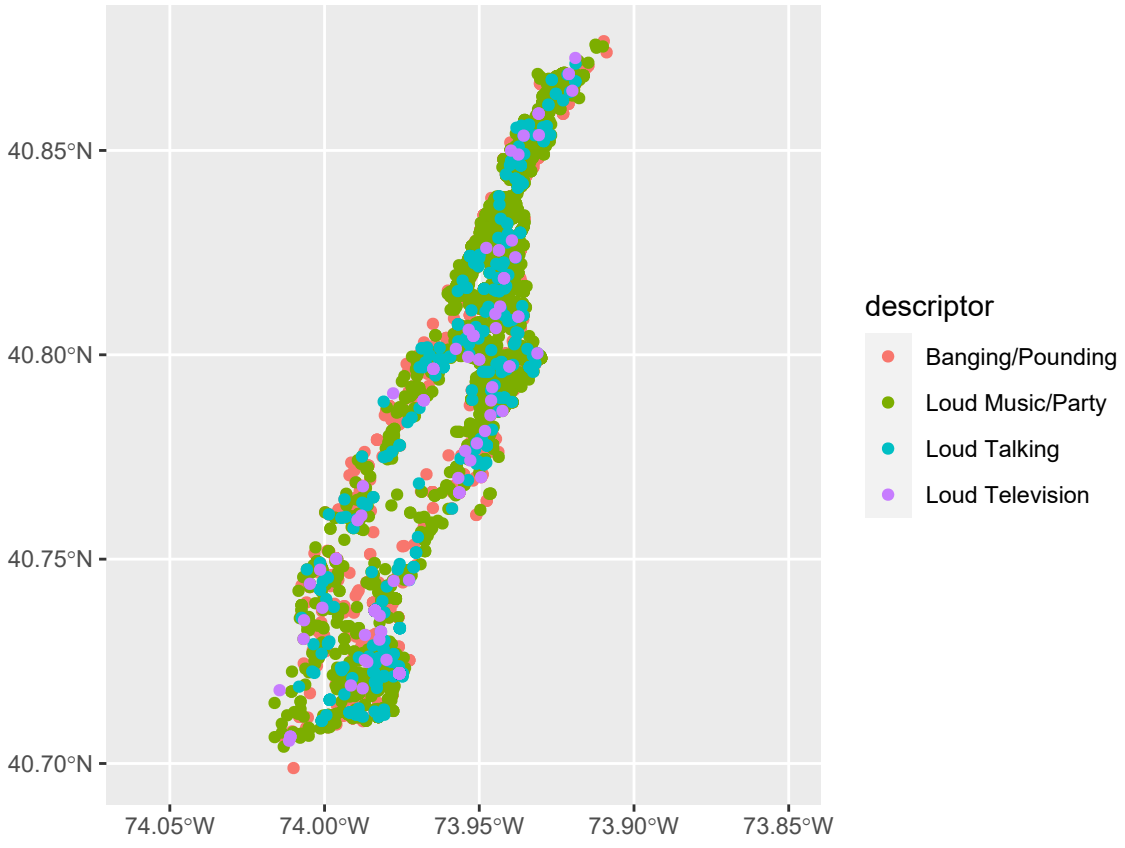
Now, the *sf* object looks like follows. Note how the SRID and prj4string are filled.

```
manhattan_noise_sf
```

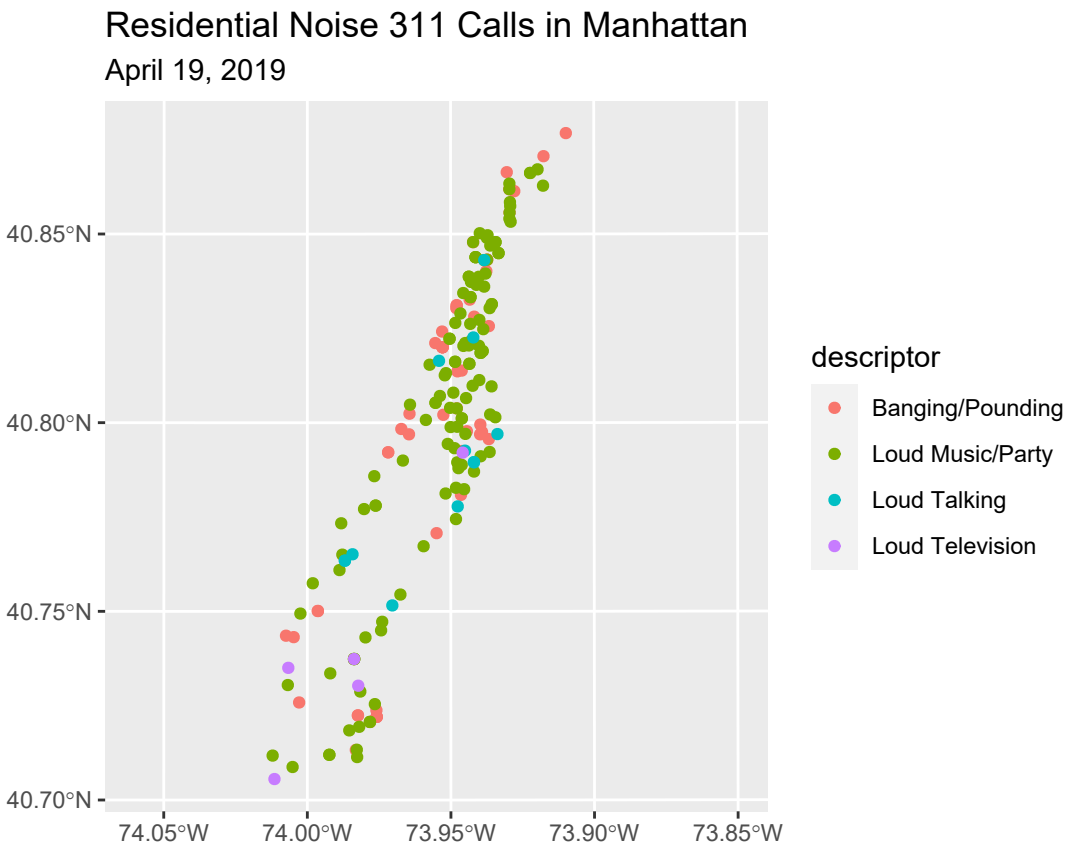
```
#> Simple feature collection with 68582 features and 5 fields
#> Geometry type: POINT
#> Dimension:     XY
#> Bounding box:  xmin: -74.018 ymin: 40.69889 xmax: -73.90809 ymax: 40.87783
#> Geodetic CRS:  WGS 84
#> # A tibble: 68,582 x 6
#>    datetime            descriptor incident_zip incident_address open_data_chann~
#>  * <dttm>              <chr>             <dbl> <chr>            <chr>
#>  1 2020-07-25 15:02:08 Loud Musi~        10040 281 WADSWORTH A~ MOBILE
#>  2 2020-07-25 02:58:27 Loud Musi~        10031 600 WEST  142 S~ ONLINE
#>  3 2020-07-25 21:35:59 Loud Musi~        10034 33 INDIAN ROAD   ONLINE
#>  4 2020-07-26 01:33:28 Loud Musi~        10034 65 VERMILYEA AV~ ONLINE
#>  5 2020-07-25 02:07:27 Loud Musi~        10003 317 2 AVENUE     ONLINE
#>  6 2020-07-25 20:34:37 Loud Talk~        10009 131 AVENUE B     MOBILE
#>  7 2020-07-26 00:30:54 Loud Talk~        10026 358 WEST  118 S~ MOBILE
#>  8 2020-07-26 01:36:19 Loud Musi~        10032 502 WEST  170 S~ ONLINE
#>  9 2020-07-25 23:21:11 Loud Musi~        10033 600 WEST  183 S~ PHONE
#> 10 2020-07-25 21:12:56 Loud Musi~        10016 134 EAST   28 S~ PHONE
#> # ... with 68,572 more rows, and 1 more variable: geometry <POINT [°]>
```

Here is a quick, simple plot of the data to verify everything is good. First, we select one day from the dataset to simplify the plot. As *sf* is tidyverse compatible, we can use `dplyr::filter` to do that. The `lubridate` package in tidyverse can help us process `datetime` data. Then, we use `ggplot` and `geom_sf` to plot the data. Last, we use a free basemap from Stamen for the plotting. If the dataset is wrong, particularly its coordinates reference system, you will see the locations of those geometries will be way off or completely disappear. We will learn more about the mapping in future sessions.

```
ggplot(data = manhattan_noise_sf  %>%
         dplyr::filter(month(datetime) == 4 & year(datetime) > 2019) ) +
  geom_sf(aes(color=descriptor)) +
  coord_sf(xlim = c(-74.06, -73.85), default_crs = sf::st_crs(4326) )
```




```
man_day_noise_sf <- manhattan_noise_sf  %>%
         dplyr::filter(day(datetime) == 19 & month(datetime) == 4 & year(datetime) > 2019);

ggplot(data = man_day_noise_sf) +
  geom_sf(aes(color=descriptor)) +
  coord_sf(xlim =  c(-74.06, -73.85), crs = sf::st_crs(2831), default_crs = sf::st_crs(4326)) +
  labs(title = "Residential Noise 311 Calls in Manhattan", subtitle = 'April 19, 2019')
```



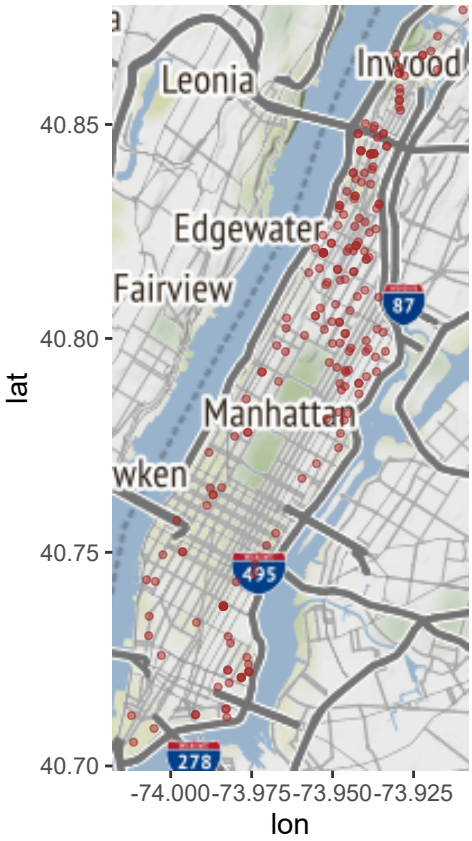
Now, we use a base map to plot it again.

```
# We can get a simple basemap. Adjust zoom with the map extent.
# Higher zoom means more details but needs more time to load.
# Start from small and increase to an appropriate level. Will learn more on this.

# In most cases we can use the extent of the data to retrieve the basemap
manhattan_noise_sf %>% st_bbox() %>% as.vector() %>%
  ggmap::get_stamenmap(zoom = 11, messaging = FALSE) -> baseMap;
```

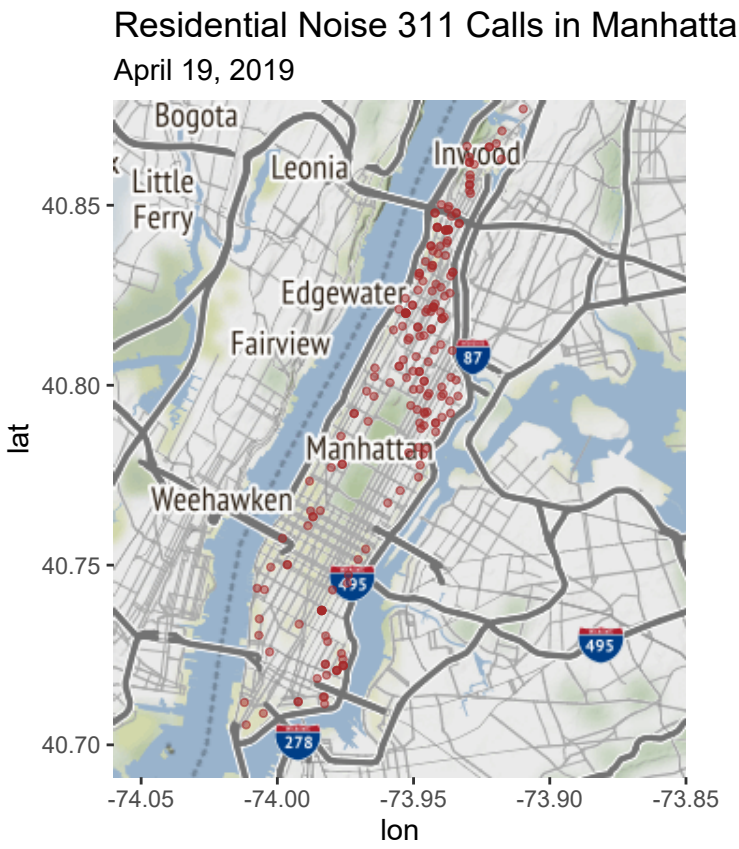
```
#> Source : http://tile.stamen.com/terrain/11/602/768.png
#> Source : http://tile.stamen.com/terrain/11/603/768.png
#> Source : http://tile.stamen.com/terrain/11/602/769.png
#> Source : http://tile.stamen.com/terrain/11/603/769.png
#> Source : http://tile.stamen.com/terrain/11/602/770.png
#> Source : http://tile.stamen.com/terrain/11/603/770.png
```

```
# This is the ggplot style plotting. Again, will come back on this for more.
ggmap(baseMap) +
  geom_point(aes(x=X, y=Y),
             data = man_day_noise_sf %>% st_coordinates() %>% tibble::as_tibble(),
             color = 'brown',
             size = 1,
             alpha = .5
```

```
            )
```




```
# but for the long shape of Manhattan, it might be better to manually choose the basemap
baseMapExt <- c(-74.06, 40.691, -73.85, 40.879)

ggmap::get_stamenmap(baseMapExt, zoom = 11, messaging = FALSE) -> baseMap;

ggmap(baseMap) +
  geom_point(aes(x=X, y=Y),
             data = man_day_noise_sf %>% st_coordinates() %>% tibble::as_tibble(),
             color = 'brown',
             size = 1,
             alpha = .5
             ) +
  labs(title = "Residential Noise 311 Calls in Manhattan", subtitle = 'April 19, 2019')
```



Of course, *mapview* comes in handy for creating simple interactive maps quickly.

```
mapview(man_day_noise_sf, zcol='descriptor', layer.name='Manhattan Noise')
```

1.3.2 *sp* and *sf* conversion

While *sf* is the new standard, many classic packages are still using *sp*. When needed, we can convert *sf* objects to corresponding *sp* objects or vice versa. Using sf::st_as_sf and sf::as_Spatial, sp objects and sf objects can be converted to each other.

```
# convert sf to sp
class(manhattan_noise_sf)
manhattan_noise_sp_ <- sf::as_Spatial(manhattan_noise_sf)
class(manhattan_noise_sp_)

# Convert sp to sf
manhattan_noise_sf_ <- sf::st_as_sf(manhattan_noise_sp_)
class(manhattan_noise_sf_)
```

1.3.3 Save *sf* objects

We will save this object as a Shapefile on our hard drive for later use. (Note that by default `st_write` checks if the file already exists, and if so it will not overwrite it. If you need to force it to overwrite use the option `delete_layer = TRUE`.)

```
#
st_write(manhattan_noise_sf, "data/nyc", "ManhattanNoise", driver = "ESRI Shapefile")
```

```
# to force the save:
st_write(manhattan_noise_sf, "data/nyc/ManhattanNoise.shp", delete_layer = TRUE)

#
st_write(man_day_noise_sf, './data/nyc/manhattan_day_noise.shp')
```

Shapefile is an outdated file format and has poor support for very large integers and datetime data types, for example. To completely save the data in R format, we can also directly save them as R data files. Of course, these files can only be used by R. Alternatively, we can use database-compatible format like GeoPackage, which is an OGC standard and increasingly popular, particular with QGIS.

```
# Save data to RData file
save(manhattan_noise_sf, man_day_noise_sf,
     file = './data/nyc/manhattan_noise_sf.RData')

# To get the data back into R
# load(file='./data/nyc/manhattan_noise_sf.RData')

# Save data to a GeoPackage file/database
st_write(manhattan_noise_sf,
         dsn = './data/nyc/man_data.gpkg',
         layer='manhattan_noise',
         delete_layer = TRUE)

st_write(man_day_noise_sf,
         dsn = './data/nyc/man_data.gpkg',
         layer='manhattan_noise_one_day',
         delete_layer = TRUE)

# Read in from geopackage
man_one_day_noise <- st_read(dsn = './data/nyc/man_data.gpkg',
        layer='manhattan_noise_one_day')
```

1.4 Reprojecting or Projection Transformation

So far, all our data are using geographic coordinates in degrees. Sometimes, we may have to change the coordinates into a new Coordinate Reference System (CRS). For example, we retrieved census tracts data for the entire country, which are normally using geographic coordinates. For a local application at NYC, we need to reproject the data to a CRS that minimizes the distortion locally. With a local map projection, we can more accurately measures distances, areas, and angles.

Functions to *transform*, or *reproject* spatial objects typically take the following two arguments:

- the spatial object to transform
- a CRS object with the new projection definition

You can transform

- a *sf* object with `st_transform()`

- a `Spatial*` object with `spTransform()`
- a `raster` object with `projectRaster()`

The perhaps trickiest part here is to determine the definition of the projection, which needs to be a character string in proj4 format or a number for SRID (spatial reference ID). You can look it up online. For example for UTM zone 33N (EPSG:32633), its SRID is 32633 and its proj4 string would be:

```
+proj=utm +zone=33 +ellps=WGS84 +datum=WGS84 +units=m +no_defs
```

You can retrieve the CRS:

- from an *sf* object with `st_crs()`
- from an existing `Spatial*` object with `proj4string()`
- from a `raster` object with `crs()`

1.4.1 Transform or reproject *sf* objects

Let us check `nyc_sf_merged` object and reproject it to a local projection in State Plane Long Island, that is the map projection the NYC.

```
st_crs(man_day_noise_sf)
```

```
#> Coordinate Reference System:
#>   User input: EPSG:4326
#>   wkt:
#> GEOGCRS["WGS 84",
#>     DATUM["World Geodetic System 1984",
#>         ELLIPSOID["WGS 84",6378137,298.257223563,
#>             LENGTHUNIT["metre",1]]],
#>     PRIMEM["Greenwich",0,
#>         ANGLEUNIT["degree",0.0174532925199433]],
#>     CS[ellipsoidal,2],
#>         AXIS["geodetic latitude (Lat)",north,
#>             ORDER[1],
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>         AXIS["geodetic longitude (Lon)",east,
#>             ORDER[2],
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>     USAGE[
#>         SCOPE["Horizontal component of 3D system."],
#>         AREA["World."],
#>         BBOX[-90,-180,90,180]],
#>     ID["EPSG",4326]]
```

```
# then we transform/reproject it to SPCS Long Island, 2831
man_day_noise_2831 <- st_transform(man_day_noise_sf, 2831)

st_crs(man_day_noise_2831)
```

```
#> Coordinate Reference System:
#>   User input: EPSG:2831
#>   wkt:
```

```
#> PROJCRS["NAD83(HARN) / New York Long Island",
#>     BASEGEOGCRS["NAD83(HARN)",
#>         DATUM["NAD83 (High Accuracy Reference Network)",
#>             ELLIPSOID["GRS 1980",6378137,298.257222101,
#>                 LENGTHUNIT["metre",1]]],
#>         PRIMEM["Greenwich",0,
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>         ID["EPSG",4152]],
#>     CONVERSION["SPCS83 New York Long Island zone (meters)",
#>         METHOD["Lambert Conic Conformal (2SP)",
#>             ID["EPSG",9802]],
#>         PARAMETER["Latitude of false origin",40.1666666666667,
#>             ANGLEUNIT["degree",0.0174532925199433],
#>             ID["EPSG",8821]],
#>         PARAMETER["Longitude of false origin",-74,
#>             ANGLEUNIT["degree",0.0174532925199433],
#>             ID["EPSG",8822]],
#>         PARAMETER["Latitude of 1st standard parallel",41.0333333333333,
#>             ANGLEUNIT["degree",0.0174532925199433],
#>             ID["EPSG",8823]],
#>         PARAMETER["Latitude of 2nd standard parallel",40.6666666666667,
#>             ANGLEUNIT["degree",0.0174532925199433],
#>             ID["EPSG",8824]],
#>         PARAMETER["Easting at false origin",300000,
#>             LENGTHUNIT["metre",1],
#>             ID["EPSG",8826]],
#>         PARAMETER["Northing at false origin",0,
#>             LENGTHUNIT["metre",1],
#>             ID["EPSG",8827]]],
#>     CS[Cartesian,2],
#>         AXIS["easting (X)",east,
#>             ORDER[1],
#>             LENGTHUNIT["metre",1]],
#>         AXIS["northing (Y)",north,
#>             ORDER[2],
#>             LENGTHUNIT["metre",1]],
#>     USAGE[
#>         SCOPE["Engineering survey, topographic mapping."],
#>         AREA["United States (USA) - New York - counties of Bronx; Kings; Nassau; New York; Que
#>         BBOX[40.47,-74.26,41.3,-71.8]],
#>     ID["EPSG",2831]]
```

We see that the CRS are different for the two. One is geographic coordinate systems (longitude, latitude) using WGS84 with a unit of decimal degree. And the other is New York State Plane Long Island with a unit of meter.

We can also see the proj4 strings of the two.

```
st_crs(man_day_noise_sf)$proj4string
st_crs(man_day_noise_2831)$proj4string
```

: we have `+proj=lcc...` and `+proj=longlat...`. LCC refers to Lambert Conic Conformal, which

is a projected coordinate system with numeric units.

We can use the `st_bbox()` method from the *sf* package to compare the coordinates before and after reprojection and confirm that we actually have transformed them. `st_bbox()` returns the *min* and *max* values of the two dimensions of a *sf* spatial object.

```
sf::st_bbox(man_day_noise_sf) # bounding box
```

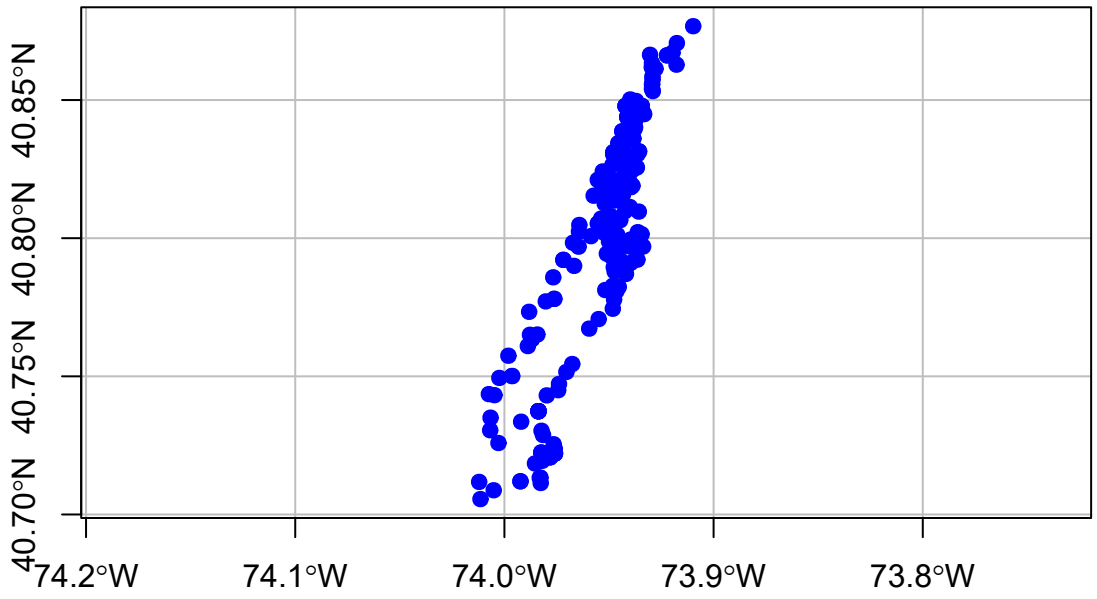
```
#>      xmin      ymin      xmax      ymax
#> -74.01208  40.70558 -73.90975  40.87673
```

```
st_bbox(man_day_noise_2831)
```

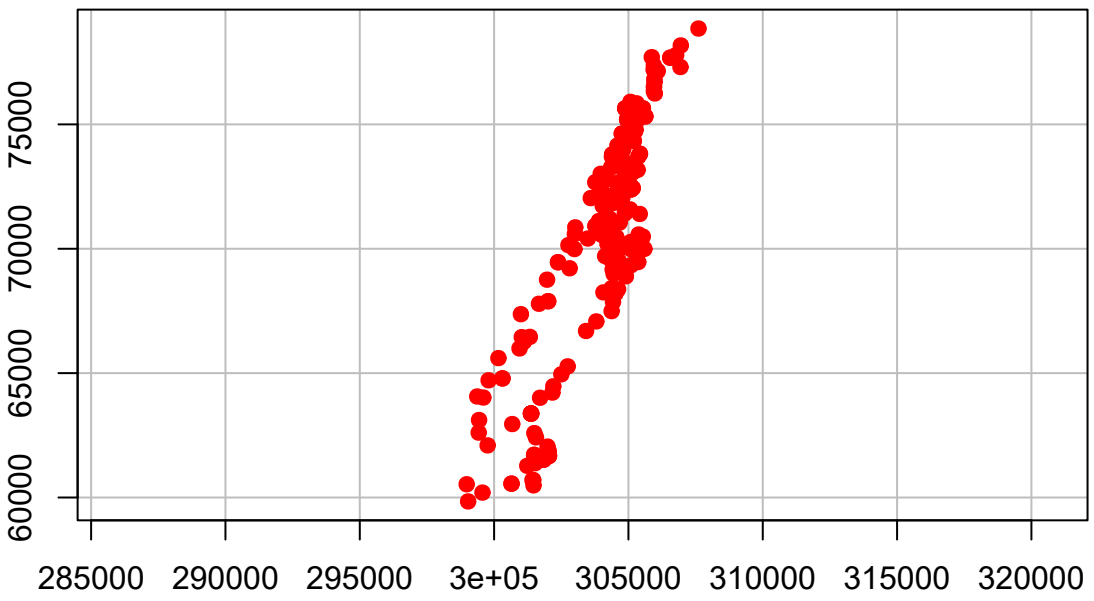
```
#>      xmin      ymin      xmax      ymax
#> 298979.03  59842.73 307607.32  78853.14
```

We can also visually compare the two map projections. While the differences are little bit hard to see, the locally projected map does stretch more along the horizontal direction.

```
#par(mfrow=c(1,2))
plot(man_day_noise_sf %>% st_geometry(),pch=19,col='blue', axes=TRUE,xaxs='r', graticule=st_
```




```
plot(man_day_noise_2831 %>% st_geometry(),pch=19,col='red', axes=TRUE,xaxs='i', graticule=st
```



We will see later that using a local map projection with a unit of meter is necessary for spatial operations like buffer or other distance-based analyses.

Set CRS and Reproject (transform)

- Assigning a new CRS to a spatial object does not change its coordinates but the CRS determines how the software understand the coordinates. So, we must choose and set a CRS that is consistent with the coordinates. Otherwise, the coordinates will be misinterpreted.
- Reprojecting spatial data with st_transform and gTransform really changes the underlying coordinates, which we can see from the bbox values.

- In either case, the CRS must be consistent with the coordinates. It is a good practice to check projected spatial data against a basemap or a correctly referenced map to verify its CRS.
- SpatialReference.org is a very good source for CRS information.
- Some commonly used reference systems
  - US 48 Contiguous States: Albers Equal Area (ESRI:102003), Lambert Conformal (ESRI:102004)
  - New York State: UTM 18N (EPSG 3725, 3748, 26918)
  - New York City: State Plane Long Island (EPSG 2263, 2831, 3627)

1.5 Raster data in R

Raster files, as you might know, have a much more compact data structure than vectors. Because of their regular structure the coordinates do not need to be recorded for each pixel or cell in the rectangular extent. A raster is defined by:

- a CRS
- coordinates of its origin
- a distance or cell size in each direction
- a dimension or numbers of cells in each direction
- an array of cell values

Given this structure, coordinates for any cell can be computed and don't need to be stored.

The `raster` package[2] is a major extension of spatial data classes to access large rasters and in particular to process very large files. It includes object classes for `RasterLayer`, `RasterStacks`, and `RasterBricks`, functions for converting among these classes, and operators for computations on the raster data. Conversion from *sp* type objects into `raster` type objects is possible.

If we wanted to do create a raster object from scratch we would do the following:

```
# specify the RasterLayer with the following parameters:
# - minimum x coordinate (left border)
# - minimum y coordinate (bottom border)
# - maximum x coordinate (right border)
# - maximum y coordinate (top border)
# - resolution (cell size) in each dimension
r <- raster(xmn=-0.5, ymn=-0.5, xmx=4.5, ymx=4.5, resolution=c(1,1))
r
```

```
#> class      : RasterLayer
#> dimensions : 5, 5, 25  (nrow, ncol, ncell)
#> resolution : 1, 1  (x, y)
#> extent     : -0.5, 4.5, -0.5, 4.5  (xmin, xmax, ymin, ymax)
#> crs        : +proj=longlat +datum=WGS84 +no_defs
```

Note that this raster object **has a CRS defined!** If the crs argument is missing when creating the Raster object, the x coordinates are within -360 and 360 and the y coordinates are within -90 and 90, the WGS84 projection is used by default!

Good to know.

To add some values to the cells we could the following.

```
class(r)
```

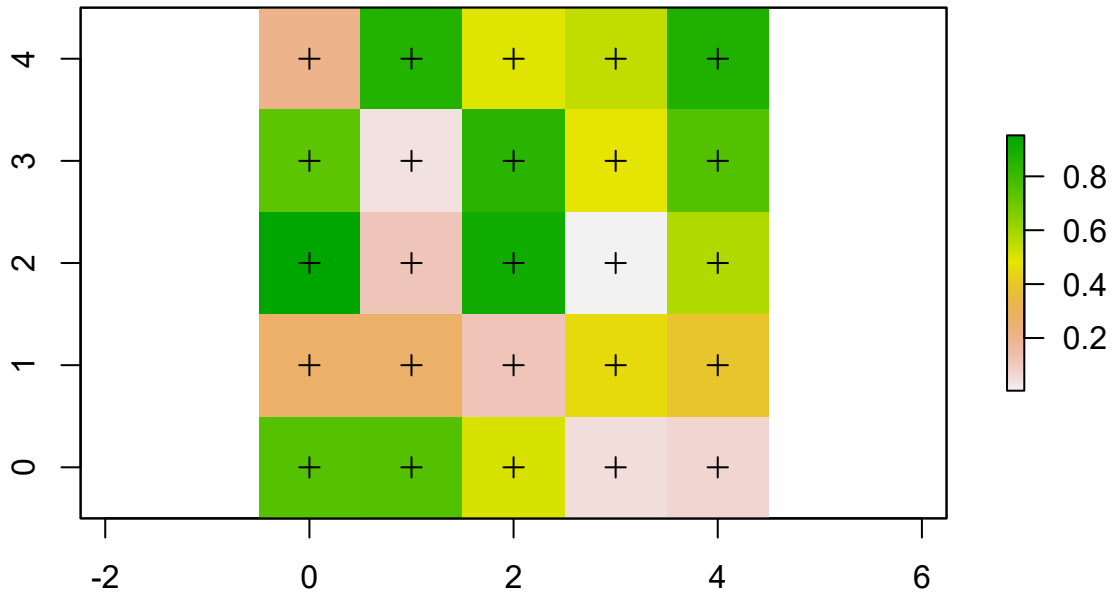
```
#> [1] "RasterLayer"
#> attr(,"package")
#> [1] "raster"
```

```
r <- setValues(r, runif(25))
class(r)
```

```
#> [1] "RasterLayer"
#> attr(,"package")
#> [1] "raster"
```

[2]Note that *sp* also allows to work with raster structures. The `GridTopology` class is the key element of raster representations. It contains: (a) the center coordinate pair of the south-west raster cell, (b) the two cell sizes in the metric of the coordinates, giving the step to successive centers, and (c) the numbers of cells for each dimension. There is also a `SpatialPixels` object which stores grid topology and coordinates of the actual points.

```
plot(r); points(coordinates(r), pch=3)
```



(See the `rasterVis` package for more advanced plotting of `Raster*` objects.)

RasterLayer objects can also be created from a matrix.

```
class(volcano)
```

```
#> [1] "matrix" "array"
```

```
volcano.r <- raster(volcano)
class(volcano.r)
```

```
#> [1] "RasterLayer"
#> attr(,"package")
#> [1] "raster"
```

And to read in a raster file we can use the `raster()` function. This raster is generated as part of the NEON Harvard Forest field site.

```
library(raster)
HARV <- raster("data/HARV_RGB_Ortho.tif")
```

Typing the name of the object will give us what's in there:

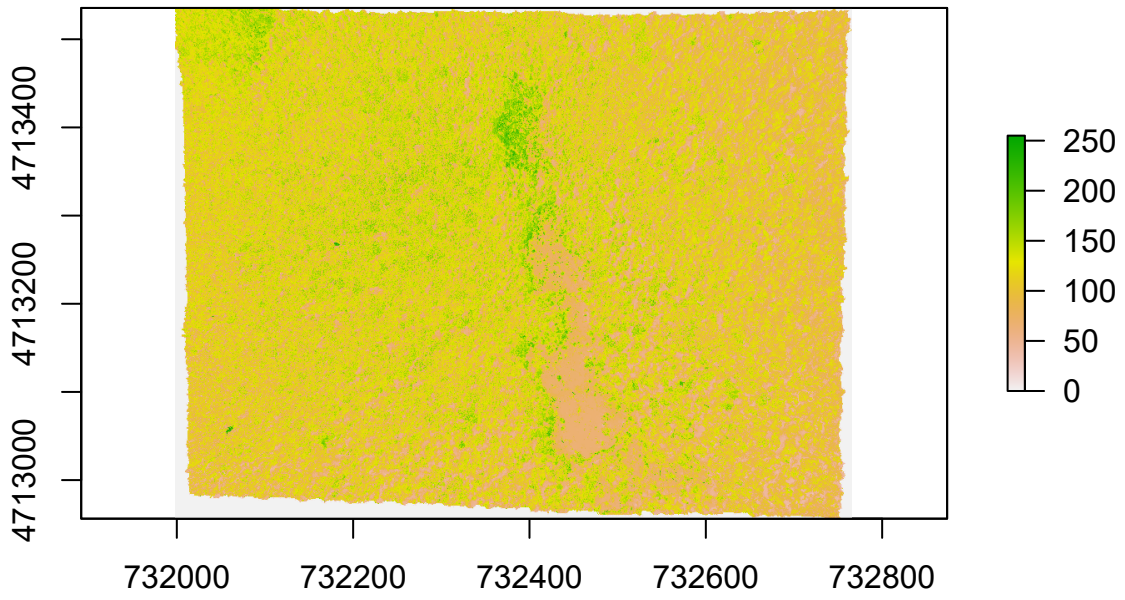
```
HARV
```

```
#> class      : RasterLayer
#> band       : 1  (of  3  bands)
#> dimensions : 2317, 3073, 7120141  (nrow, ncol, ncell)
```

```
#> resolution : 0.25, 0.25  (x, y)
#> extent     : 731998.5, 732766.8, 4712956, 4713536  (xmin, xmax, ymin, ymax)
#> crs        : +proj=utm +zone=18 +datum=WGS84 +units=m +no_defs
#> source     : HARV_RGB_Ortho.tif
#> names      : HARV_RGB_Ortho
#> values     : 0, 255  (min, max)
```

We can plot it like this:

```
plot(HARV)
```



We can find out about the Coordinate Reference System with this:

```
crs(HARV)
```

```
#> Coordinate Reference System:
#> Deprecated Proj.4 representation:
#>  +proj=utm +zone=18 +datum=WGS84 +units=m +no_defs
#> WKT2 2019 representation:
#> PROJCRS["WGS 84 / UTM zone 18N",
#>     BASEGEOGCRS["WGS 84",
#>         DATUM["World Geodetic System 1984",
#>             ELLIPSOID["WGS 84",6378137,298.257223563,
#>                 LENGTHUNIT["metre",1]]],
#>         PRIMEM["Greenwich",0,
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>         ID["EPSG",4326]],
```

```
#>     CONVERSION["UTM zone 18N",
#>         METHOD["Transverse Mercator",
#>             ID["EPSG",9807]],
#>         PARAMETER["Latitude of natural origin",0,
#>             ANGLEUNIT["degree",0.0174532925199433],
#>             ID["EPSG",8801]],
#>         PARAMETER["Longitude of natural origin",-75,
#>             ANGLEUNIT["degree",0.0174532925199433],
#>             ID["EPSG",8802]],
#>         PARAMETER["Scale factor at natural origin",0.9996,
#>             SCALEUNIT["unity",1],
#>             ID["EPSG",8805]],
#>         PARAMETER["False easting",500000,
#>             LENGTHUNIT["metre",1],
#>             ID["EPSG",8806]],
#>         PARAMETER["False northing",0,
#>             LENGTHUNIT["metre",1],
#>             ID["EPSG",8807]]],
#>     CS[Cartesian,2],
#>         AXIS["(E)",east,
#>             ORDER[1],
#>             LENGTHUNIT["metre",1]],
#>         AXIS["(N)",north,
#>             ORDER[2],
#>             LENGTHUNIT["metre",1]],
#>     USAGE[
#>         SCOPE["Engineering survey, topographic mapping."],
#>         AREA["Between 78Â°W and 72Â°W, northern hemisphere between equator and 84Â°N, onshore
#>         BBOX[0,-78,84,-72]],
#>     ID["EPSG",32618]]
```

See what you can do with such an object:

```
methods(class=class(HARV))
```

```
#>   [1] !                     !=                    $
#>   [4] $<-                   %in%                  [
#>   [7] [[                    [[<-                  [<-
#>  [10] ==                    addLayer              adjacent
#>  [13] aggregate             all.equal             area
#>  [16] Arith                 as.array              as.character
#>  [19] as.data.frame         as.factor             as.integer
#>  [22] as.list               as.logical            as.matrix
#>  [25] as.raster             as.vector             asFactor
#>  [28] atan2                 bandnr                barplot
#>  [31] bbox                  boundaries            boxplot
#>  [34] brick                 buffer                calc
#>  [37] cellFromRowCol        cellFromRowColCombine cellFromXY
#>  [40] cellStats             clamp                 click
#>  [43] clump                 coerce                colFromCell
#>  [46] colFromX              colSums               Compare
```

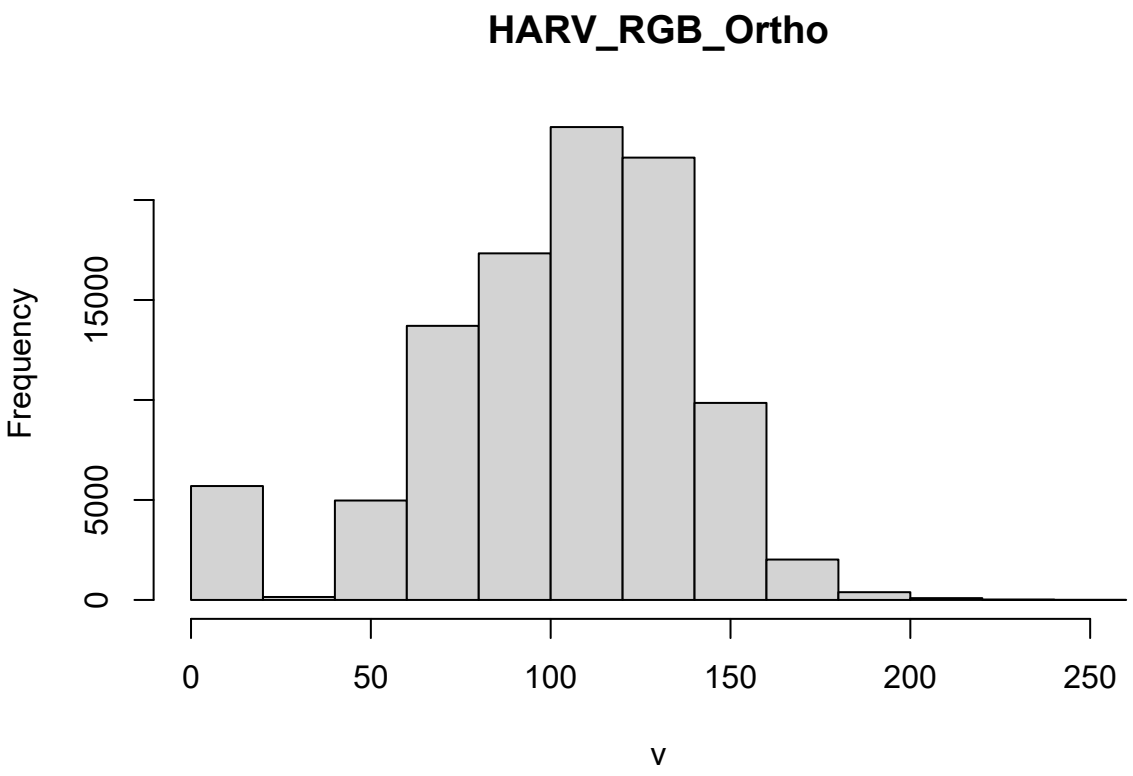
```
#>  [49] contour              coordinates          corLocal
#>  [52] couldBeLonLat        cover                crop
#>  [55] crosstab             crs<-                cut
#>  [58] cv                   density              dim
#>  [61] dim<-                direction            disaggregate
#>  [64] distance             extend               extent
#>  [67] extract              flip                 focal
#>  [70] freq                 getValues            getValuesBlock
#>  [73] getValuesFocal       gridDistance         hasValues
#>  [76] head                 hist                 image
#>  [79] init                 inMemory             interpolate
#>  [82] intersect            is.factor            is.finite
#>  [85] is.infinite          is.na                is.nan
#>  [88] isLonLat             KML                  labels
#>  [91] layerize             length               levels
#>  [94] levels<-             lines                localFun
#>  [97] log                  Logic                mapView
#> [100] mask                 match                Math
#> [103] Math2                maxValue             mean
#> [106] merge                metadata             minValue
#> [109] modal                mosaic               names
#> [112] names<-              ncell                ncol
#> [115] ncol<-               nlayers              nrow
#> [118] nrow<-               origin               origin<-
#> [121] overlay              persp                plot
#> [124] predict              print                proj4string
#> [127] proj4string<-        quantile             raster
#> [130] rasterize            ratify               readAll
#> [133] readStart            readStop             reclassify
#> [136] rectify              res                  res<-
#> [139] resample             RGB                  rotate
#> [142] rowColFromCell       rowFromCell          rowFromY
#> [145] rowSums              sampleRandom         sampleRegular
#> [148] sampleStratified     scale                select
#> [151] setMinMax            setValues            shift
#> [154] show                 spplot               stack
#> [157] stackSelect          stretch              subs
#> [160] subset               Summary              summary
#> [163] t                    tail                 terrain
#> [166] text                 trim                 unique
#> [169] update               values               values<-
#> [172] Which                which.max            which.min
#> [175] wkt                  writeRaster          writeStart
#> [178] writeStop            writeValues          xFromCell
#> [181] xFromCol             xmax                 xmax<-
#> [184] xmin                 xmin<-               xres
#> [187] xyFromCell           yFromCell            yFromRow
#> [190] ymax                 ymax<-               ymin
#> [193] ymin<-               yres                 zonal
#> [196] zoom
```

```
#> see '?methods' for accessing help and source code
```

We can explore the distribution of values contained within our raster using the hist() function which produces a histogram. Histograms are often useful in identifying outliers and bad data values in our raster data.

```
hist(HARV)
```

```
#> Warning in .hist1(x, maxpixels = maxpixels, main = main, plot = plot, ...): 1%
#> of the raster cells were used. 100000 values used.
```



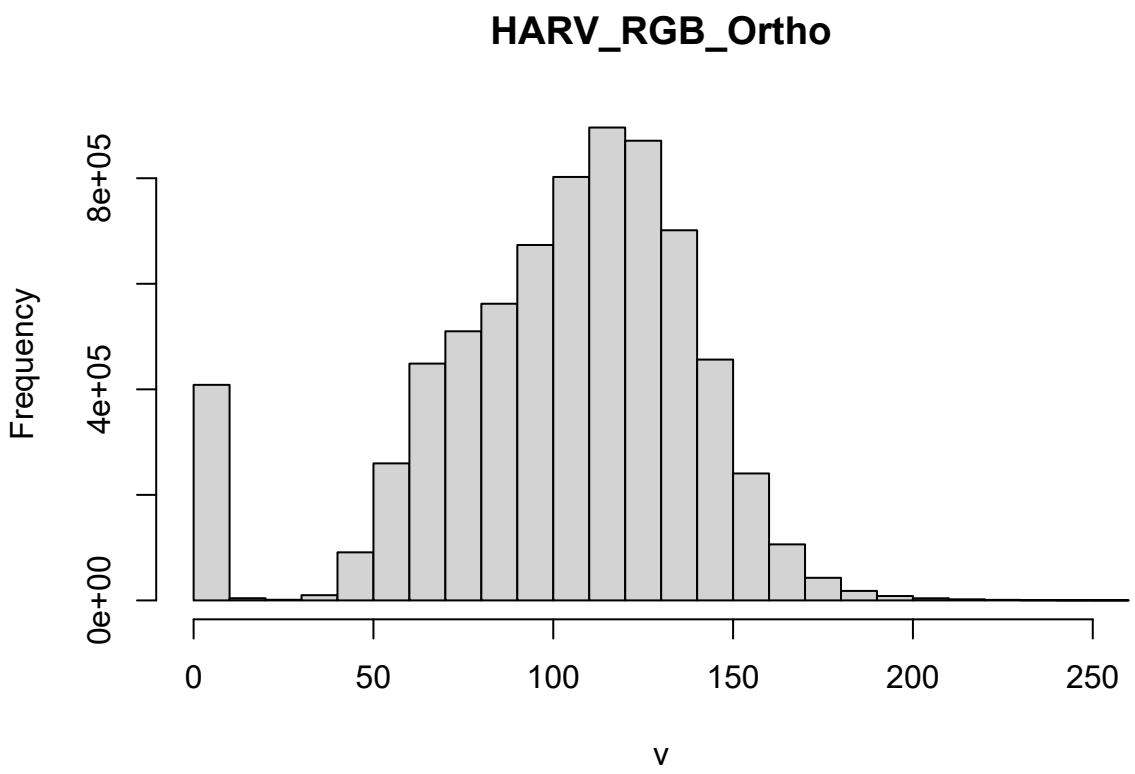
Notice that a warning message is produced when R creates the histogram.

This warning is caused by the default maximum pixels value of 100,000 associated with the hist function. This maximum value is to ensure processing efficiency as our data become larger! We can force the `hist` function to use all cell values.

```
ncell(HARV)
```

```
#> [1] 7120141
```

```
hist(HARV, maxpixels = ncell(HARV))
```



At times it may be useful to explore raster metadata before loading them into R. This can be done with:

```
GDALinfo("path-to-raster-here")
```

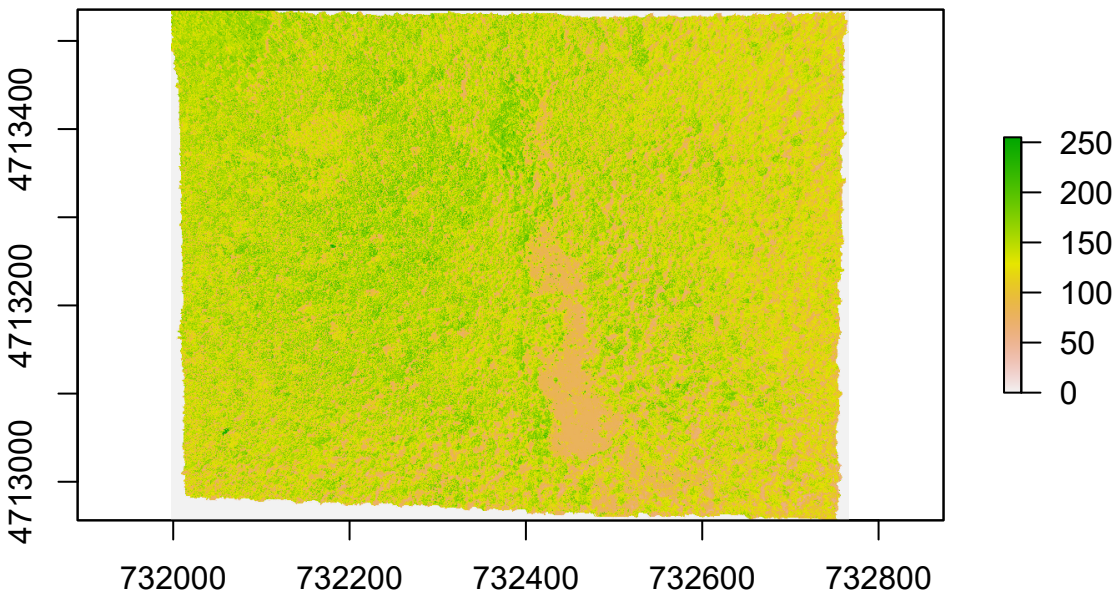
A raster dataset can contain one or more bands. We can view the number of bands in a raster using the `nlayers()` function.

```
nlayers(HARV)
```

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

We can use the `raster()` function to import one single band from a *single* **OR** from a *multi-band* raster. For multi-band raster, we can specify which band we want to read in.

```
HARV_Band2 <- raster("data/HARV_RGB_Ortho.tif", band = 2)
plot(HARV_Band2)
```



To bring in all bands of a multi-band raster, we use the `stack()` function.

```
HARV_stack <- stack("data/HARV_RGB_Ortho.tif")

# how many layers?
nlayers(HARV_stack)
```

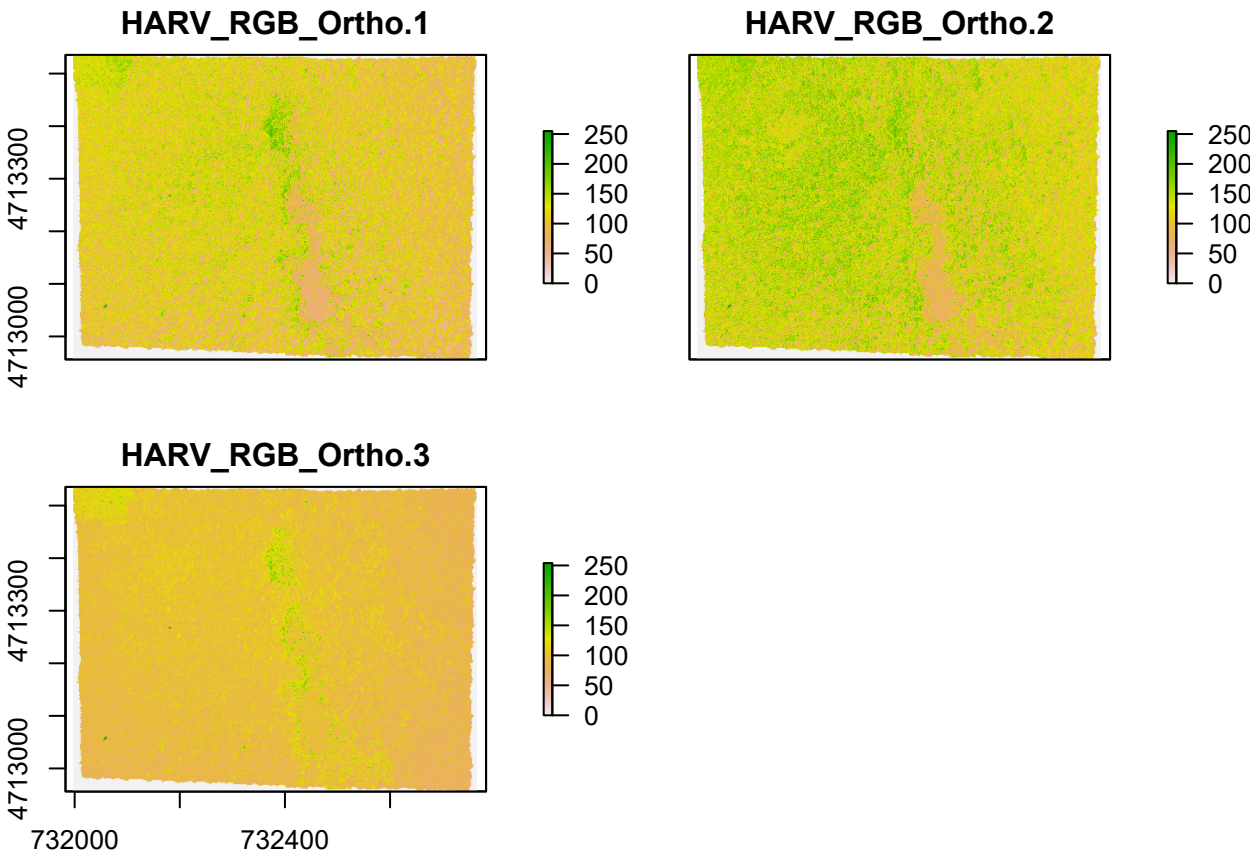
```
#> [1] 3
```

```
# view attributes of stack object
HARV_stack
```

```
#> class      : RasterStack
#> dimensions : 2317, 3073, 7120141, 3  (nrow, ncol, ncell, nlayers)
#> resolution : 0.25, 0.25  (x, y)
#> extent     : 731998.5, 732766.8, 4712956, 4713536  (xmin, xmax, ymin, ymax)
#> crs        : +proj=utm +zone=18 +datum=WGS84 +units=m +no_defs
#> names      : HARV_RGB_Ortho.1, HARV_RGB_Ortho.2, HARV_RGB_Ortho.3
#> min values :                 0,                 0,                 0
#> max values :               255,               255,               255
```

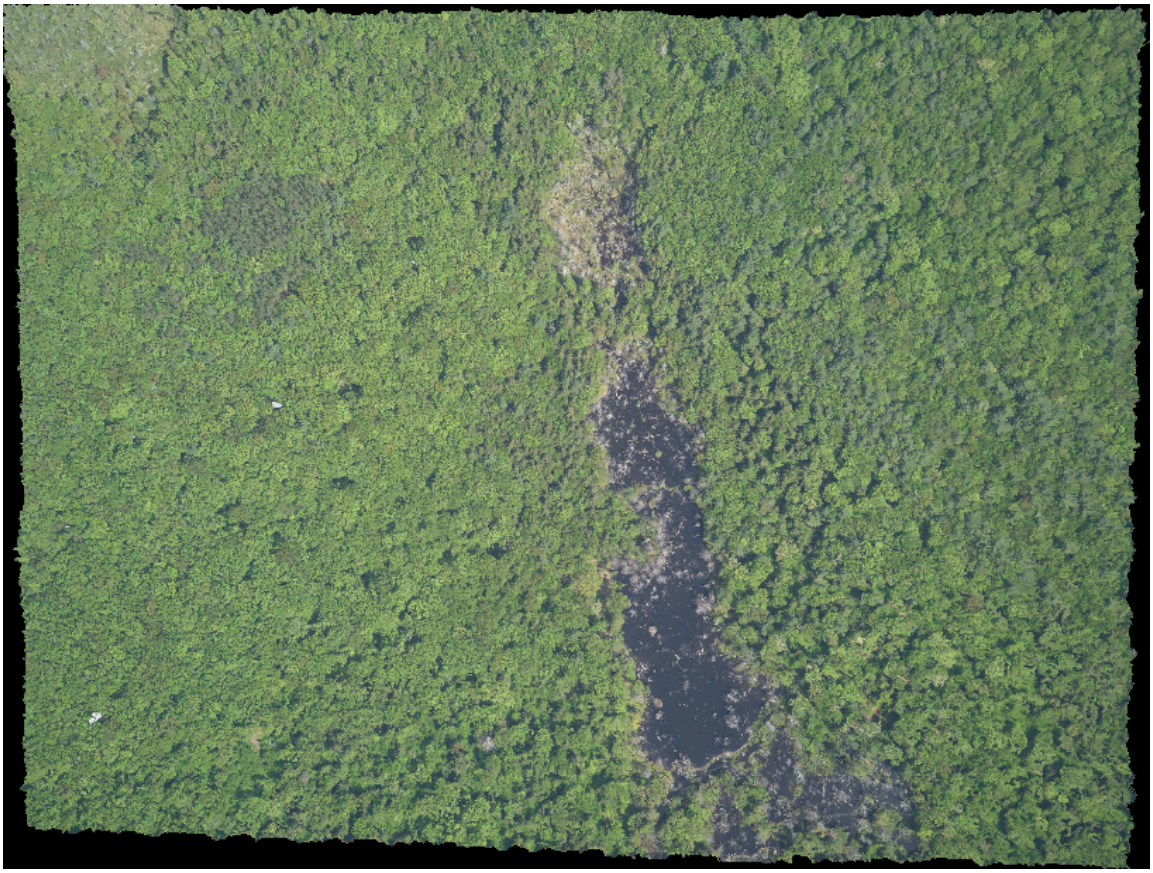
What happens when we plot?

```
plot(HARV_stack)
```



If we know that it is an RGB multiband raster we can plot them all in one

```
plotRGB(HARV_stack)
```



1.5.1 RasterStack vs RasterBrick

The R `RasterStack` and `RasterBrick` object types can both store multiple bands. However, how they store each band is different. The bands in a `RasterStack` are stored as links to raster data that is located somewhere on our computer. A `RasterBrick` contains all of the objects stored within the actual R object. Since in the `RasterBrick`, all of the bands are stored within the actual object its object size is much larger than the `RasterStack` object.

In most cases, we can work with a `RasterBrick` in the same way we might work with a `RasterStack`. However, a `RasterBrick` is often more efficient and faster to process - which is important when working with larger files.

We can turn a `RasterStack` into a `RasterBrick` in R by using `brick(StackName)`. Use the `object.size()` function to compare stack and brick R objects.

```
object.size(HARV_stack)
```

```
#> 51224 bytes
```

```
HARV_brick <- brick(HARV_stack)
object.size(HARV_brick)
```

```
#> 170898912 bytes
```

Going back to the *sp* package, a simple grid can be built like this:

```
# specify the grid topology with the following parameters:
# - the smallest coordinates for each dimension, here: 0,0
# - cell size in each dimension, here: 1,1
```

```
# - number of cells in each dimension, here: 5,5
gtopo <- GridTopology(c(0,0), c(1,1), c(5,5)) # create the grid
datafr <- data.frame(runif(25)) # make up some data
SpGdf <- SpatialGridDataFrame(gtopo, datafr) # create the grid data frame
summary(SpGdf)
```

```
#> Object of class SpatialGridDataFrame
#> Coordinates:
#>       min max
#> [1,] -0.5 4.5
#> [2,] -0.5 4.5
#> Is projected: NA
#> proj4string : [NA]
#> Grid attributes:
#>   cellcentre.offset cellsize cells.dim
#> 1                 0        1         5
#> 2                 0        1         5
#> Data attributes:
#>    runif.25.
#>  Min.   :0.0982
#>  1st Qu.:0.4461
#>  Median :0.6745
#>  Mean   :0.6100
#>  3rd Qu.:0.7878
#>  Max.   :0.9915
```

1.6 Lab Assignment

For the R-Spatial Section labs, we will do some spatial data visualization on COVID-19 in NYC. More specifically, we will explore the distribution of confirmed cases across the city and their relationships with some demographic variables and essential services related to retail food stores and public health services.

The first lab is rather simple and straightforward but will be the foundation for the next steps.

Tasks for the first lab are:

1. Set up a R project for the R-Spatial section.
2. Read the NYC postal areas in Shapefiles into *sf* objects. As NYC DOH publishes COVID-19 data by zip code, we will utilize the postal area data later.
3. Read and process the NYC public health services spreadsheet data. Create *sf* objects from geographic coordinates.
4. Read and process the NYS retail food stores data. Create *sf* objects from geographic coordinates for NYC.
5. Use simple mapping method, either based on `ggmap`+`ggplot` or *mapview*, with a basemap to verify the above datasets in terms of their geometry locations.
6. Save the three *sf* objects in a RData file or in a single GeoPackage file/database.

The assignment and data are available on Blackboard. The data are also available for download at the Dropbox site.

Chapter 2

Spatial data manipulation in R

Learning Objectives

- Join attribute data to a polygon vector file
- Select data using attribute and spatial relationships
- Reproject spatial data
- Aggregate spatial data using spatial relationships

There are a wide variety of spatial, topological, and attribute data operations we can perform with R. Lovelace et al's recent publication[1] goes into great depth about this and is highly recommended.

In this section we will look at just a few examples for libraries and functions that allow us to process spatial data in R and perform some commonly used operations.

2.1 Attribute Join

An attribute join on vector data brings tabular data into a geographic context. It refers to the process of joining data in tabular format to data in a format that holds the geometries (polygon, line, or point)[2].

If you have done attribute joins of shapefiles in GIS software like *ArcGIS* or *QGIS* you know that you need a **unique identifier** in both the attribute table of the Shapefile and the table to be joined.

First we will load the CSV table `nyc_census_tracts_pophu.csv` into a tibble dataframe in R tidyverse and name it `nyc_pophu` for population and housing unit information.

```
nyc_pophu <- readr::read_csv("data/nyc/nyc_census_tracts_pophu.csv", lazy = FALSE)
str(nyc_pophu)
```

```
#> spec_tbl_df [2,164 x 12] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
#>  $ GEOID10   : num [1:2164] 3.6e+10 3.6e+10 3.6e+10 3.6e+10 3.6e+10 ...
#>  $ STATEFP   : num [1:2164] 36 36 36 36 36 36 36 36 36 36 ...
#>  $ COUNTYFP  : chr [1:2164] "005" "005" "005" "005" ...
#>  $ TRACTCE   : chr [1:2164] "002300" "002701" "004100" "004800" ...
```

[1] Lovelace, R., Nowosad, J., & Muenchow, J. (2019). Geocomputation with R. CRC Press.

[2] Per the ESRI specification a Shapefile must have an attribute table, so when we read it into R with the `readOGR` command from the *sp* package it automatically becomes a `Spatial*Dataframe` and the attribute table becomes the data.frame.

```
#>  $ AFFGEOID  : chr [1:2164] "1400000US36005002300" "1400000US36005002701" "1400000US3600500410
#>  $ POPULATION: num [1:2164] 4774 3016 6476 3999 7421 ...
#>  $ MEDAGE    : num [1:2164] 29 29 23 30 28 28 29 44 38 39 ...
#>  $ MEDHHINC  : num [1:2164] 14479 20153 18636 30301 21016 ...
#>  $ MEDHOUSING: num [1:2164] 406 627 655 1066 753 ...
#>  $ PCPREWAR  : num [1:2164] 26.8 40.6 31.3 51.1 37.3 55.5 52 19.7 18.9 18.7 ...
#>  $ MEDYRBUILT: num [1:2164] 1957 1952 1955 1942 1950 ...
#>  $ DENSITY   : num [1:2164] 118158 94522 90795 68668 90478 ...
#>  - attr(*, "spec")=
#>   .. cols(
#>   ..   GEOID10 = col_double(),
#>   ..   STATEFP = col_double(),
#>   ..   COUNTYFP = col_character(),
#>   ..   TRACTCE = col_character(),
#>   ..   AFFGEOID = col_character(),
#>   ..   POPULATION = col_double(),
#>   ..   MEDAGE = col_double(),
#>   ..   MEDHHINC = col_double(),
#>   ..   MEDHOUSING = col_double(),
#>   ..   PCPREWAR = col_double(),
#>   ..   MEDYRBUILT = col_double(),
#>   ..   DENSITY = col_double()
#>   .. )
#>  - attr(*, "problems")=<externalptr>
```

2.1.1 How to do this in *sf*

Now let's read the spatial data from a Shapefile into a *sf* object. Check out the column names of `nyc_census_tracts_sf` and of `nyc_pophu` to determine which one might contain the unique identifier for the join.

```
nyc_census_tracts_sf <- st_read('data/nyc/nyc_census_tracts.shp')
```

```
#> Reading layer `nyc_census_tracts' from data source
#>   `D:\Cloud_Drive\Dropbox (Hunter College)\Workspace\RSpace\R-Spatial_Book\data\nyc\nyc_census
#>   using driver `ESRI Shapefile'
#> Simple feature collection with 2164 features and 10 fields
#> Geometry type: MULTIPOLYGON
#> Dimension:     XY
#> Bounding box:  xmin: -74.25609 ymin: 40.50203 xmax: -73.70002 ymax: 40.91758
#> Geodetic CRS:  NAD83
```

```
str(nyc_census_tracts_sf)
```

```
#> Classes 'sf' and 'data.frame':   2164 obs. of  11 variables:
#>  $ GEOID   : chr  "36005002300" "36005002701" "36005004100" "36005004800" ...
#>  $ STATEFP : chr  "36" "36" "36" "36" ...
#>  $ COUNTYFP: chr  "005" "005" "005" "005" ...
#>  $ TRACTCE : chr  "002300" "002701" "004100" "004800" ...
#>  $ AFFGEOID: chr  "1400000US36005002300" "1400000US36005002701" "1400000US36005004100" "140000
#>  $ NAME    : chr  "23" "27.01" "41" "48" ...
#>  $ LSAD    : chr  "CT" "CT" "CT" "CT" ...
```

```
#>  $ ALAND   : num  104645 82641 184733 150832 212430 ...
#>  $ AWATER  : num  0 0 0 0 0 0 0 0 0 0 ...
#>  $ CBSA    : chr  "New York-Newark-Jersey City, NY-NJ-PA" "New York-Newark-Jersey City, N
#>  $ geometry:sfc_MULTIPOLYGON of length 2164; first list element: List of 1
#>   ..$ :List of 1
#>   .. ..$ : num [1:11, 1:2] -73.9 -73.9 -73.9 -73.9 -73.9 ...
#>   ..- attr(*, "class")= chr [1:3] "XY" "MULTIPOLYGON" "sfg"
#>  - attr(*, "sf_column")= chr "geometry"
#>  - attr(*, "agr")= Factor w/ 3 levels "constant","aggregate",..: NA NA NA NA NA NA NA NA
#>   ..- attr(*, "names")= chr [1:10] "GEOID" "STATEFP" "COUNTYFP" "TRACTCE" ...
```

To join the nyc_pophu data frame with nyc_census_tracts_sf we can use merge in the base package like this:

```
nyc_sf_merged <- base::merge(nyc_census_tracts_sf, nyc_pophu, by.x = "GEOID", by.y = "GEOID1
names(nyc_sf_merged)
```

```
#>  [1] "GEOID"      "STATEFP.x"  "COUNTYFP.x" "TRACTCE.x"  "AFFGEOID.x"
#>  [6] "NAME"       "LSAD"       "ALAND"      "AWATER"     "CBSA"
#> [11] "STATEFP.y"  "COUNTYFP.y" "TRACTCE.y"  "AFFGEOID.y" "POPULATION"
#> [16] "MEDAGE"     "MEDHHINC"   "MEDHOUSING" "PCPREWAR"   "MEDYRBUILT"
#> [21] "DENSITY"    "geometry"
```

We see the new attribute columns added, as well as the geometry column.

```
# A basic plot to verify the data: use zcol to choose a subset of the data to plot
mapview(nyc_sf_merged, zcol=c('POPULATION','MEDHHINC'))
```

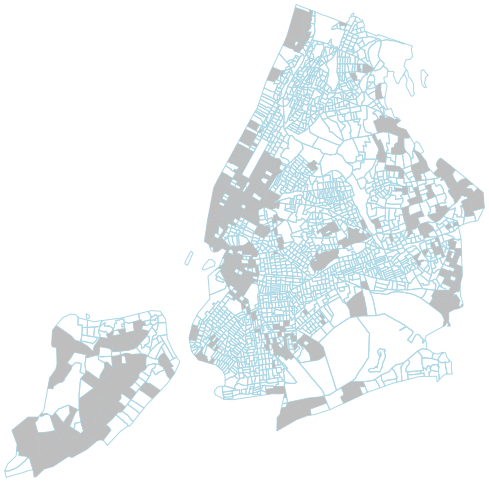
Alternatively, we can use tidyverse function join (left_join, right_join, inner_join etc) to conduct the task. Note that this requires the two matching columns being the same type.

```
dplyr::left_join(nyc_census_tracts_sf %>% dplyr::mutate(GEOID=as.numeric(as.character(GEOID)
                 nyc_pophu,
                 by = c('GEOID' = 'GEOID10')) -> nyc_sf_merged

names(nyc_sf_merged)
```

2.2 Attribute Selection (or non-spatial subsetting)

sf objects are also tidyverse compatible *data.frame*. Using ***dplyr::filter***, we can easy select a subset (rows) from ***sf*** data using tidyverse functions like dplyr::filter with logical expressions. The ***dplyr::select*** can select columns.

```
high_income_tracts <- nyc_sf_merged %>% dplyr::filter(MEDHHINC > 80000)
plot(nyc_sf_merged %>% st_geometry(), col='NA', border='lightblue', lwd=0.2)
high_income_tracts %>% st_geometry() %>% plot(col='grey', border='NA', add=TRUE)
```



2.3 Spatial Query

To perform spatial query, we must first understand spatial relationships in spatial analysis and GIScience. An excellent source for these relationships is GITTA Spatial Queries

The most useful spatial (or topological) relationships are illustrated below.

And these basic relations form a hierarchy. For example, the *intersects* relationship actually has a few subtypes like meet (or touch), overlap, and contain, etc. Anything that is not *disjoint* would be an *intersects*.

Naming rules for spatial functions

- ***sf***, as well as ST_SQL in PostGIS, uses verbs for functions of spatial relationships and use nouns for functions of spatial operations.
- Spatial relationship functions like st_overlaps, st_covers, and st_intersects examine the topological relationships between spatial objects.
- Of course, some functions named by prepositions and adjectives like st_within and st_equal are only meaningful for spatial relationships.
- Spatial relationship functions do not produce new geometries. They just tell use their relationships, with which we can conduct query, selection, and aggregation.
- By contrast, spatial operations will create new geometries. For example, st_intersection will produce new geometries in ***sf*** classes.

For the next example our goal is to select all census tracts that are adjacent to the central park.

Think about this for a moment – what might be the steps you'd follow?

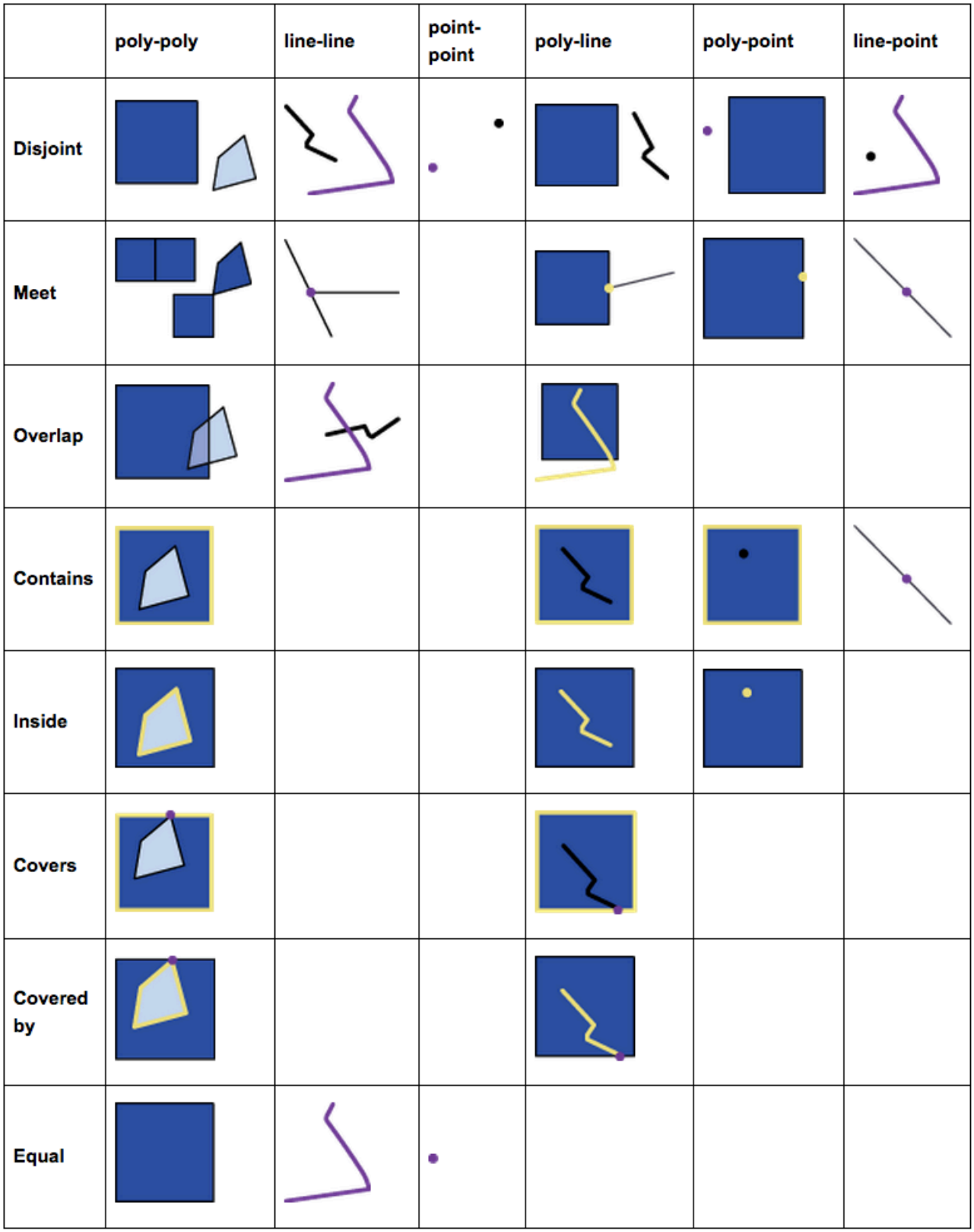


Figure 2.1: Spatial Relations

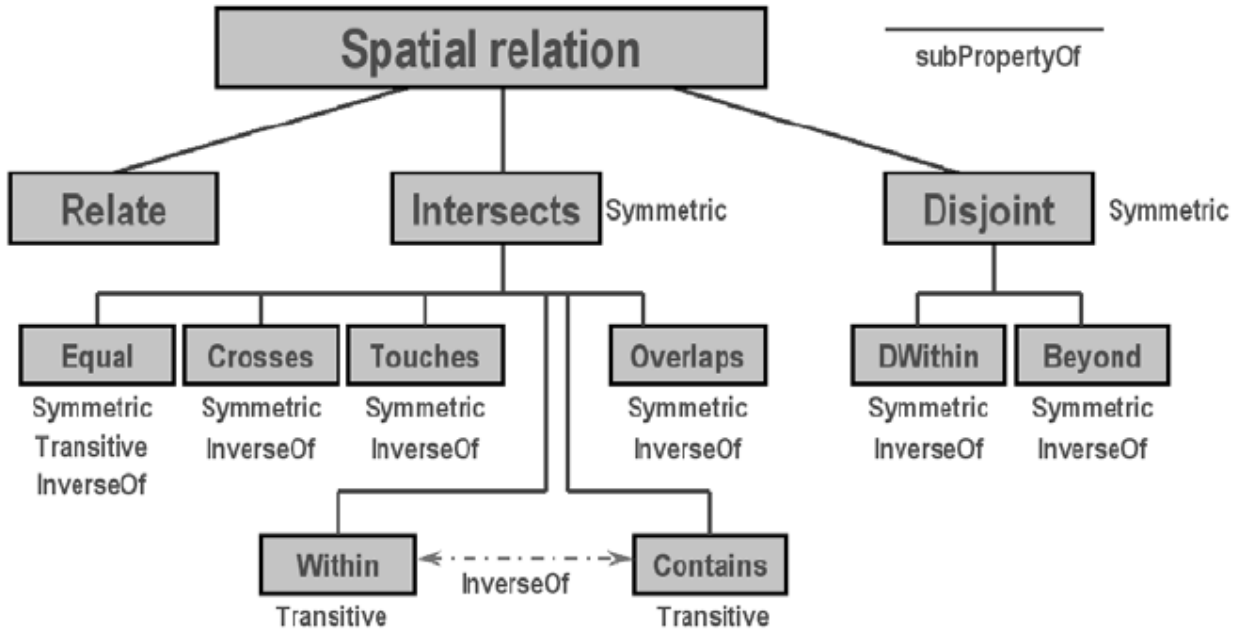


Figure 2.2: Hierarchy of Spatial Relations

```
## How about:

# 1. Get the census tract polygons.
# 2. Find the census tract that contains the central park.
# 3. Select all census tract polygons that intersect with or touch the central park tract
```

2.3.1 Using the *sf* package

We will use `nyc_sf_merged` for the census tract polygons. Using a desktop GIS or `mapview` package in R, we can easily find out the GEOID of the central park tract is 36061014300.

First, let's use non-spatial query to find the central park census tract.

```
# Note the GEOID column is a numeric type.
central_partk <- nyc_sf_merged %>% filter(GEOID == 36061014300)
```

Now we can use this `central_park` to select all census tract polygons that intersect with the tract. In order to determine the polygons we use `st_intersects`, a geometric binary which returns a vector of logical values, which we we can use for subsetting. Note the difference to `st_intersection`, which performs a geometric operation and creates a new ***sf*** object which cuts out the area of the buffer from the polygons a like cookie cutter.

Let us try this:

```
central_park_intersects <- st_intersects(central_partk, nyc_sf_merged)
class(central_park_intersects)
```

We have created a `sgbp` object, which is a "Sparse Geometry Binary Predicate". It is a so called sparse matrix, which is a list with integer vectors only holding the indices for each polygon that intersects. In our case we only have one vector, because we only intersect with one buffer polygon,

so we can extract this first vector with `philly_buf_intersects[[1]]` and use it for subsetting:

```
# Subsetting using the results
nyc_sel_sf <- nyc_sf_merged[central_park_intersects[[1]],]
# or in tidyverse style
nyc_sel_sf <- nyc_sf_merged %>% dplyr::slice(central_park_intersects[[1]])


# plot
plot(st_geometry(nyc_sf_merged), border="#aaaaaa", main="Census tracts near Central Park")
plot(st_geometry(nyc_sel_sf), add=T, col="lightpink")
plot(st_geometry(nyc_sel_sf), add=T, col=NA, border= 'lightgrey',lwd = 1)
```


```
# or Interactive map using mapview
mapview(nyc_sel_sf)
```

2.4 Reprojecting or Projection Transform

Occasionally we may have to change the coordinates of our spatial object into a new Coordinate Reference System (CRS).

2.4.1 Transform or reproject *sf* objects

Let us check `nyc_sf_merged` object and reproject it to a local projection in State Plane Long Island, that is the local map projection the NYC.

```
st_crs(nyc_sf_merged)
```

```
#> Coordinate Reference System:
#>   User input: NAD83
#>   wkt:
#> GEOGCRS["NAD83",
#>     DATUM["North American Datum 1983",
#>         ELLIPSOID["GRS 1980",6378137,298.257222101,
#>             LENGTHUNIT["metre",1]]],
#>     PRIMEM["Greenwich",0,
#>         ANGLEUNIT["degree",0.0174532925199433]],
#>     CS[ellipsoidal,2],
#>         AXIS["latitude",north,
#>             ORDER[1],
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>         AXIS["longitude",east,
#>             ORDER[2],
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>     ID["EPSG",4269]]
```

```
# then we transform/reproject it to SPCS Long Island, 2831
nyc_sf_2831 <- st_transform(nyc_sf_merged, 2831)
st_crs(nyc_sf_2831)
```

```
#> Coordinate Reference System:
#>   User input: EPSG:2831
#>   wkt:
#> PROJCRS["NAD83(HARN) / New York Long Island",
#>     BASEGEOGCRS["NAD83(HARN)",
#>         DATUM["NAD83 (High Accuracy Reference Network)",
#>             ELLIPSOID["GRS 1980",6378137,298.257222101,
#>                 LENGTHUNIT["metre",1]]],
#>         PRIMEM["Greenwich",0,
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>         ID["EPSG",4152]],
#>     CONVERSION["SPCS83 New York Long Island zone (meters)",
#>         METHOD["Lambert Conic Conformal (2SP)",
#>             ID["EPSG",9802]],
#>         PARAMETER["Latitude of false origin",40.1666666666667,
#>             ANGLEUNIT["degree",0.0174532925199433],
#>             ID["EPSG",8821]],
#>         PARAMETER["Longitude of false origin",-74,
#>             ANGLEUNIT["degree",0.0174532925199433],
#>             ID["EPSG",8822]],
#>         PARAMETER["Latitude of 1st standard parallel",41.0333333333333,
#>             ANGLEUNIT["degree",0.0174532925199433],
#>             ID["EPSG",8823]],
#>         PARAMETER["Latitude of 2nd standard parallel",40.6666666666667,
#>             ANGLEUNIT["degree",0.0174532925199433],
#>             ID["EPSG",8824]],
```

```
#>         PARAMETER["Easting at false origin",300000,
#>             LENGTHUNIT["metre",1],
#>             ID["EPSG",8826]],
#>         PARAMETER["Northing at false origin",0,
#>             LENGTHUNIT["metre",1],
#>             ID["EPSG",8827]]],
#>     CS[Cartesian,2],
#>         AXIS["easting (X)",east,
#>             ORDER[1],
#>             LENGTHUNIT["metre",1]],
#>         AXIS["northing (Y)",north,
#>             ORDER[2],
#>             LENGTHUNIT["metre",1]],
#>     USAGE[
#>         SCOPE["Engineering survey, topographic mapping."],
#>         AREA["United States (USA) - New York - counties of Bronx; Kings; Nassau; New York
#>         BBOX[40.47,-74.26,41.3,-71.8]],
#>     ID["EPSG",2831]]
```

We see that the CRS are different for the two. One is geographic coordinate systems (longitude, latitude) using NAD83 and the other is New York State Plane Long Island.

We can also see the proj4 strings of the two.

```
st_crs(nyc_sf_merged)$proj4string
st_crs(nyc_sf_2831)$proj4string
```

: we have `+proj=lcc...` and `+proj=longlat...`. LCC refers to Lambert Conic Conformal, which is a projected coordinate system with numeric units.

We can use the `st_bbox()` method from the *sf* package to compare the coordinates before and after transformation and confirm that we actually have transformed them. `st_bbox()` returns the *min* and *max* values of the two dimensions of a *sf* spatial object.

```
sf::st_bbox(nyc_sf_merged) # bounding box
```

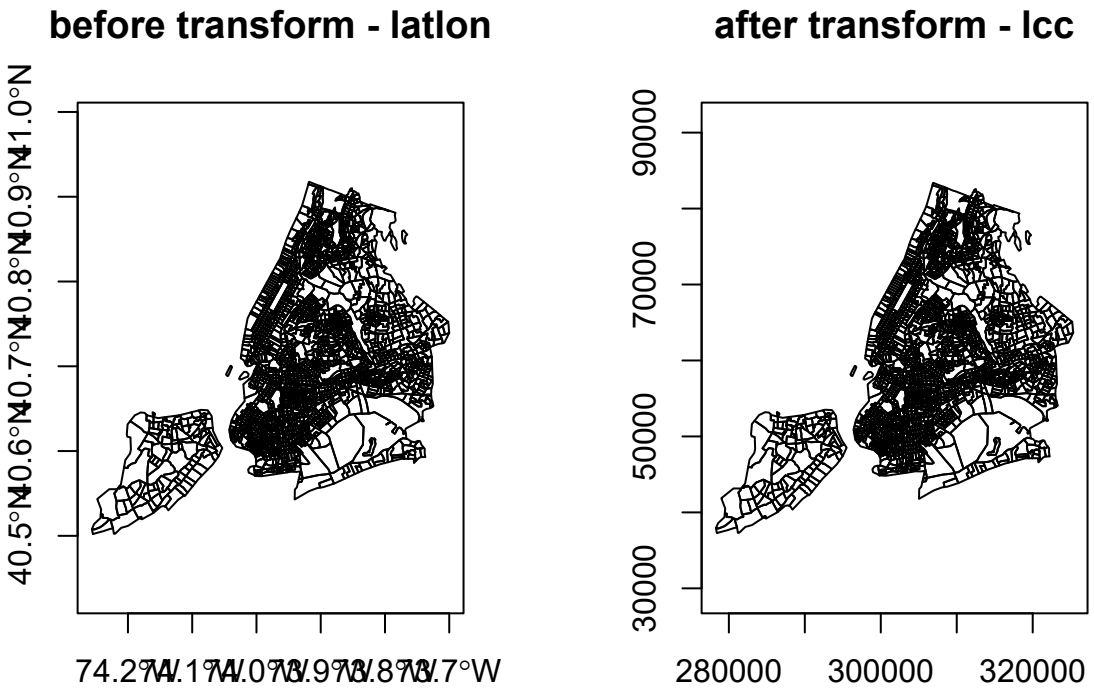
```
#>      xmin      ymin      xmax      ymax
#> -74.25609  40.50203 -73.70002  40.91758
```

```
st_bbox(nyc_sf_2831)
```

```
#>      xmin      ymin      xmax      ymax
#> 278294.41  37271.09 325338.38  83388.58
```

We can also compare them visually with:

```
par(mfrow=c(1,2))
plot(st_geometry(nyc_sf_merged), axes=TRUE, main = "before transform - latlon")
plot(st_geometry(nyc_sf_2831), axes=TRUE, main = "after transform - lcc")
```



Lastly, let us save the reprojected file as `nyc_tracts_2831` shapefile, as we may use it later on.

```
st_write(nyc_sf_2831, 'data/nyc/nyc_tracts_2831.shp')
```

Set CRS and Reproject (transform)

- Assigning a new CRS to a spatial object does not change its coordinates but the CRS determines how the software understand the coordinates. So, we must choose and set a CRS that is consistent with the coordinates. Otherwise, the coordinates will be misinterpreted.
- Reprojecting spatial data with st_transform and gTransform really changes the underlying coordinates, which we can see from the bbox values.

- In either case, the CRS must be consistent with the coordinates. It is a good practice to check projected spatial data against a basemap or a correctly referenced map to verify its CRS.
- SpatialReference.org is a very good source for CRS information.
- Some commonly used reference systems
  - US 48 Contiguous States: Albers Equal Area (ESRI:102003), Lambert Conformal (ESRI:102004)
  - New York State: UTM 18N (EPSG 3725, 3748, 26918)
  - New York City: State Plane Long Island (EPSG 2263, 2831, 3627)

2.4.2 Spatial Query with Reprojection

Another scenario where we need to transform spatial data into a different projection is to apply distances. This is because local projections tend to minimize distortion and they also use units like meter or feet. Here we use a distance buffer example to illustrate both transformation/reprojection and spatial query. In this example, we try to find the noise complaint within 200 meter from Hunter

College, 09/01/2019 to 09/01/2020.

First, we search the geographic coordinates of Hunter College and find (-73.9645, 40.7678). Then, we create a spatial object for this location. As we don't really use any attribute information for this location, a `sfc` object is sufficient.

```
# make a simple feature point with CRS
hunter_college_sfc <- st_sfc(st_point(c(-73.9645, 40.7678)), crs = 4269)
# Verify the location using mapview with basemap or ggmaps
mapview(hunter_college_sfc)
```

We can also use what we learned earlier to directly create a *sf* object.

```
# make a simple feature point with CRS
hunter_college_sf <- st_as_sf(data.frame(x=-73.9645, y=40.7678), coords = c('x','y'), crs =
# Verify the location using mapview with basemap or ggmaps
mapview(hunter_college_sf)
```

Now, let's read the Manhattan noise data and check on their coordinate systems. If they have the same CRS with a unit of meter, we can use them directly. Otherwise, we have to convert them to a local map projection with a meter unit.

```
man_noise_sf <- sf::st_read('data/nyc/ManhattanNoise.shp')
st_crs(hunter_college_sf)
st_crs(man_noise_sf)
```

It turns out both use geographic coordiantes with a unit of decimal degree. But one is using WGS84 (4326) and the other is using NAD83 (4326). Although they are very similar to each other, none of them have a unit of meter. So, we must convert both to the same CRS using New York Long Island (State Plain Coordinate System, SPCS) with a unit of meter. The SRID is 2831. Note 2263 is also New York Long Island, but its unit is feet.

To avoid saving all the intermediate results, we use pipe `%>%` to get the buffer and to conduct the spatial query.

```
  hunter_college_sf %>% sf::st_transform(2831) %>% # transform the data to 2831
    sf::st_buffer(200) %>% # create the buffer
    sf::st_intersects(man_noise_sf # intersects with the transformed noise data
                      %>% st_transform(2831)) %>%
    magrittr::extract2(., 1) %>% # Get the indices of the noise points within 200 meter buff
    dplyr::slice(man_noise_sf, .) %>% # Get those noise points
    mapview::mapview() # Map them out using a simple interactive map
```

This chunk of code may be difficult to understand. You can break it down and explicitly save the intermediate data using variables.

2.4.3 Raster reprojection

Here is what it would look like to reproject the HARV raster used earlier to a WGS84 projection. We see that the original projection is in UTM.

```
# if you need to load again:
#HARV <- raster("data/HARV_RGB_Ortho.tif")
raster::crs(HARV)
```

```
#> Coordinate Reference System:
#> Deprecated Proj.4 representation:
#>  +proj=utm +zone=18 +datum=WGS84 +units=m +no_defs
#> WKT2 2019 representation:
#> PROJCRS["WGS 84 / UTM zone 18N",
#>     BASEGEOGCRS["WGS 84",
#>         DATUM["World Geodetic System 1984",
#>             ELLIPSOID["WGS 84",6378137,298.257223563,
#>                 LENGTHUNIT["metre",1]]],
#>         PRIMEM["Greenwich",0,
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>         ID["EPSG",4326]],
#>     CONVERSION["UTM zone 18N",
#>         METHOD["Transverse Mercator",
#>             ID["EPSG",9807]],
#>         PARAMETER["Latitude of natural origin",0,
#>             ANGLEUNIT["degree",0.0174532925199433],
#>             ID["EPSG",8801]],
#>         PARAMETER["Longitude of natural origin",-75,
#>             ANGLEUNIT["degree",0.0174532925199433],
#>             ID["EPSG",8802]],
#>         PARAMETER["Scale factor at natural origin",0.9996,
#>             SCALEUNIT["unity",1],
#>             ID["EPSG",8805]],
#>         PARAMETER["False easting",500000,
#>             LENGTHUNIT["metre",1],
#>             ID["EPSG",8806]],
#>         PARAMETER["False northing",0,
#>             LENGTHUNIT["metre",1],
#>             ID["EPSG",8807]]],
#>     CS[Cartesian,2],
#>         AXIS["(E)",east,
#>             ORDER[1],
#>             LENGTHUNIT["metre",1]],
#>         AXIS["(N)",north,
#>             ORDER[2],
#>             LENGTHUNIT["metre",1]],
#>     USAGE[
#>         SCOPE["Engineering survey, topographic mapping."],
#>         AREA["Between 78Â°W and 72Â°W, northern hemisphere between equator and 84Â°N, onshore
#>         BBOX[0,-78,84,-72]],
#>     ID["EPSG",32618]]
```

```
HARV_WGS84 <- projectRaster(HARV, crs=CRS("+proj=longlat +datum=WGS84"))
```

Let's look at the coordinates to see the effect:

```
extent(HARV)
```

```
#> class      : Extent
#> xmin       : 731998.5
```

```
#> xmax       : 732766.8
#> ymin       : 4712956
#> ymax       : 4713536
```

```
extent(HARV_WGS84)
```

```
#> class      : Extent
#> xmin       : -72.17505
#> xmax       : -72.16544
#> ymin       : 42.53393
#> ymax       : 42.5394
```

```
ncell(HARV)
```

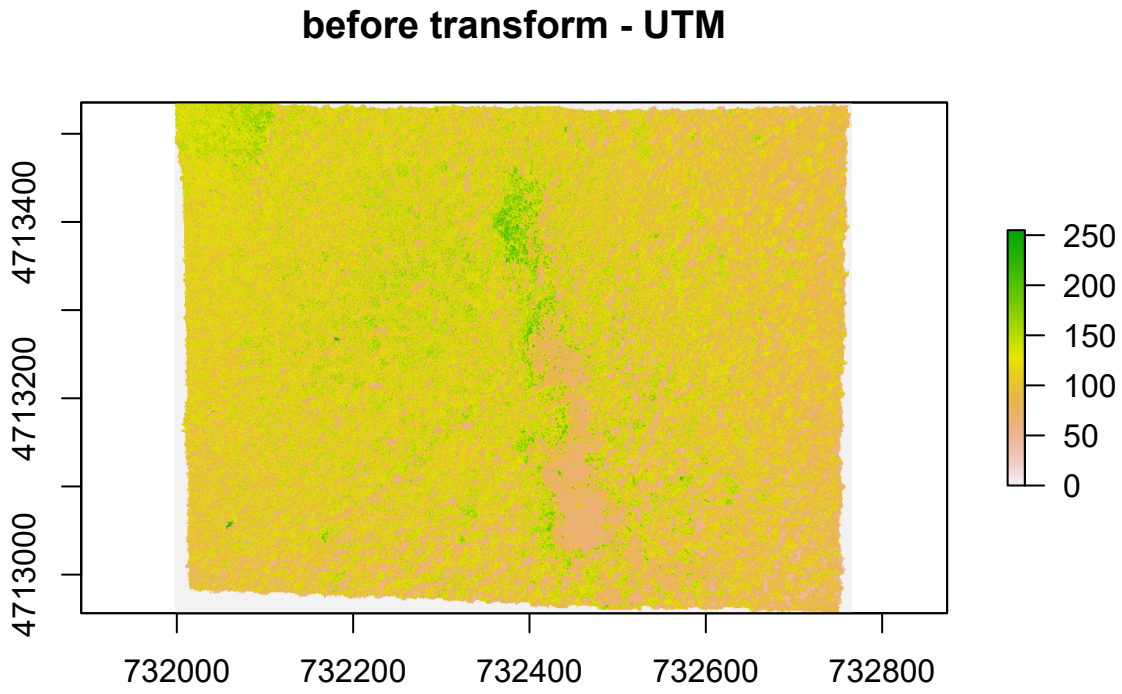
```
#> [1] 7120141
```

```
ncell(HARV_WGS84)
```

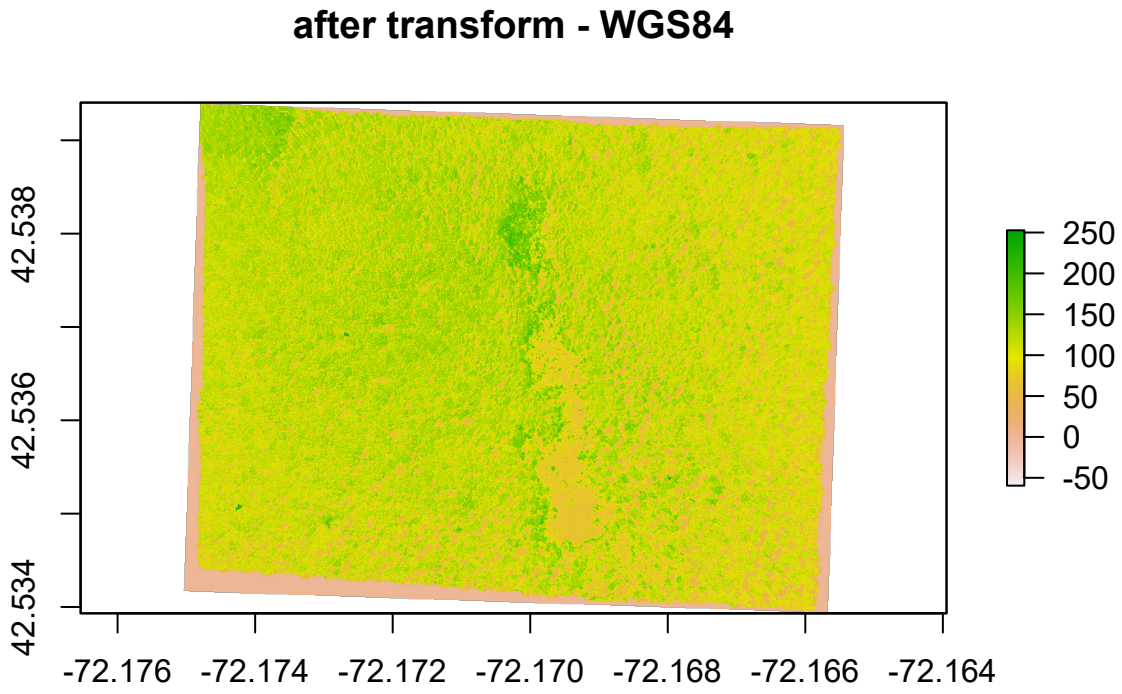
```
#> [1] 7687552
```

And here is the visual proof:

```
plot(HARV, main = "before transform - UTM")
```




```
plot(HARV_WGS84, main = "after transform - WGS84")
```



2.5 Spatial Join and Aggregation: Points in Polygons

Now that we have both noise and census tracts we will forge ahead and ask for the density of noise complaint for **each census tract** in Manhattan: $\frac{noisecomplaints}{area}$

To achieve this this we join the points of noise complaints to the census tract polygon and count them up for each polygon. You might be familiar with this operation from ArcGIS, QGIS, or other GIS packages. Note that for topological relationships not based on distances, it will work as long as both data have the same coordinate systems.

2.5.1 With *sf*

We will use piping and build up our object in the following way. First we calculate the area for each tract. We use the `st_area` function on the geometry column and add the result.

```
nyc_census_tracts_sf %>%
  filter(COUNTYFP == '061') %>%
  mutate(tract_area = st_area(geometry)) %>%
  head()
```

Next, we use st_join to perform a spatial join with the points:

```
nyc_census_tracts_sf %>%
  filter(COUNTYFP == '061') %>%
  mutate(tract_area = st_area(geometry)) %>%
  st_transform(4326) %>%
```

```
  st_join(man_noise_sf) %>%
  head()
```

Now we can group by a variable that unique identifies the census tracts, (we choose *GEOID*) and use `summarize` to count the points for each tract and calculate the rate of noise complaints. Since our units are in sq meter. multiply by by 1000000 to get sq km. We also need to carry over the area, which we can use `max`, `min`, `mean`, or `unique` as all the values are the same for the same census tract.

We also assign the output to a new object `crime_rate`.

```
noise_rate_sf <- nyc_census_tracts_sf %>%
      filter(COUNTYFP == '061') %>%
      mutate(tract_area = st_area(geometry)) %>%
      st_transform(4326) %>%
      st_join(man_noise_sf) %>%
      group_by(GEOID) %>%
      summarize(n_noise = n(),
                tract_area = max(tract_area),
                noise_rate = n_noise/tract_area * 1e6)
```

And here is a simple plot:

```
mapview(noise_rate_sf, zcol='noise_rate', legend=FALSE)
```

Finally, we write this out for later:

```
st_write(noise_rate_sf, "data/nyc/man_noise_rate.shp")
```

2.5.2 *sp* - *sf* comparison

how to..	for *sp* objects	for *sf* objects
join attributes	`sp::merge()`	`dplyr::*_join()` (also `sf::merge()`)
reproject	`spTransform()`	`st_transform()`
retrieve (or assign) CRS	`proj4string()`	`st_crs()`
count points in polygons	`aggregate()` or `over()`	`st_within()` or `aggregate()`
buffer	`rgeos::gBuffer()` (separate package)	`st_buffer()`
select by location	`g*` functions from `rgeos`	st_* geos functions in *sf*

Here are some additional packages that use vector data:

- `stplanr`: Functionality and data access tools for transport planning, including origin-destination analysis, route allocation and modelling travel patterns.
- `bikedata`: Data from public hire bicycle systems,including London, New York, Chicago, Washington DC, Boston, Los Angeles, and Philadelphia

2.6 Spatial Operations

2.6.1 Spatial Measures

This type of operations measure certain values from spatial objects. For example, we can derive the boundary box and centroid using `sf::st_bbox` and `sf::st_centroid`. More basic measures are

areas and length with `sf::st_area` and `sf::st_length`.

2.6.2 Geometric Operations

More interestingly, we can conduct certain geometric operations such as buffer and overlay. In spatial analysis, overlay contains a few types.

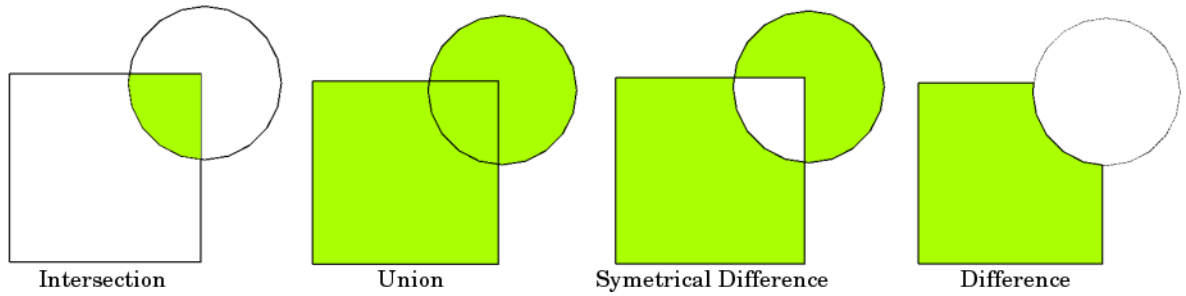


Figure 2.3: Overlay Types

And this web page has an excellent explanation to the spatial operations on vector data, including buffer, operations involving one (unary) or multiple layers of spatial objects.

Self Exercise

- Think about how to conduct these spatial operations using basic ***sf*** st_ and tidyverse functions.
- Dissolve: merge (data.frame), dplyr::group_by, st_union
- Append: dplyr::bind_rows, st_combine (may not give you what you expect, though)
- Identity: st_union and then st_intersection
- Erase: st_difference
- Split: st_intersection by feature ID

Geometric operations are algorithmically difficult and computationally expensive. So, quite often, results are not what we expected. More mature packages perform better. In this case, sp-family packages are better than sf-family.

```
buf_sf <-  hunter_college_sf %>% sf::st_transform(2831) %>% # reproject the data to 2831
    sf::st_buffer(800) # create the buffer

man_tracts_sf <- nyc_census_tracts_sf %>%
  st_transform(2831) %>%
  filter(COUNTYFP == '061')

buf_sel_sf <- buf_sf %>%
    sf::st_intersects(man_tracts_sf) %>% # intersects with the transformed noise data
    magrittr::extract2(., 1) %>% # Get the indices of the noise points within 200 meter buffer
    dplyr::slice(man_tracts_sf, .)

par(mfrow=c(1,2), mar = c(4, 0.1, 0.8, 0.1))

plot(buf_sf %>% st_geometry(),
     col='grey',
     main = 'Buffer Geometry')
```

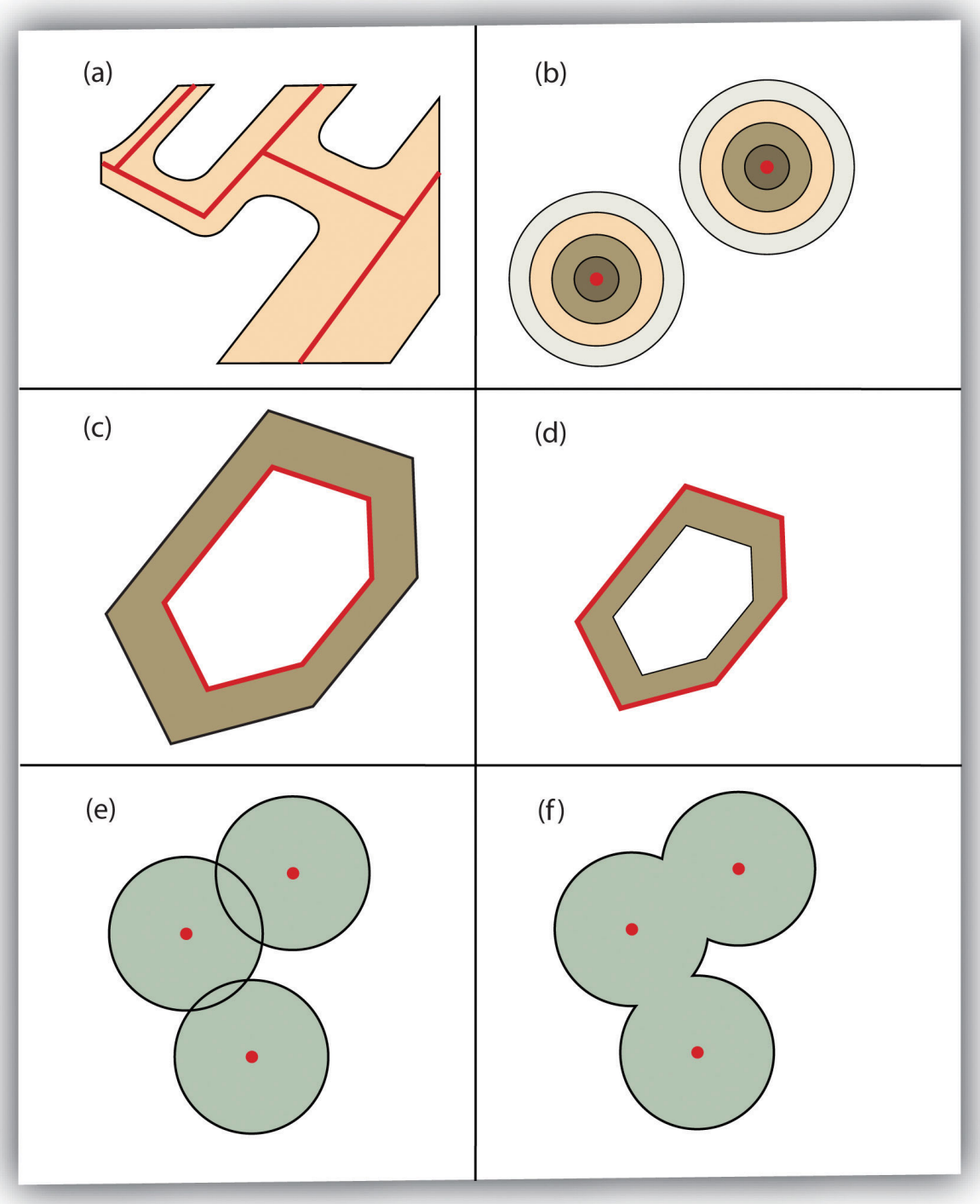


Figure 2.4: Buffer Options: (a) Variable Width Buffers, (b) Multiple Ring Buffers, (c) Doughnut Buffer, (d) Setback Buffer, (e) Nondissolved Buffer, (f) Dissolved Buffer

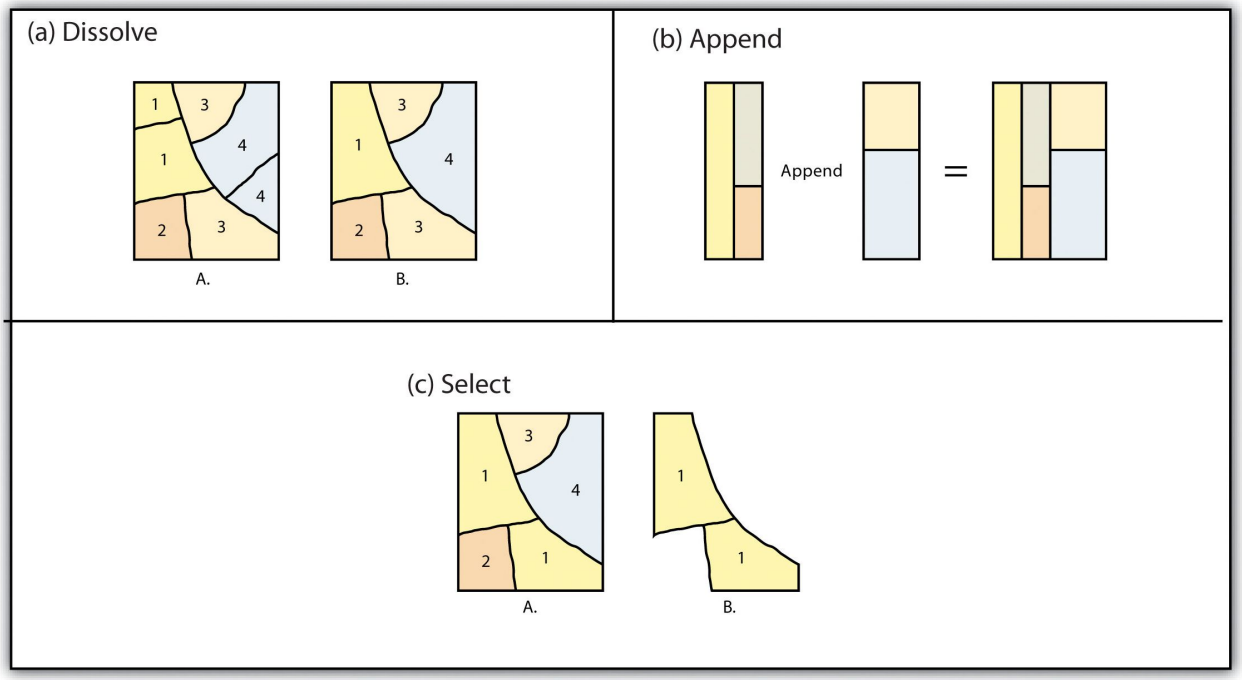
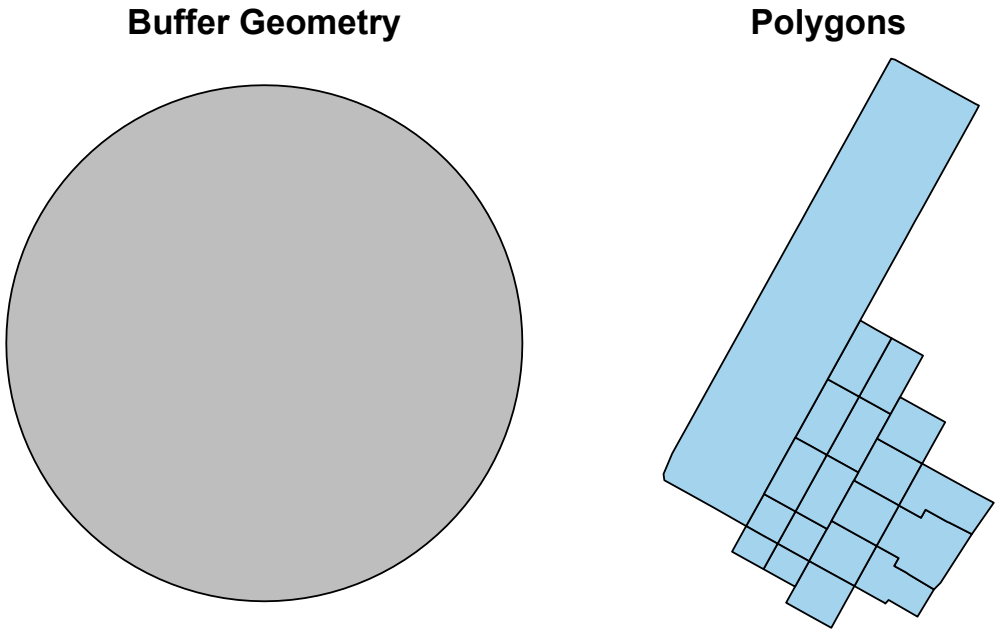


Figure 2.5: Single Layer Geometric Operations

```
plot(buf_sel_sf  %>% st_geometry(),
     col='lightskyblue2',
     main="Polygons")
```



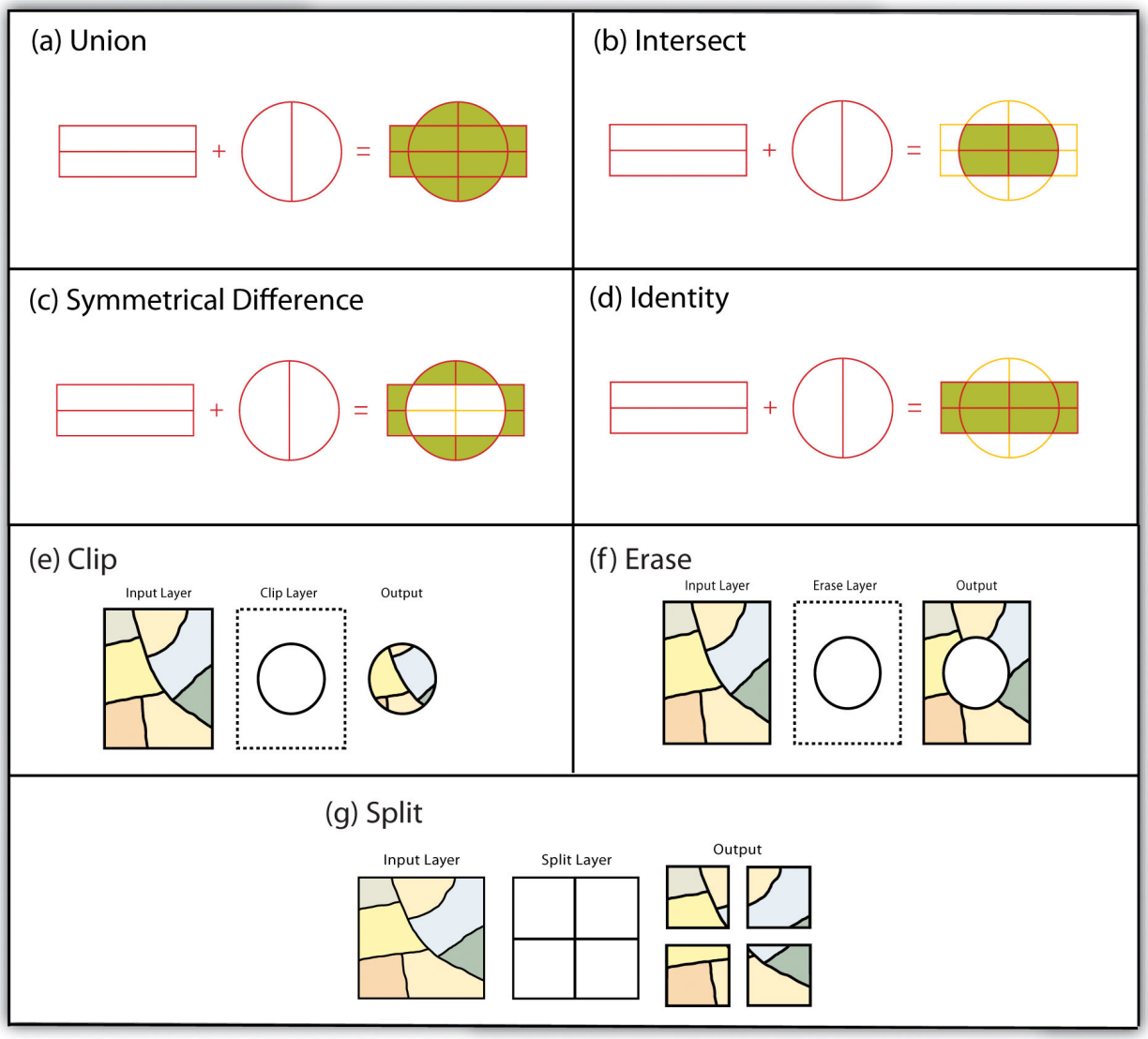
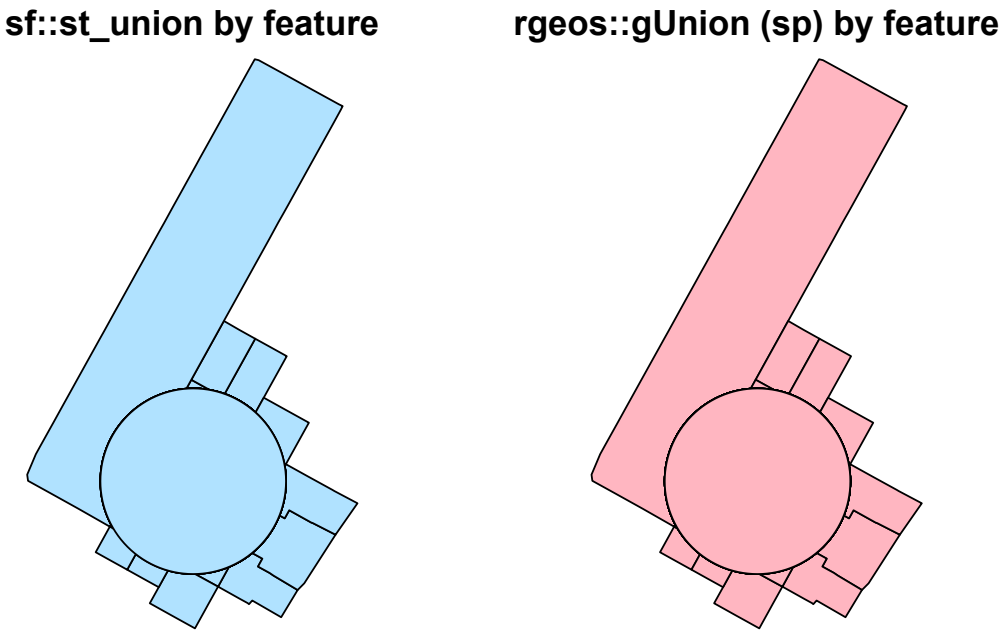
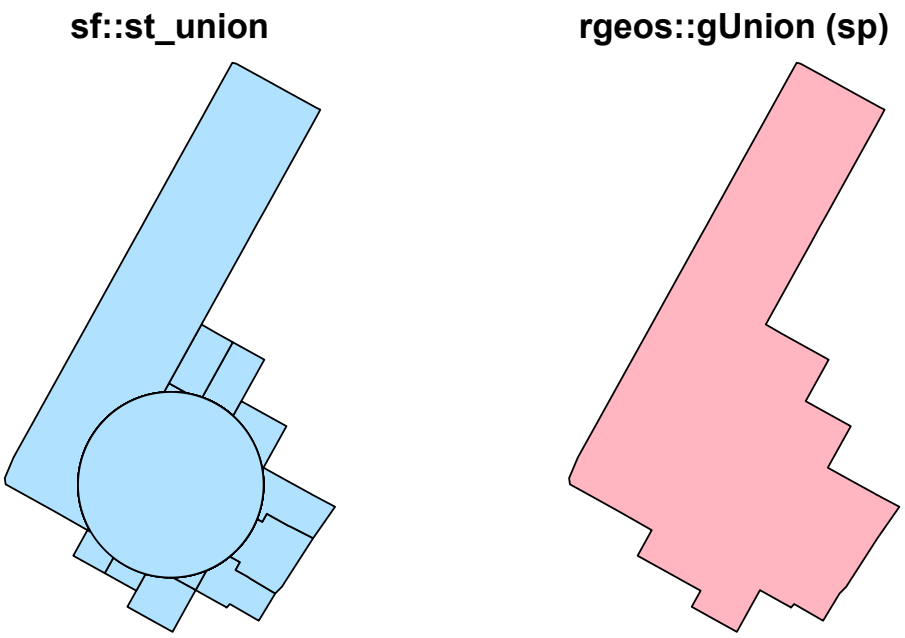


Figure 2.6: Multiple Layer Geometric Operations

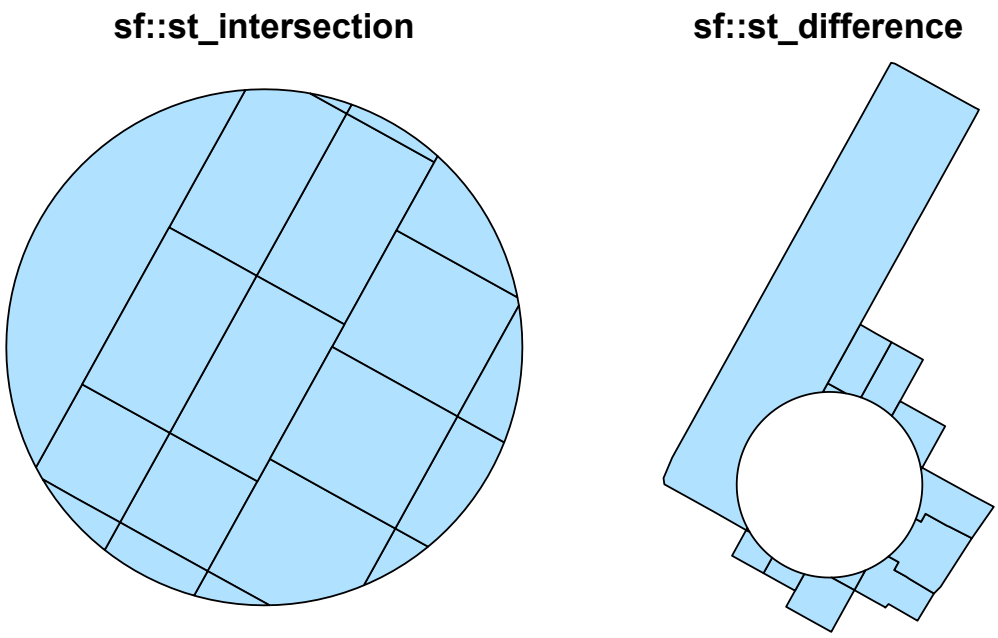
```
st_union(buf_sel_sf, buf_sf,by_feature = TRUE) %>%
  st_geometry() %>%
  plot(col='lightskyblue1', main = 'sf::st_union by feature')

rgeos::gUnion(buf_sf %>% sf::as_Spatial(),
              buf_sel_sf %>% sf::as_Spatial(),
              byid = TRUE) %>%
  plot(col='lightpink', main = 'rgeos::gUnion (sp) by feature')
```



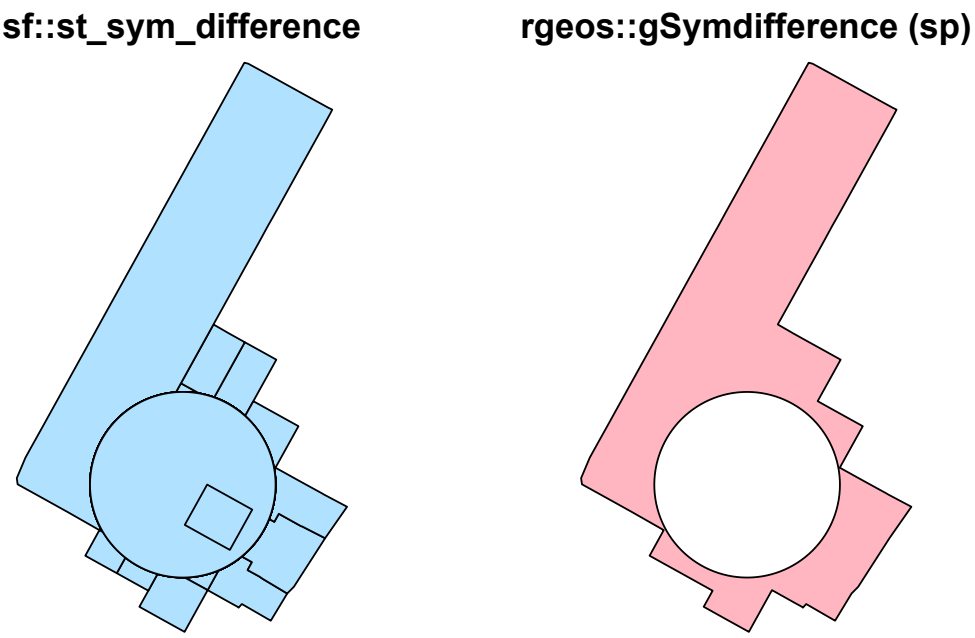

```
st_union(buf_sel_sf, buf_sf,by_feature = FALSE) %>%
  st_geometry() %>%
  plot(col='lightskyblue1', main = 'sf::st_union')

rgeos::gUnion(buf_sf %>% sf::as_Spatial(),
              buf_sel_sf %>% sf::as_Spatial()) %>%
  plot(col='lightpink', main = 'rgeos::gUnion (sp)')
```




```
st_intersection(buf_sel_sf, buf_sf) %>%
  st_geometry() %>%
  plot(col='lightskyblue1', main = 'sf::st_intersection')

st_difference(buf_sel_sf, buf_sf) %>% st_geometry() %>%
  plot(col='lightskyblue1',main = 'sf::st_difference')
```

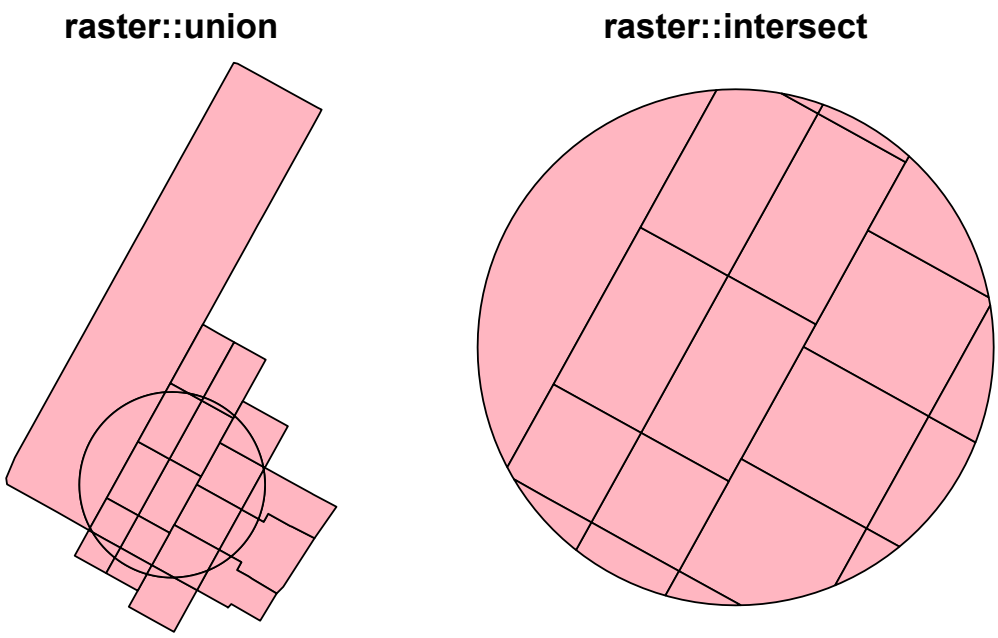



```
st_sym_difference(buf_sel_sf, buf_sf) %>%
  st_geometry() %>%
  plot(col='lightskyblue1',main = 'sf::st_sym_difference')

rgeos::gSymdifference(buf_sf %>% sf::as_Spatial(),
                      buf_sel_sf %>% sf::as_Spatial(),
                      drop_lower_td = TRUE) %>%
  plot(col='lightpink', main = 'rgeos::gSymdifference (sp)')
```



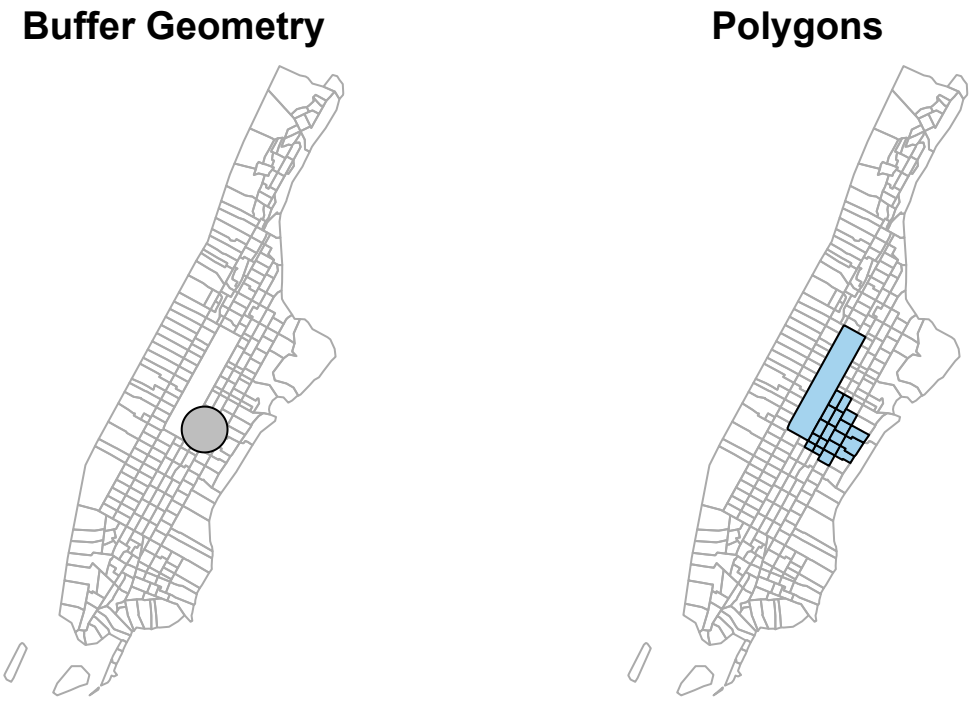
Interestingly, the ***raster*** package seems to have better *intersect* and *union* methods. Results from these methods are similar to what popular GIS software like ArcGIS produces. More importantly, they keep all the attributes from both layers.

```
par(mfrow=c(1,2), mar = c(4, 0.1, 0.8, 0.1));
raster::union(buf_sf %>% sf::st_geometry() %>% sf::as_Spatial(),
              buf_sel_sf %>% sf::st_geometry() %>% sf::as_Spatial()) %>%
  plot(col='lightpink', main = 'raster::union');

raster::intersect(buf_sf %>% sf::st_geometry() %>% sf::as_Spatial(),
                  buf_sel_sf %>% sf::st_geometry() %>% sf::as_Spatial()) %>%
  plot(col='lightpink', main = 'raster::intersect')
```

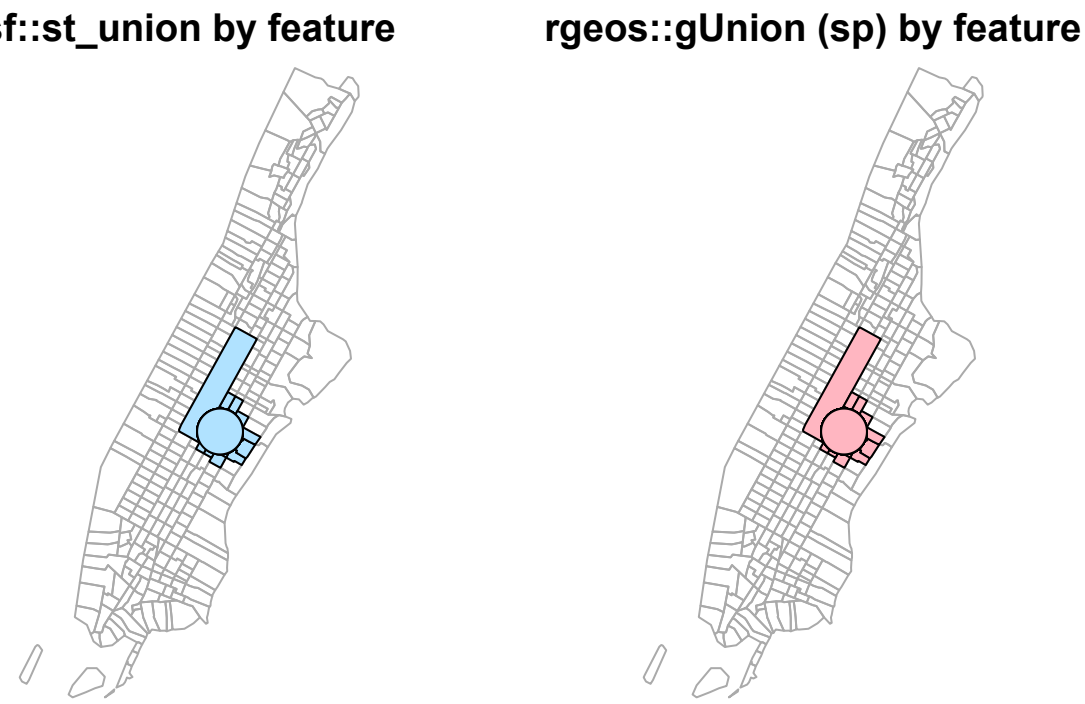


To visualize all these different operations with the orginal dataset.

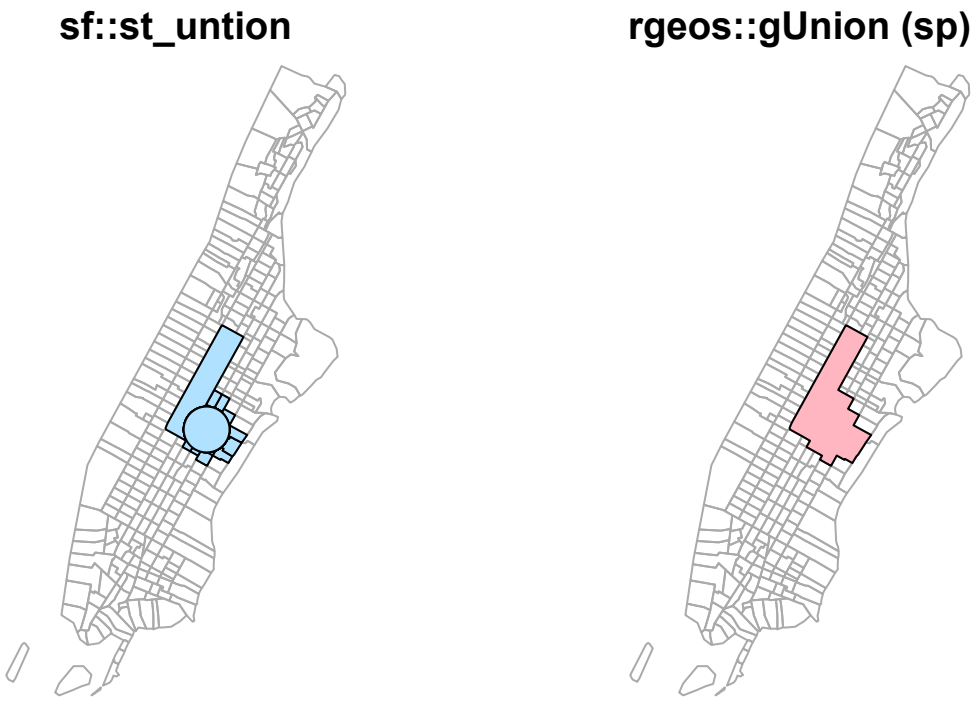
```
# Various types of spatial operations
par(mfrow=c(1,2), mar = c(4, 0.1, 0.8, 0.1))

plot(st_geometry(man_tracts_sf), border="#aaaaaa", main="Buffer Geometry")
plot(buf_sf %>% st_geometry(),
     col='grey',
     add=T)

plot(st_geometry(man_tracts_sf), border="#aaaaaa", main="Polygons")
plot(buf_sel_sf  %>% st_geometry(),
     col='lightskyblue2',
     add=T)
```



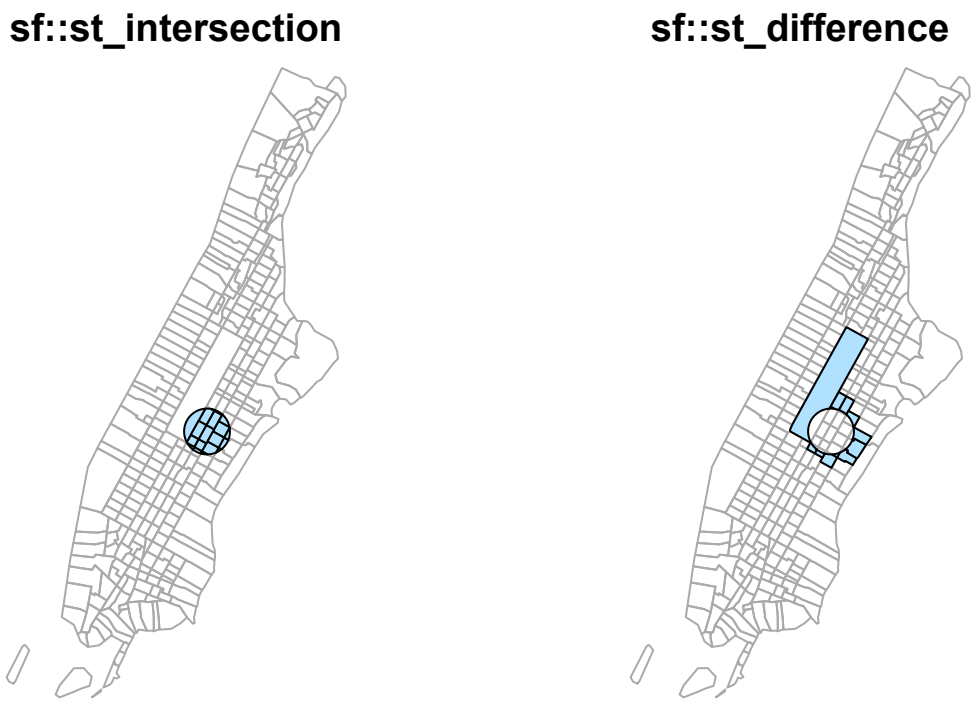

```
plot(st_geometry(man_tracts_sf), border="#aaaaaa", main="sf::st_union by feature")
st_union(buf_sel_sf, buf_sf,by_feature = TRUE) %>%
  st_geometry() %>%
  plot(col='lightskyblue1', add = T)

plot(st_geometry(man_tracts_sf), border="#aaaaaa", main='rgeos::gUnion (sp) by feature')
rgeos::gUnion(buf_sf %>% sf::as_Spatial(),
              buf_sel_sf %>% sf::as_Spatial(),
              byid = TRUE) %>%
  plot(col='lightpink', add = T)
```



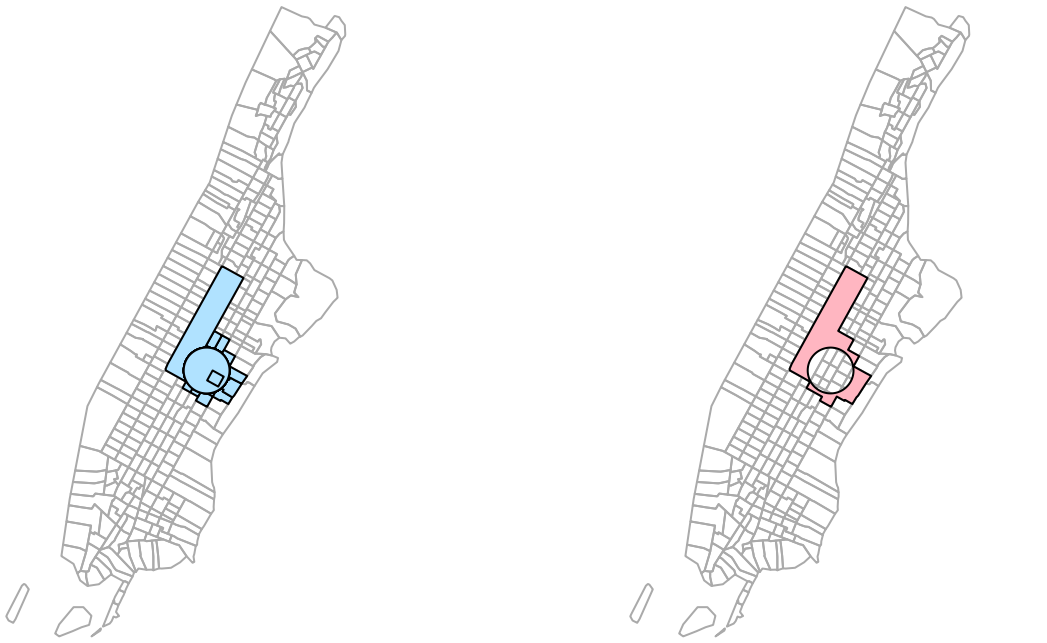

```
plot(st_geometry(man_tracts_sf), border="#aaaaaa", main='sf::st_untion')
st_union(buf_sel_sf, buf_sf,by_feature = FALSE) %>%
  st_geometry() %>%
  plot(col='lightskyblue1', add = T)

plot(st_geometry(man_tracts_sf), border="#aaaaaa", main='rgeos::gUnion (sp)')
rgeos::gUnion(buf_sf %>% sf::as_Spatial(),
              buf_sel_sf %>% sf::as_Spatial()) %>%
  plot(col='lightpink', add = T)
```



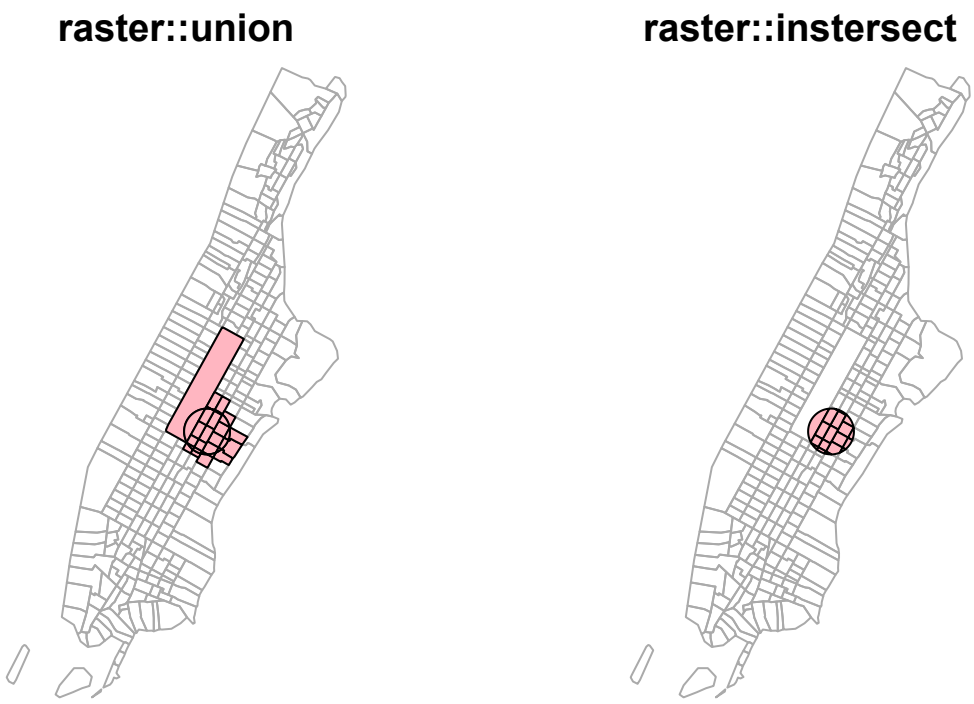

```
plot(st_geometry(man_tracts_sf), border="#aaaaaa", main='sf::st_intersection')
st_intersection(buf_sel_sf, buf_sf) %>%
  st_geometry() %>%
  plot(col='lightskyblue1', add = T)

plot(st_geometry(man_tracts_sf), border="#aaaaaa", main='sf::st_difference')
st_difference(buf_sel_sf, buf_sf) %>% st_geometry() %>%
  plot(col='lightskyblue1', add = T)
```




```
plot(st_geometry(man_tracts_sf), border="#aaaaaa", main='sf::st_sym_difference')
st_sym_difference(buf_sel_sf, buf_sf) %>%
  st_geometry() %>%
  plot(col='lightskyblue1', add = T)

plot(st_geometry(man_tracts_sf), border="#aaaaaa", main='rgeos::gSysdifference (sp)')
rgeos::gSymdifference(buf_sf %>% sf::as_Spatial(),
                      buf_sel_sf %>% sf::as_Spatial(),
                      drop_lower_td = TRUE) %>%
  plot(col='lightpink', add = T)
```

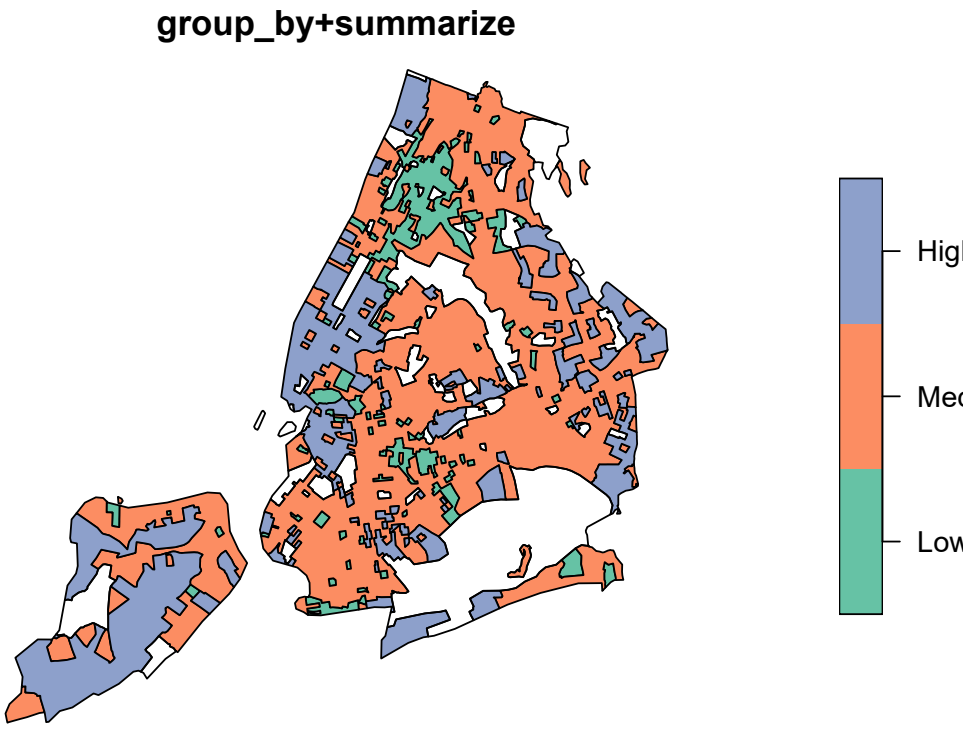



```
par(mfrow=c(1,2), mar = c(4, 0.1, 0.8, 0.1))

plot(st_geometry(man_tracts_sf), border="#aaaaaa", main='raster::union')
raster::union(buf_sf %>% sf::st_geometry() %>% sf::as_Spatial(),
              buf_sel_sf %>% sf::st_geometry() %>% sf::as_Spatial()) %>%
  plot(col='lightpink', add = T)

plot(st_geometry(man_tracts_sf), border="#aaaaaa", main='raster::instersect')
raster::intersect(buf_sf %>% sf::st_geometry() %>% sf::as_Spatial(),
                  buf_sel_sf %>% sf::st_geometry() %>% sf::as_Spatial()) %>%
  plot(col='lightpink', add = T)
```

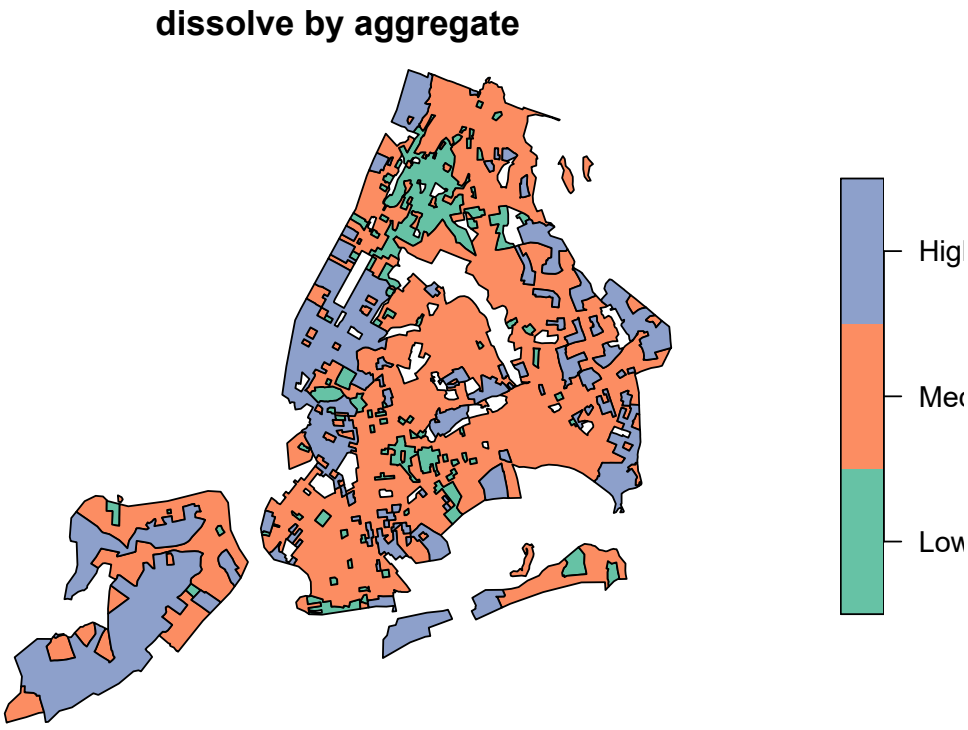


Earlier, we used ***group_by + summarize (sf)*** and ***aggregate (sp)*** to conduct spatial aggregation or spatial join. When these functions apply to a single spatial object (unary operations), they could do the dissolve. Note that those internal "sliver" polygons are caused by inaccurate boundaries. For example, two neighboring polygons should have exactly the same boundary at where they touch. But if not, those sliver polygons will be produced during spatial operations. That also partially explains why sf performs very bad because sf assumes data strictly following Simple Features specifications. Obviously, the Philly data do not.

```
par(mfrow=c(1,2), mar = c(4, 0.1, 0.8, 0.1));
#philly_sf_merged %>% st_geometry() %>% plot();
nyc_sf_merged %>%
  dplyr::mutate(incomeFactor = cut(MEDHHINC,
                                   c(0, 30000, 80000, Inf),
                                   c('Low', 'Medium', 'High'))) %>%
  dplyr::select(incomeFactor, geometry ) %>%
  group_by(incomeFactor) %>%
  summarise() %>% plot(main="group_by+summarize");
```




```
groupFactor <-  cut(nyc_sf_merged$MEDHHINC,
                    c(0, 30000, 80000, Inf),
                    c('Low', 'Medium', 'High'));

nyc_sf_merged %>%
  dplyr::select(MEDHHINC, geometry ) %>%
  aggregate(by = list(FemDocLevel = groupFactor), sum) %>%
  magrittr::extract('FemDocLevel') %>% plot(main="dissolve by aggregate")
```



2.7 Information for Raster Operations

Some helpful packages that deal with raster data:

- `landscapetools` provides utility functions to complete tasks involved in common landscape analysis.
- `getlandsat`: Get Landsat 8 Data from Amazon Public Data Sets
- `MODIStsp`: automates the creation of time series of rasters derived from MODIS Land Products data
- `FedData`: Download geospatial Data from federated data sources, including the The National Elevation Dataset digital elevation models, the Global Historical Climatology Network, the National Land Cover Database, and more.

2.8 Lab Assignment

The second lab is to aggregate data from different sources to the zip codes as the core covid-19 data are available at that scale.

Main tasks for the second lab are:

1. Join the COVID-19 data to the NYC zip code area data (sf or sp polygons).
2. Aggregate the NYC food retails store data (points) to the zip code data, so that we know how many retail stores in each zip code area. Note that not all locations are for food retail. And we need to choose the specific types according to the data.
3. Aggregate the NYC health facilities (points) to the zip code data. Similarly, choose appropriate subtypes such as nursing homes from the facilities.
4. Join the Census ACS population, race, and age data to the NYC Planning Census Tract Data.
5. Aggregate the ACS census data to zip code area data.

In the end, we should have the confirmed and tested cases of covid-19, numbers of specific types of food stores, numbers of specific types of health facilities, and population (total population, elderly, by race, etc.) at the zip code level. We should also have boroughs, names, etc. for each zip code area.

Chapter 3

Making Maps in R

Learning Objectives

- plot an ***sf*** object
- create a choropleth map with ***ggplot***
  - add a basemap with ***ggmap***
  - use ***RColorBrewer*** to improve legend colors
  - use ***classInt*** to improve legend breaks
- create a choropleth map with ***tmap***
- create an interactive map with ***leaflet***

In the preceding examples we have used the base `plot` command to take a quick look at our spatial objects. In this section we will explore several alternatives to map spatial data with R. For more packages see the "Visualization" section of the CRAN Task View.

Mapping packages are in the process of keeping up with the development of the new *sf* package, so they typically accept both *sp* and *sf* objects. However, there are a few exceptions.

Of the packages shown here `spplot()`, which is part of the good old *sp* package, only takes *sp* objects. The development version of *ggplot2* can take *sf* objects, though `ggmap` seems to still have issues with *sf*. Both *tmap* and *leaflet* can also handle both *sp* and *sf* objects.

3.1 Plotting simple features (*sf*) with `plot`

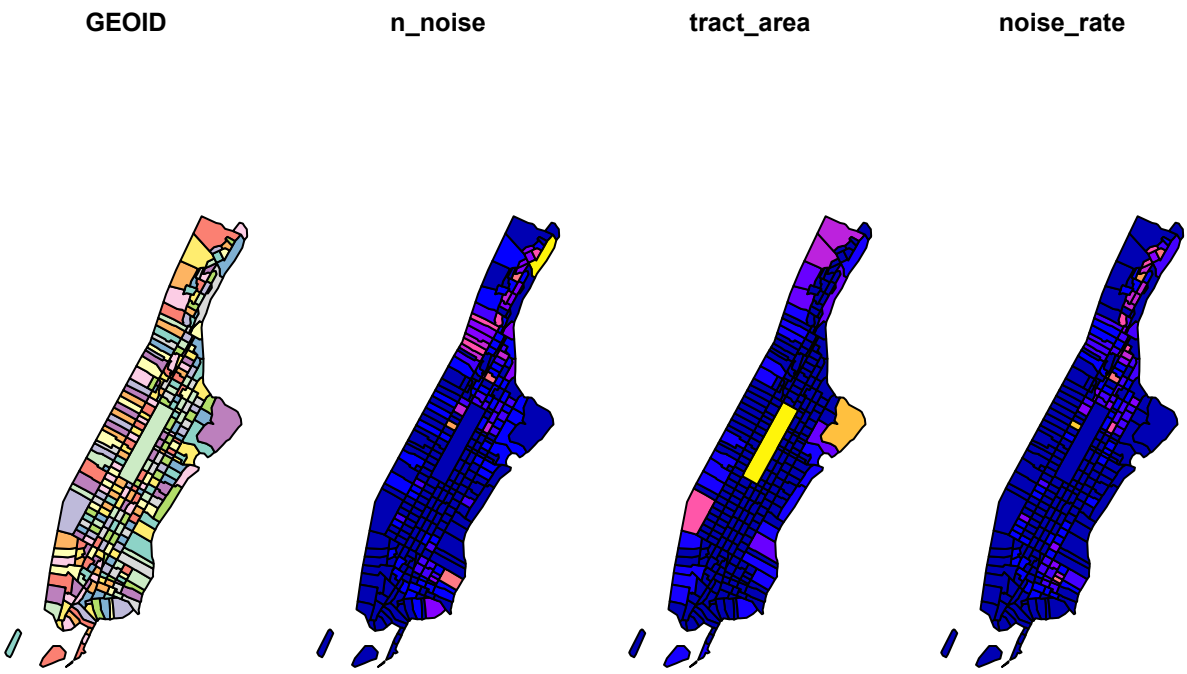
As we have already briefly seen, the *sf* package extends the base `plot` command, so it can be used on *sf* objects. If used without any arguments it will plot all the attributes using up to 10 "pretty" breaks. See its document ion by typing `?sf::plot.sf` on the console.

```
# Read Manhattan noise complaints data aggregtated in census tracts
man_noise_rate_sf <-  st_read("./data/nyc/man_noise_rate.shp", quiet = TRUE)
# Read homicides points and transform it to the same CRS as census tract above
man_noises_pt <- sf::st_read('./data/nyc/ManhattanNoise.shp') %>%
  sf::st_transform(sf::st_crs(man_noise_rate_sf));
```

```
#> Reading layer `ManhattanNoise' from data source
#>   `D:\Cloud_Drive\Dropbox (Hunter College)\Workspace\RSpace\R-Spatial_Book\data\nyc\Manha
#>   using driver `ESRI Shapefile'
```

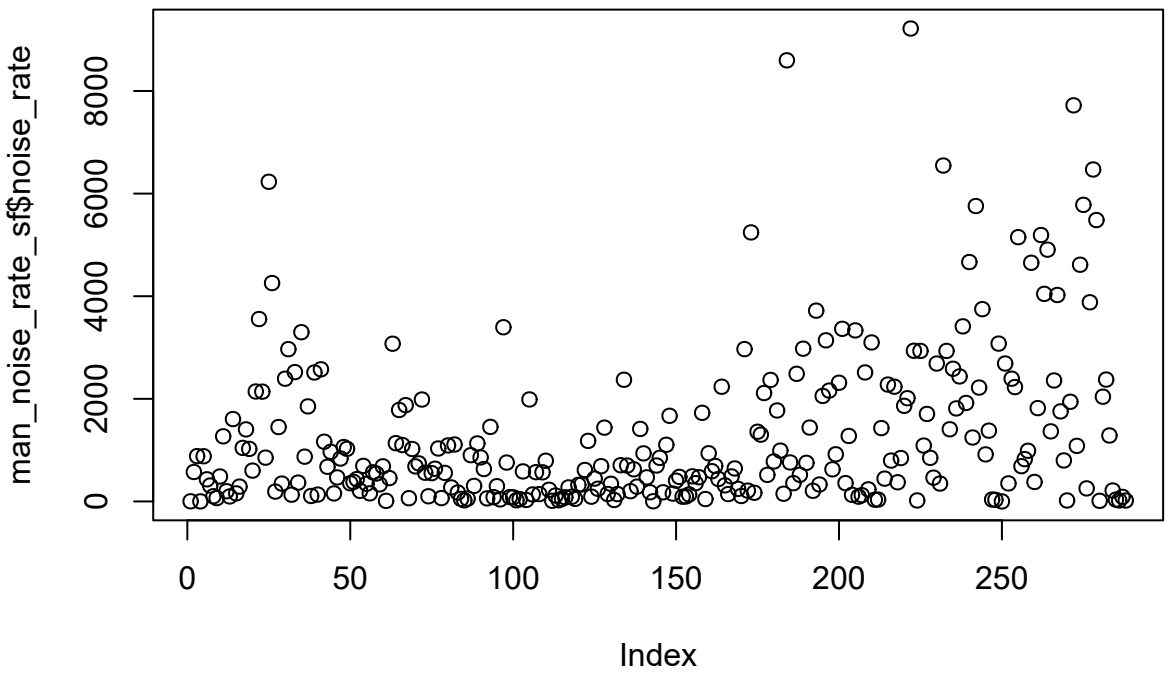
```
#> Simple feature collection with 68582 features and 5 fields
#> Geometry type: POINT
#> Dimension:     XY
#> Bounding box:  xmin: -74.018 ymin: 40.69889 xmax: -73.90809 ymax: 40.87783
#> Geodetic CRS:  WGS 84
```

```
# Simple plot.sf
plot(man_noise_rate_sf)
```

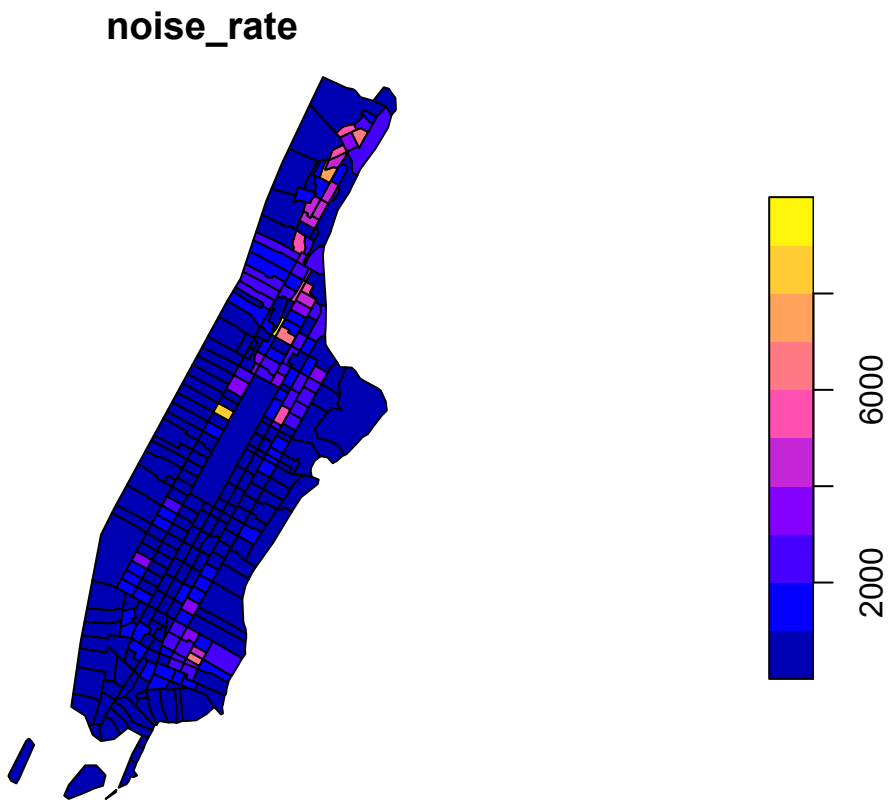


To plot a single attribute we need to provide an object of class *sf*, like so:

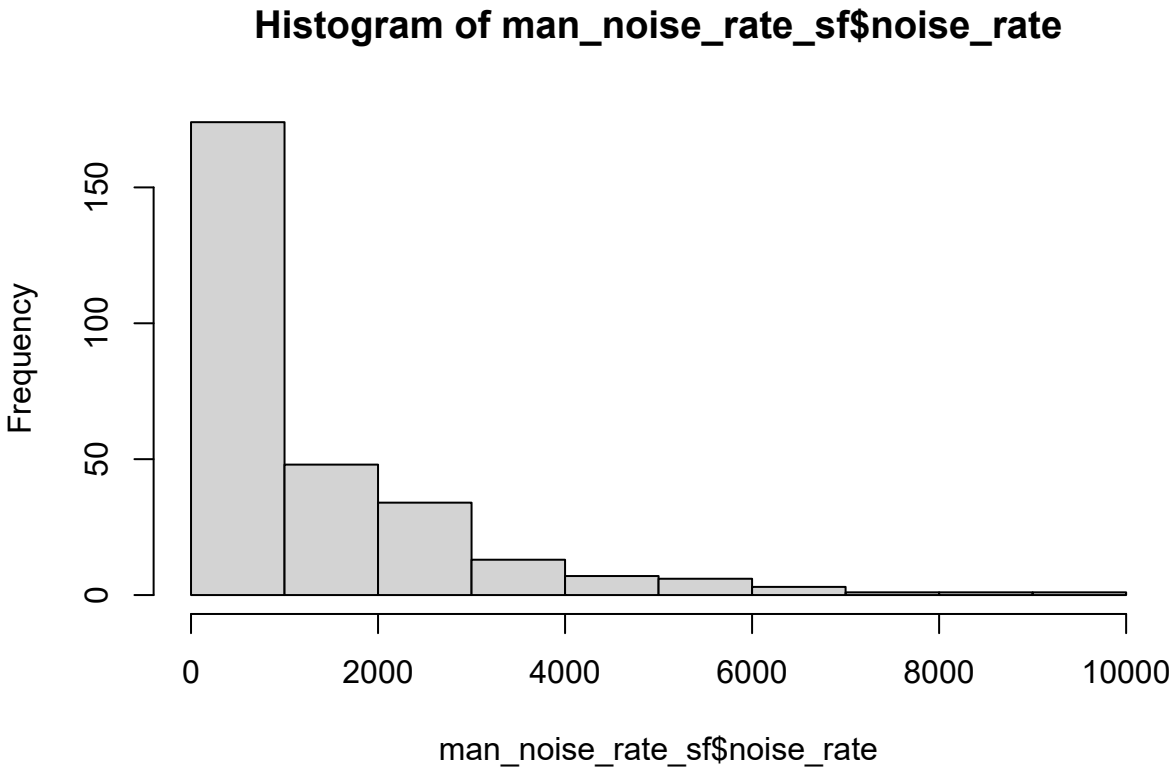
```
# Plot a single column (attribute)
plot(man_noise_rate_sf$noise_rate) # this is a numeric vector!
```




```
plot(man_noise_rate_sf["noise_rate"]) # this plot one column
```

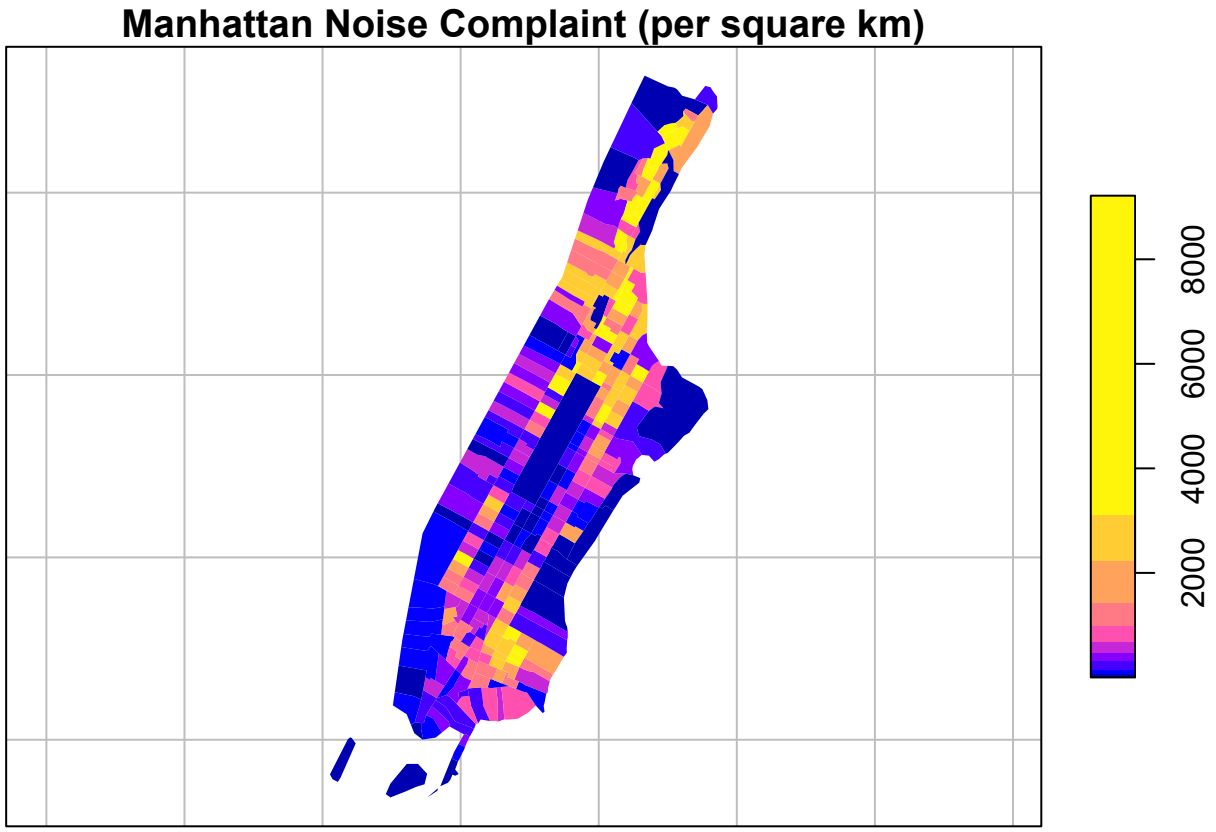


Since our values are unevenly distributed...:



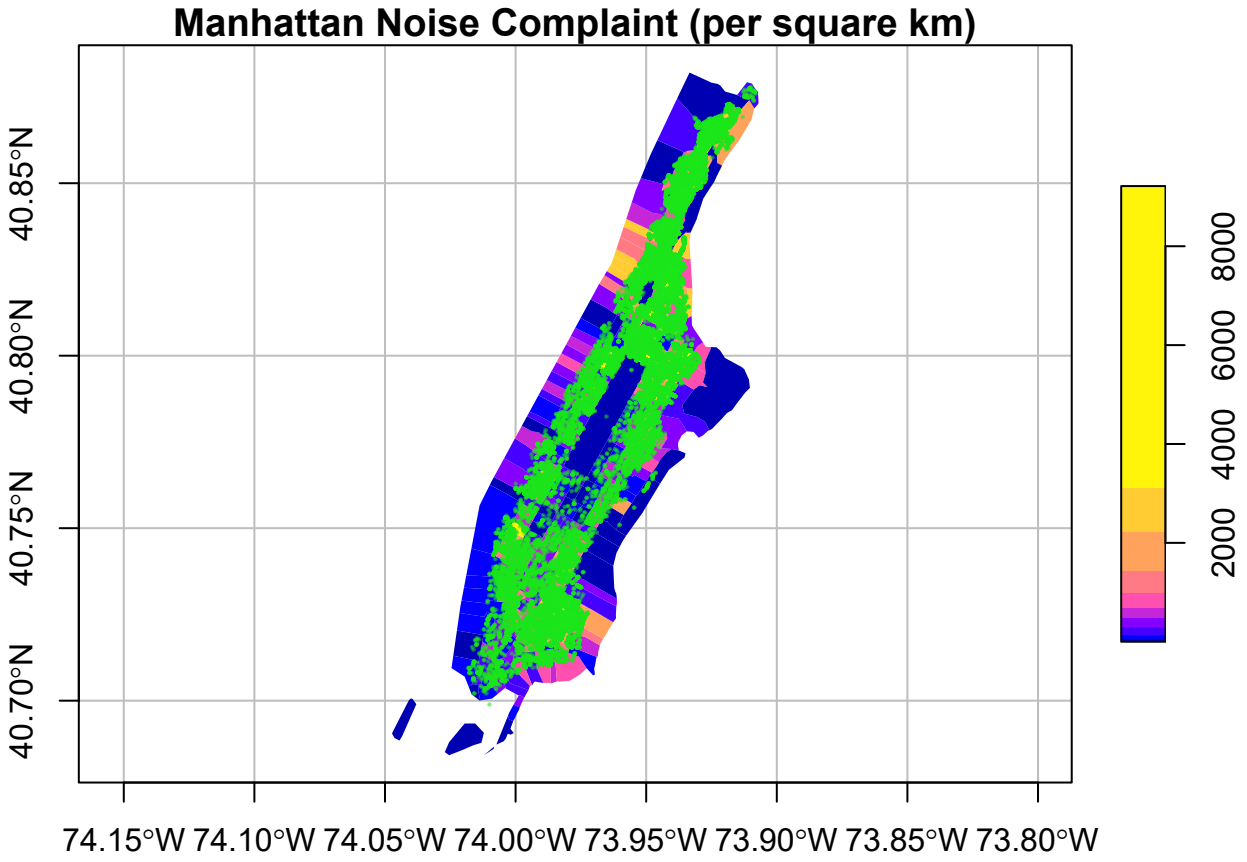
...we might want to set the breaks to quantiles in order to better distinguish the census tracts with low values. This can be done by using the `breaks` argument for the sf plot function.

```
plot(man_noise_rate_sf["noise_rate"],
     main = "Manhattan Noise Complaint (per square km)",
     breaks = "quantile",
     border = NA,
     graticule = TRUE)
```



Using the `reset=FALSE` parameter in the first `plot` on `*sf*` object and `add=TRUE` on following plots, we can plot multiple layers on the map.

```
ptColor <- rgb(0.1,0.9,0.1,0.4);
plot(man_noise_rate_sf["noise_rate"],
     main = "Manhattan Noise Complaint (per square km)",
     breaks = "quantile",
     border = NA,
     graticule = st_crs(4326),
     axes=TRUE,
     reset=FALSE)
plot(man_noises_pt["incdnt_z"], pch=19, col=ptColor, cex=0.1, add=TRUE)
```



We can change the color palette using a library called `RColorBrewer`[1]. For more about ColorBrewer palettes read this.

To make the color palettes from ColorBrewer available as R palettes we use the `brewer.pal()` function. It takes two arguments: - the number of different colors desired and - the name of the palette as character string.

We select 7 colors from the 'Orange-Red' palette and assign it to an object `pal`.

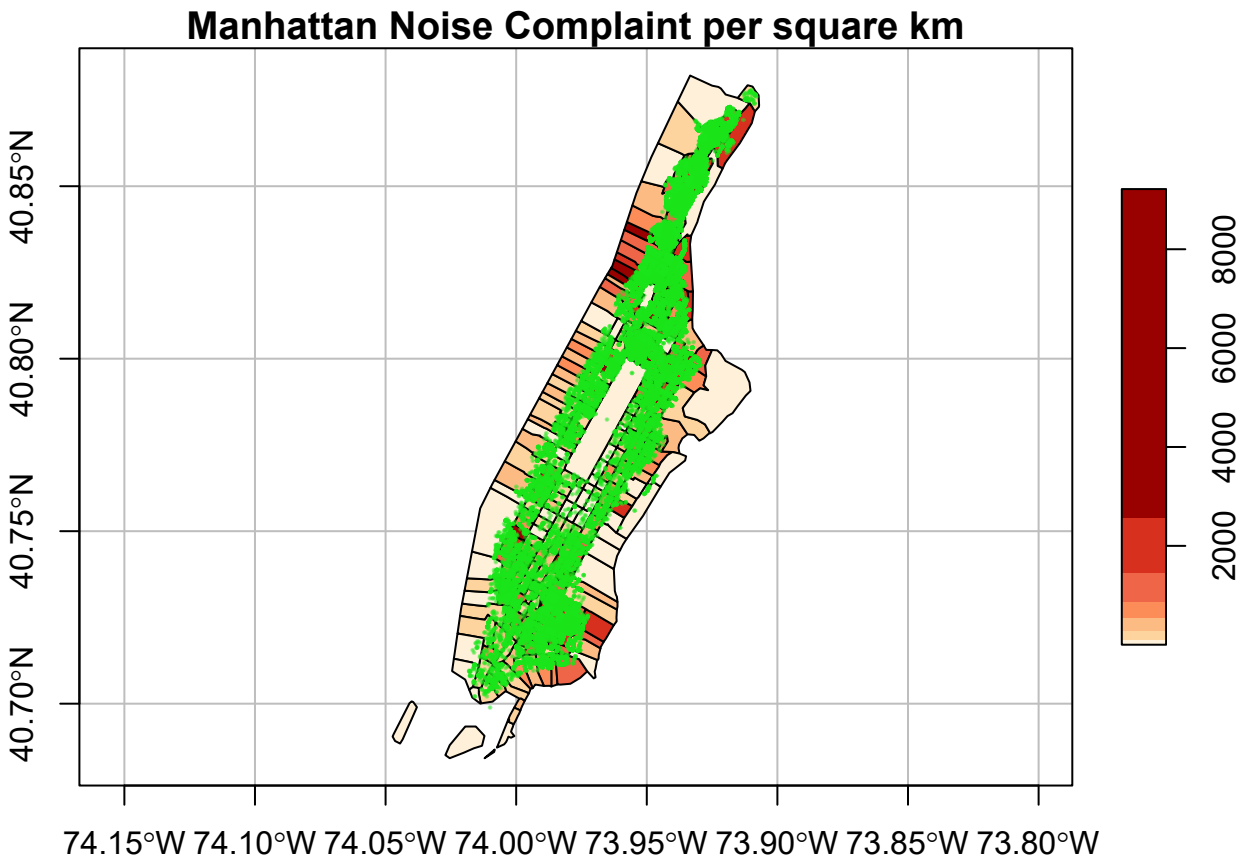
```
library(RColorBrewer)
pal <- brewer.pal(7, "OrRd") # we select 7 colors from the palette
class(pal)
```

```
#> [1] "character"
```

Finally, we add this to the plot

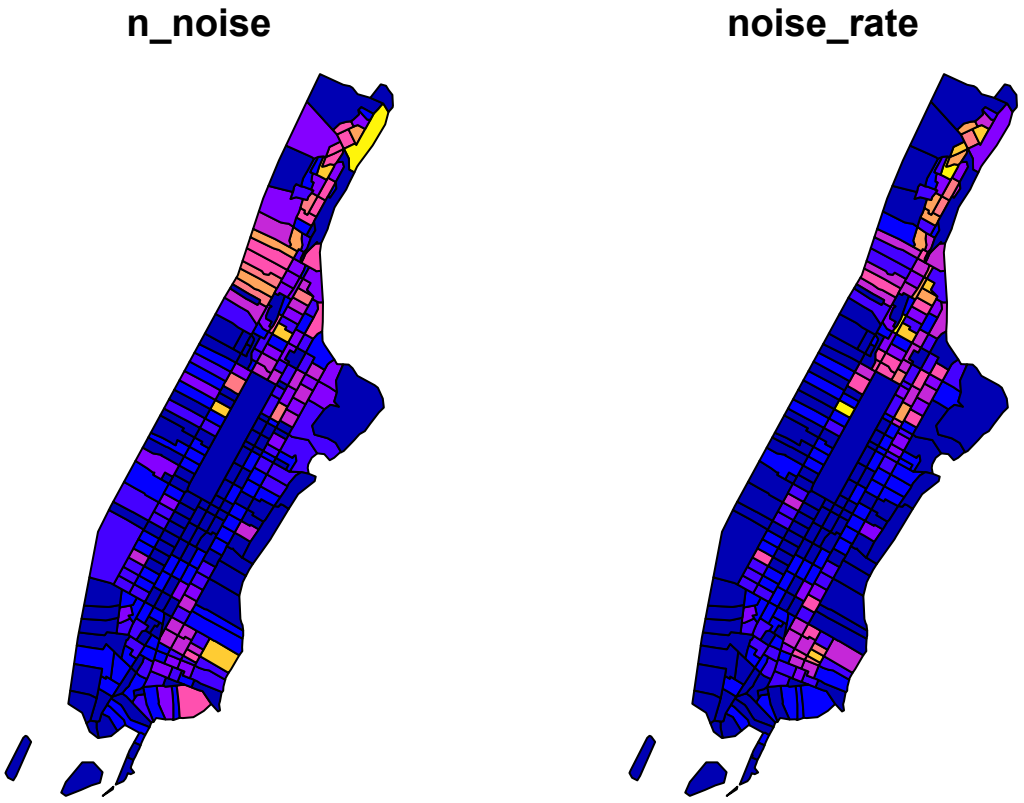
```
plot(man_noise_rate_sf["noise_rate"],
     main = "Manhattan Noise Complaint per square km",
     breaks = "quantile", nbreaks = 7,
     pal = pal,
     graticule = st_crs(4326),
     axes=TRUE,
     reset=FALSE)
plot(man_noises_pt["incdnt_z"], pch=19, col=ptColor, cex=0.1, add=TRUE)
```

[1]This is not the only way to provide color palettes. You can create your customized palette in many different ways or simply as a vector of hexbin color codes, like `c( "#FDBB84" "#FC8D59" "#EF6548")`.



We can also choose to plot multiple columns side by side using `plot.sf`. But this is only good for simple illustration as it is harder to customize the two plots.

```
plot(man_noise_rate_sf[c("n_noise","noise_rate")], breaks = 'jenks') # this plot multiple co
```

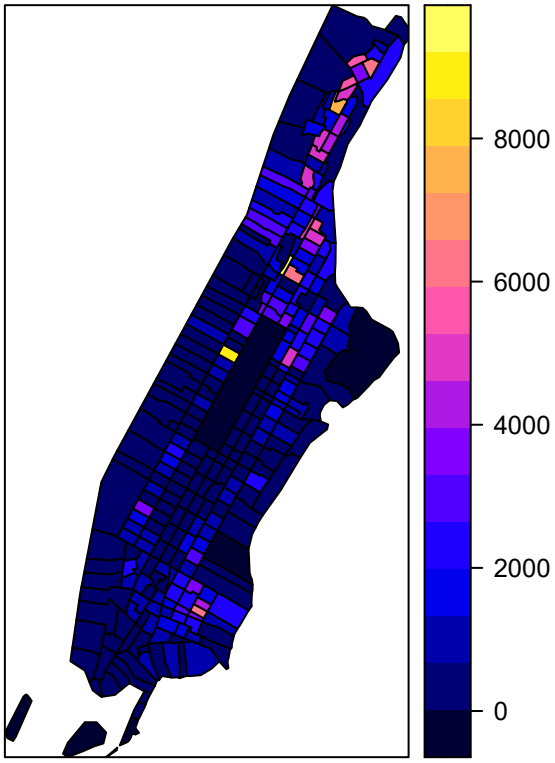


3.2 Choropleth mapping with `spplot`

sp comes with a plot command `spplot()`, which takes `Spatial*` objects to plot. `spplot()` is one of the earlier functions around to plot geographic objects.

Like plot, by default `spplot` maps everything it can find in the attribute table. Sometimes this does not work, depending on the data types in the attribute table. In order to select specific values to map we can provide the `spplot` function with the name (or names) of the attribute variable(s) we want to plot. It is the name of the column of the `Spatial*Dataframe` as character string (or a vector if several).

```
spplot(man_noise_rate_sf %>% sf::as_Spatial(), "noise_rate")
```

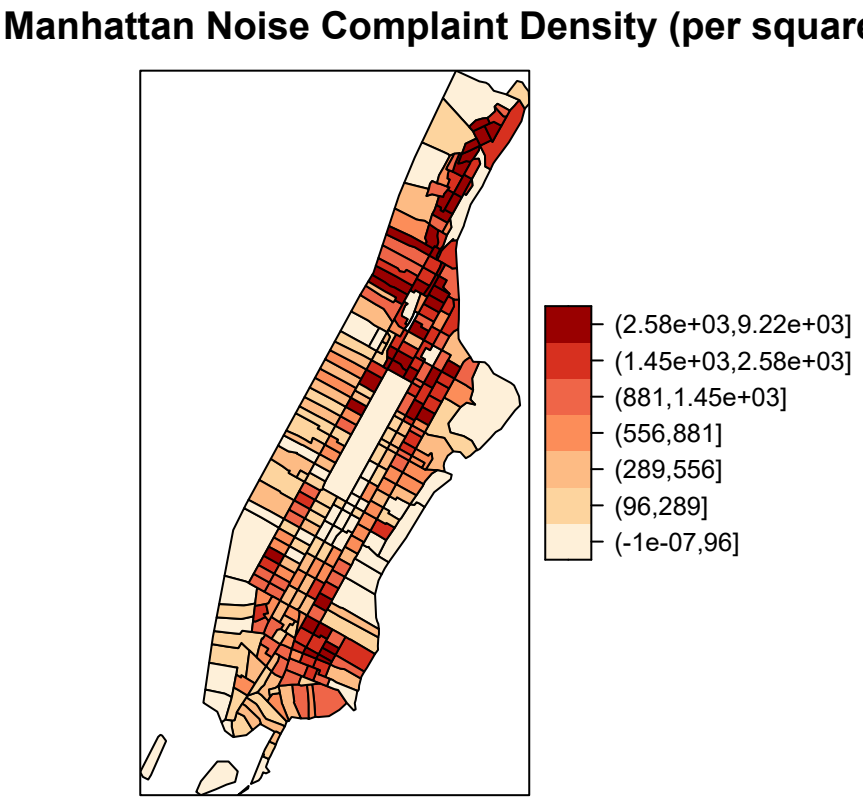


Many improvements can be made here as well, below is an example[2] [3].

```
# quantile breaks
require(classInt)
breaks_qt <- classIntervals(man_noise_rate_sf$noise_rate, n = 7, style = "quantile")
br <- breaks_qt$brks %>% as.integer()
offs <- 0.0000001
br[1] <- br[1] - offs
br[length(br)] <- br[length(br)] + offs
# categoreis for choropleth map
man_noise_rate_sf$noise_rate_bracket <- cut(man_noise_rate_sf$noise_rate, br)
# plot
spplot(man_noise_rate_sf %>% sf::as_Spatial(), "noise_rate_bracket", col.regions=pal, main =
```

[2]For more details see Chaps 2 and 3 in Applied Spatial Data Analysis with R. Also, `spplot` is a wrapper for the `lattice` package, see there for more advanced options.

[3]For the correction of breaks after using classIntervals with spplot/levelplot see here http://r.789695.n4.nabble.com/SpatialPolygon-with-the-max-value-gets-no-color-assigned-in-spplot-function-when-using-quot-at-quot-r-td4654672.html



3.3 Choropleth mapping with *ggplot2*

ggplot2 is a widely used and powerful plotting library for R. It is not specifically geared towards mapping, but one can generate great maps.

As we have learned from previous sections, the `ggplot()` syntax is different from the previous as a plot is built up by adding components with a `+`. You can start with a layer showing the raw data then add layers of annotations and statistical summaries. This allows to easily superimpose either different visualizations of one dataset (e.g. a scatterplot and a fitted line) or different datasets (like different layers of the same geographical area)[4].

For an introduction to `ggplot` check out this book by the package creator or this for more pointers.

In order to build a plot you start with initializing a ggplot object. In order to do that `ggplot()` takes:

- a data argument usually a **dataframe** and
- a mapping argument where x and y values to be plotted are supplied.

In addition, minimally a geometry to be used to determine how the values should be displayed. This is to be added after an `+`.

```
ggplot(data = my_data_frame, mapping = aes(x = name_of_column_with_x_value, y = name_of_column_wi
  geom_point()
```

Or shorter:

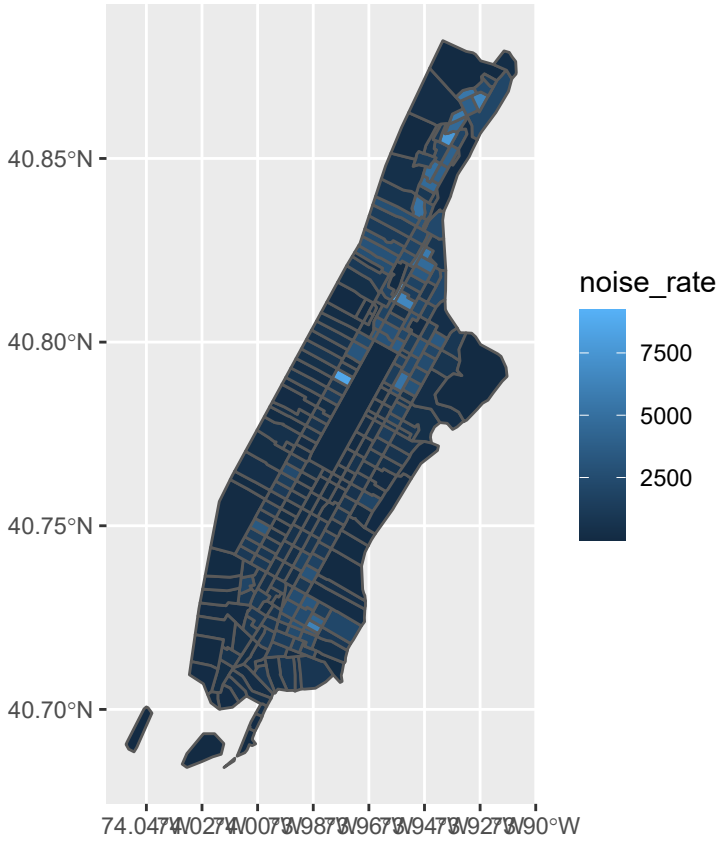
[4]See Wilkinson L (2005): "The grammar of graphics". Statistics and computing, 2nd ed. Springer, New York.

```
ggplot(my_data_frame, aes(name_of_column_with_x_value, name_of_column_with_y_value)) +
  geom_point()
```

3.3.1 Basic *sf* plotting using ggplot

The great news is that **ggplot can plot *sf* objects directly** by using `geom_sf`. So all we have to do is:

```
ggplot(man_noise_rate_sf) +
  geom_sf(aes(fill=noise_rate))
```



Homicide rate is a continuous variable and is plotted by `ggplot` as such. If we wanted to plot our map as a 'true' choropleth map we need to convert our continuous variable into a categorical one, according to whichever brackets we want to use.

This requires two steps:

- Determine the breaks with a "style", i.e. the method or algorithm used to categorize data.
- Add a categorical variable to the object which assigns each continuous value to a bracket.

We will use the `classInt` package to explicitly determine the breaks.

```
require(classInt)

# get quantile breaks. Add .00001 offset to catch the lowest value
breaks_qt <- classIntervals(c(min(man_noise_rate_sf$noise_rate) - .00001,
                              man_noise_rate_sf$noise_rate), n = 7, style = "quantile")
```

```
breaks_qt
```

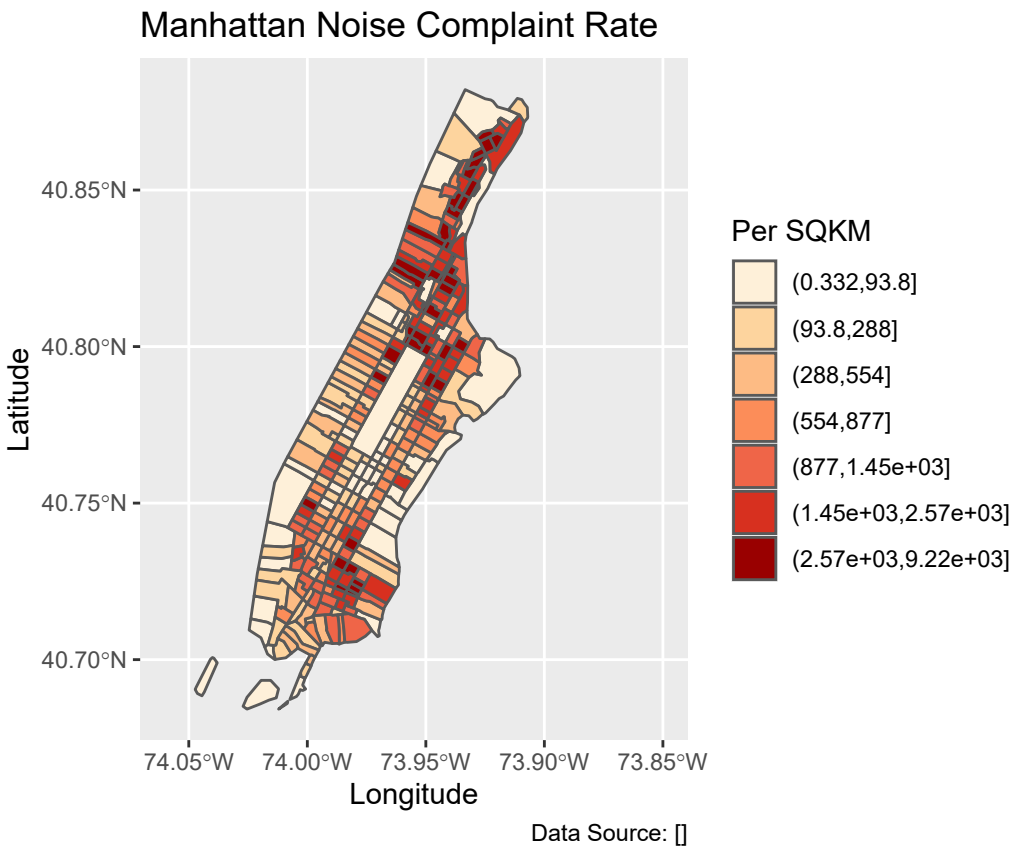
```
#> style: quantile
#> [0.3316134,93.8384)  [93.8384,288.3639)  [288.3639,553.946)  [553.946,877.4583)
#>                  42                   41                   41                   41
#> [877.4583,1452.304) [1452.304,2567.292) [2567.292,9217.026]
#>                  41                   41                   42
```

OK. We can retrieve the breaks with `breaks$brks`.

We use `cut` to divide `noise_rate` into intervals and code them according to which interval they are in.

Lastly, we can use `scale_fill_brewer` and add our color palette.

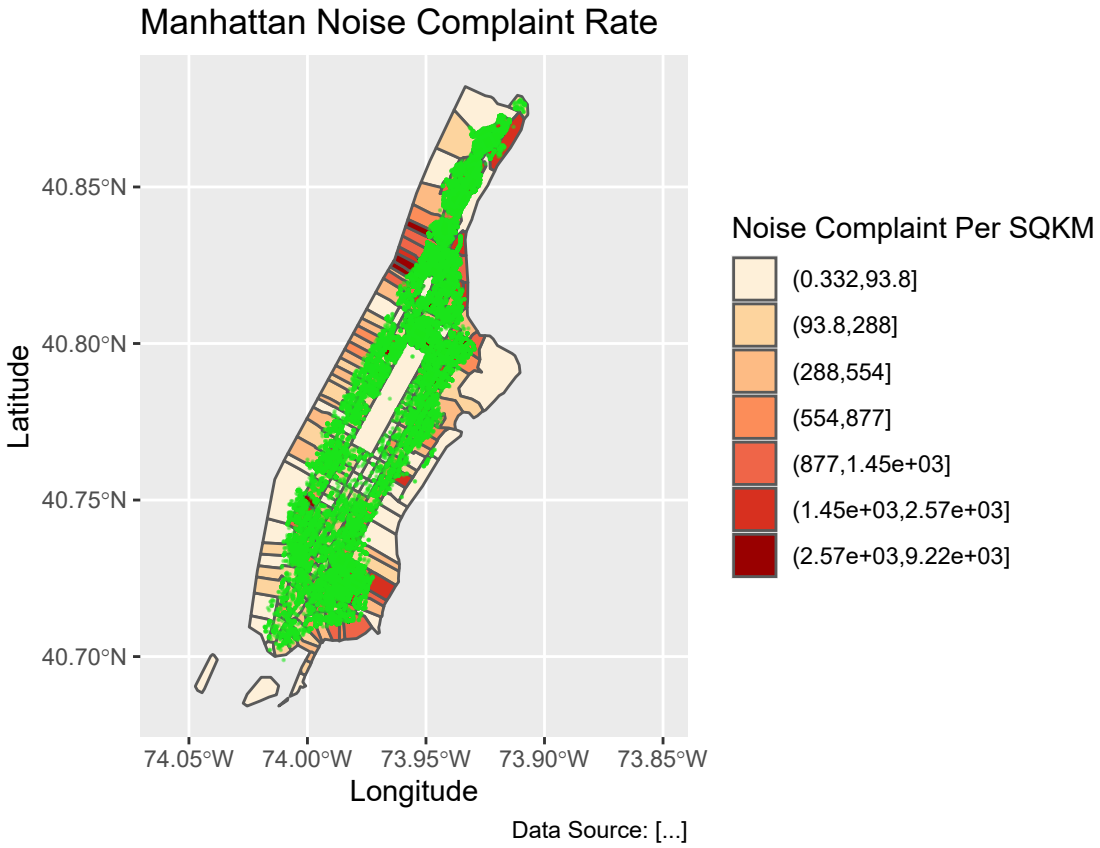
```
man_noise_rate_sf <- mutate(man_noise_rate_sf, noise_rate_cat = cut(noise_rate, breaks_qt$brks))

ggplot(man_noise_rate_sf) +
  geom_sf(aes(fill=noise_rate_cat)) +
  scale_fill_brewer(palette = "OrRd", name='Per SQKM') +
  coord_sf(xlim = c(-74.06, -73.85), default_crs = sf::st_crs(4326) ) +
  labs(x='Longitude', y='Latitude',
       title='Manhattan Noise Complaint Rate',
       caption = 'Data Source: []');
```



3.3.2 Multi-layer plotting

Similarly, we can add multiple layers to the `ggplot`.

```
ggplot(man_noise_rate_sf) +
  geom_sf(aes(fill=noise_rate_cat)) +
  geom_sf(data=man_noises_pt, colour=ptColor, size=0.1) +
  scale_fill_brewer(palette = "OrRd", name='Noise Complaint Per SQKM') +
  coord_sf(xlim = c(-74.06, -73.85), default_crs = sf::st_crs(4326) ) +
  labs(x='Longitude', y='Latitude',
       title='Manhattan Noise Complaint Rate',
       caption = 'Data Source: [...]');
```



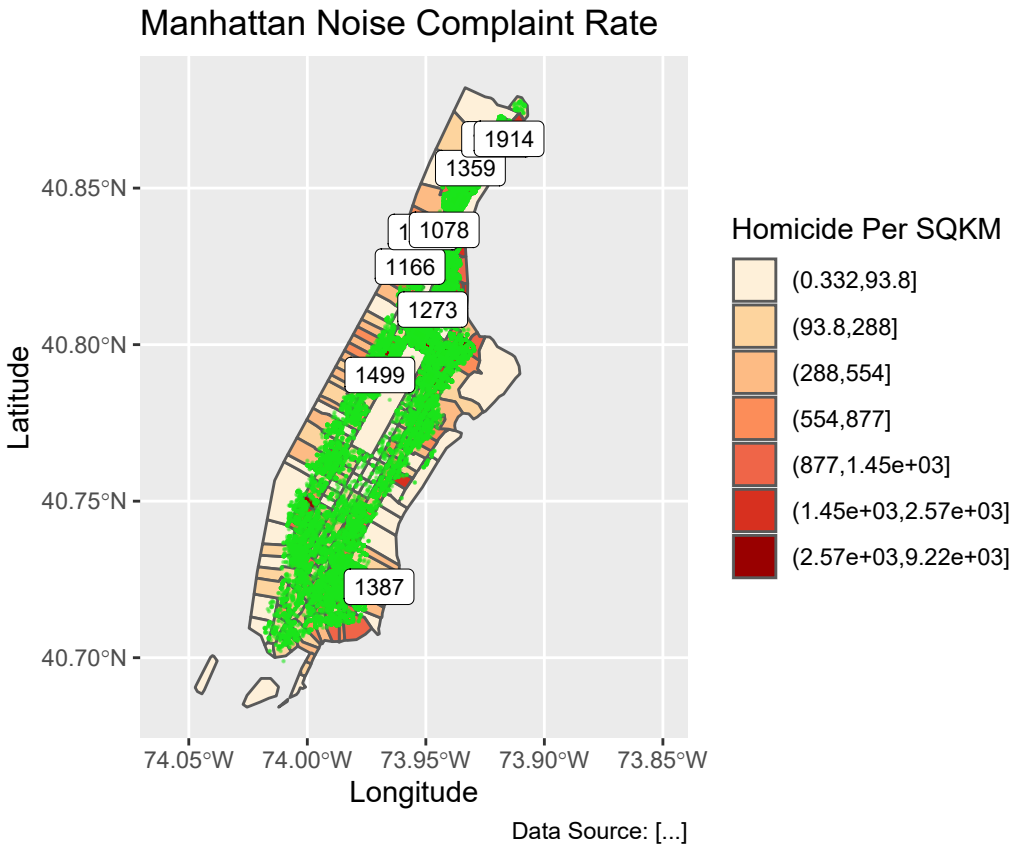
3.3.3 Label spatial objects

With `geom_sf_text` and `geom_sf_label` functions, we can add labels to mapped spatial objects. `geom_sf_label` draws a rectangle behind the text, therefore making good contrast. As in other GIS software, labeling is complicated because there would be many conflicts among labels. The results of automatic labeling is rarely satisfactory and up to the standard of cartography or academic publication. Even though packages like `ggrepel` greatly help avoid overlapping labels, manual adjustments are often necessary and preferred.

```
ggplot(man_noise_rate_sf) +
  geom_sf(aes(fill=noise_rate_cat)) +
  geom_sf(data=man_noises_pt, colour=ptColor, size=0.1) +
  geom_sf_label(data = man_noise_rate_sf %>% dplyr::filter(n_noise > 1000),
                aes(label = n_noise),
```

```
                label.size = .09,
                size = 3) +
  scale_fill_brewer(palette = "OrRd", name='Homicide Per SQKM') +
  coord_sf(xlim = c(-74.06, -73.85), default_crs = sf::st_crs(4326) ) +
  labs(x='Longitude', y='Latitude',
       title='Manhattan Noise Complaint Rate',
       caption = 'Data Source: [...]');
```

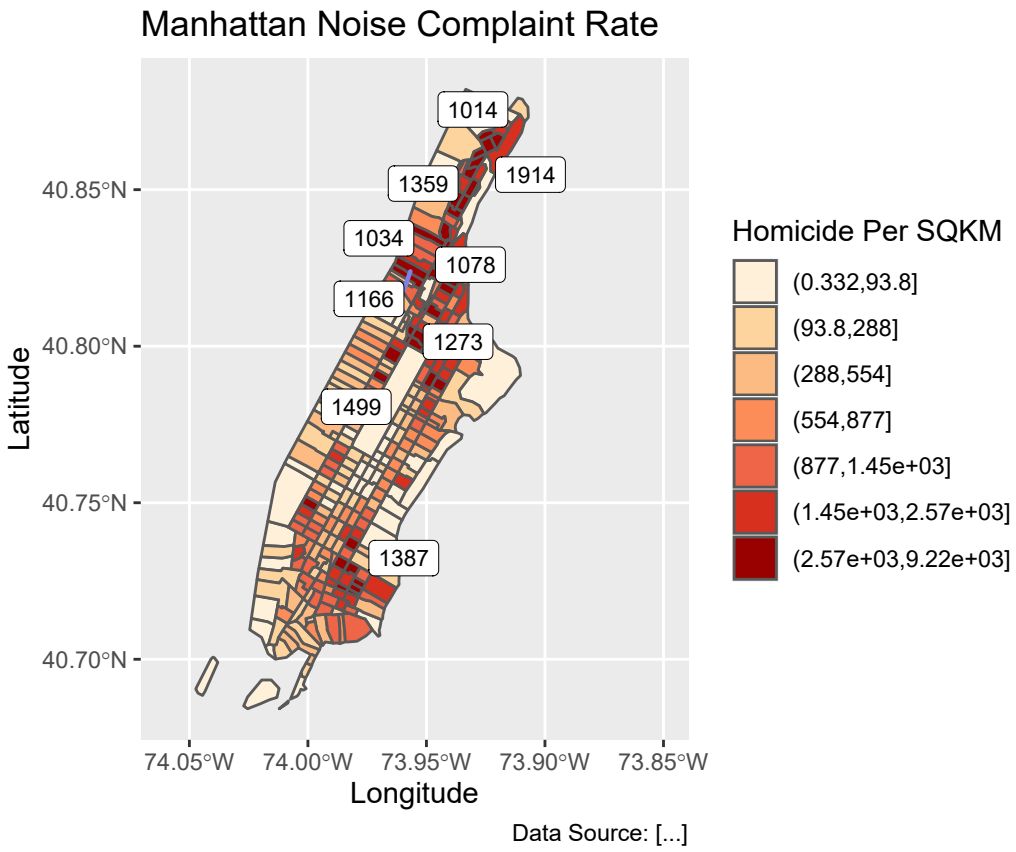
```
#> Warning in st_point_on_surface.sfc(sf::st_zm(x)): st_point_on_surface may not
#> give correct results for longitude/latitude data
```



Using `ggrepel` helps disperse those labels.

```
require(ggrepel);
labelCoords <- man_noise_rate_sf %>% sf::st_centroid() %>%
  dplyr::filter(n_noise > 1000) %>%
  sf::st_coordinates();

labelData <- man_noise_rate_sf %>% sf::st_centroid() %>%
  dplyr::filter(n_noise > 1000) %>%
  dplyr::mutate(x = labelCoords[,1], y=labelCoords[,2])


ggplot(man_noise_rate_sf) +
  geom_sf(aes(fill=noise_rate_cat)) +
  #geom_sf(data=man_noises_pt, colour=ptColor, size=0.1) +
```

```
geom_label_repel(data = labelData,
                 aes(x=x,y=y,label = n_noise),
                 label.size = .09,
                 size = 3,
                 segment.color = rgb(0.5,0.5,0.9),
                 segment.size = 0.8) +
scale_fill_brewer(palette = "OrRd", name='Homicide Per SQKM') +
coord_sf(xlim = c(-74.06, -73.85), default_crs = sf::st_crs(4326) ) +
labs(x='Longitude', y='Latitude',
     title='Manhattan Noise Complaint Rate',
     caption = 'Data Source: [...]');
```



3.3.4 Adding basemaps with ggmap

`ggmap` builds on `ggplot` and allows to pull in tiled basemaps from different services, like Google Maps, OpenStreetMaps, or Stamen Maps[5].

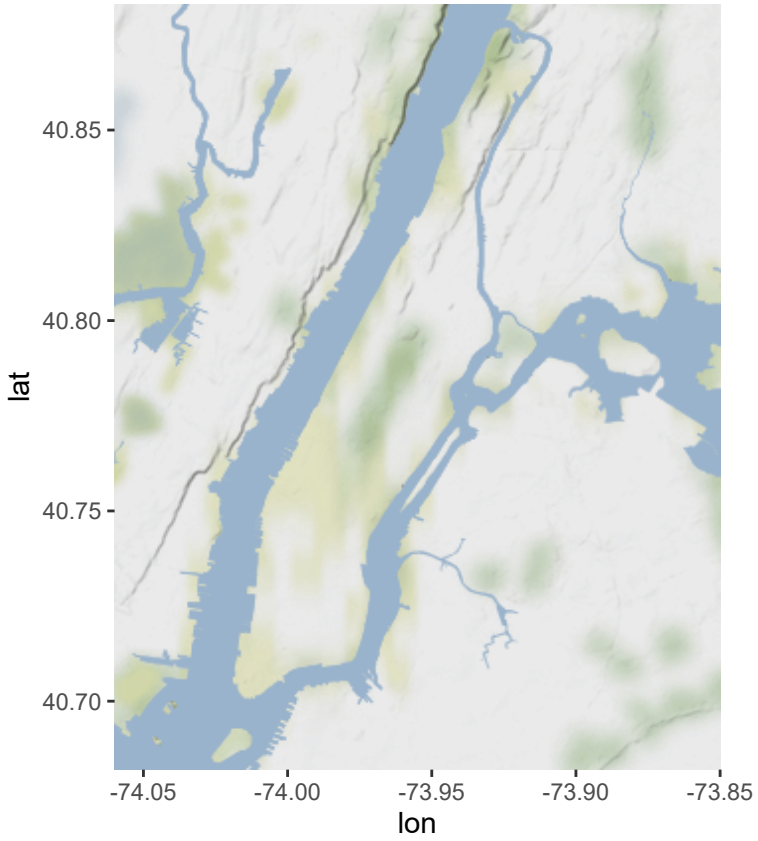
So let's overlay the map from above on a terrain map we pull from Stamen.

If you have a valid Google Maps API key, the following method still works. First we use the `get_map()` command from `ggmap` to pull down the basemap. We need to tell it the location or the boundaries of the map, the zoom level, and what kind of map service we like (default is Google terrain). It will actually download the tile. `get_map()` returns a ggmap object, name it `man_basemap`. In order to view the map we then use `ggmap()`.

[5]Google now requires an API key. Cloudmade maps retired its API so it is no longer possible to be used as basemap. `RgoogleMaps` is another library that provides an interface to query the Google server for static maps.

Currently, only the Stamen Maps allows R code to retrieve base maps without API keys. And we need to specifically call the function `get_stamenmap` to acquire the basemap. If we plan to make multiples maps for the same area, it is better to save it to a variable and reuse it.

```
require(ggmap)

# We can get a simple basemap. Adjust zoom with the map extent.
# Higher zoom means more details but needs more time to load.
# Start from small and increase to an appropriate level. Will learn more on this.

# In most cases we can use the extent of the data to retrieve the basemap
# Note those services require boundary in geographic coordinates
# and we also want to expand a little bit to completely contain the data ranges.
mapBound <- man_noise_rate_sf %>% sf::st_transform(4326) %>%
  st_bbox() %>% st_as_sfc() %>% st_buffer(0.02) %>%
  st_bbox() %>% as.numeric()

# but for the long shape of Manhattan, it might be better to manually choose the basemap
mapBound <- c(-74.06, 40.682, -73.85, 40.883)

man_basemap <- ggmap::get_stamenmap(bbox = mapBound, zoom = 11, messaging = FALSE, maptype = 'ter

ggmap(man_basemap)
```



Then we can reuse the code from the ggplot example above, just replacing the first line, where we initialized a ggplot object above

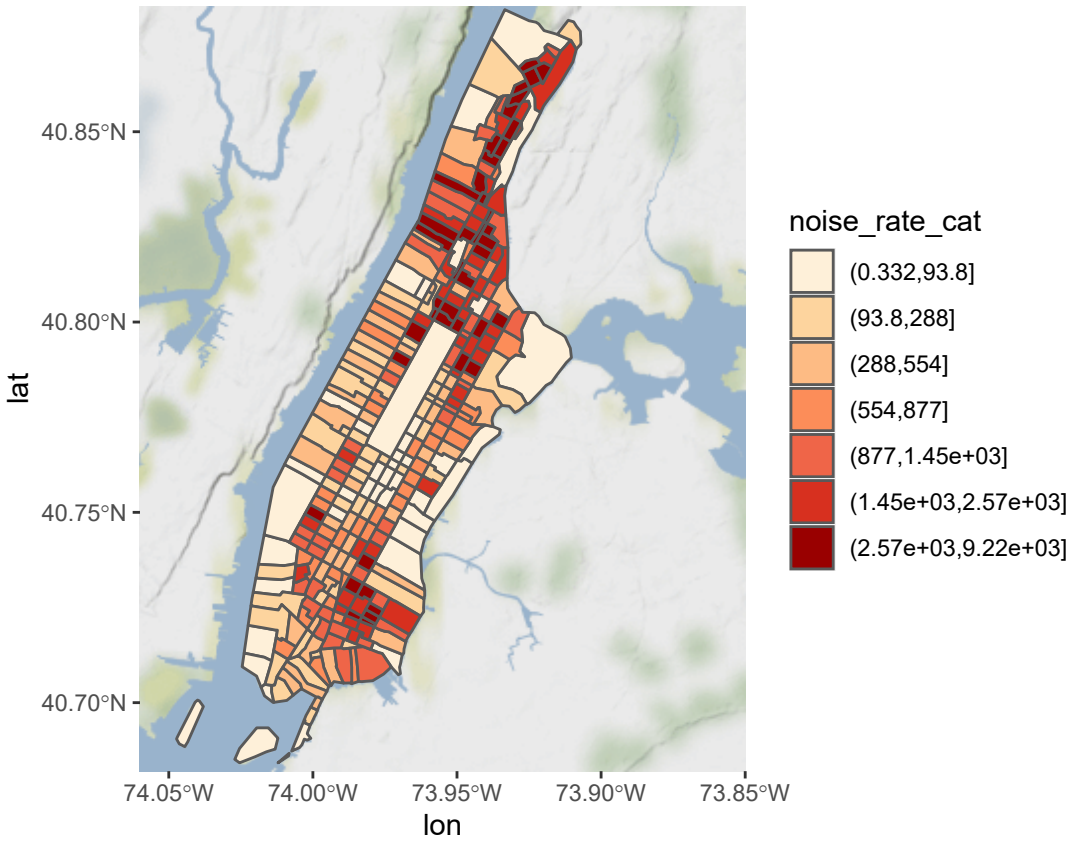
```
    ggplot() +
```

with the line to call our basemap:

```
    ggmap(man_basemap) +
```

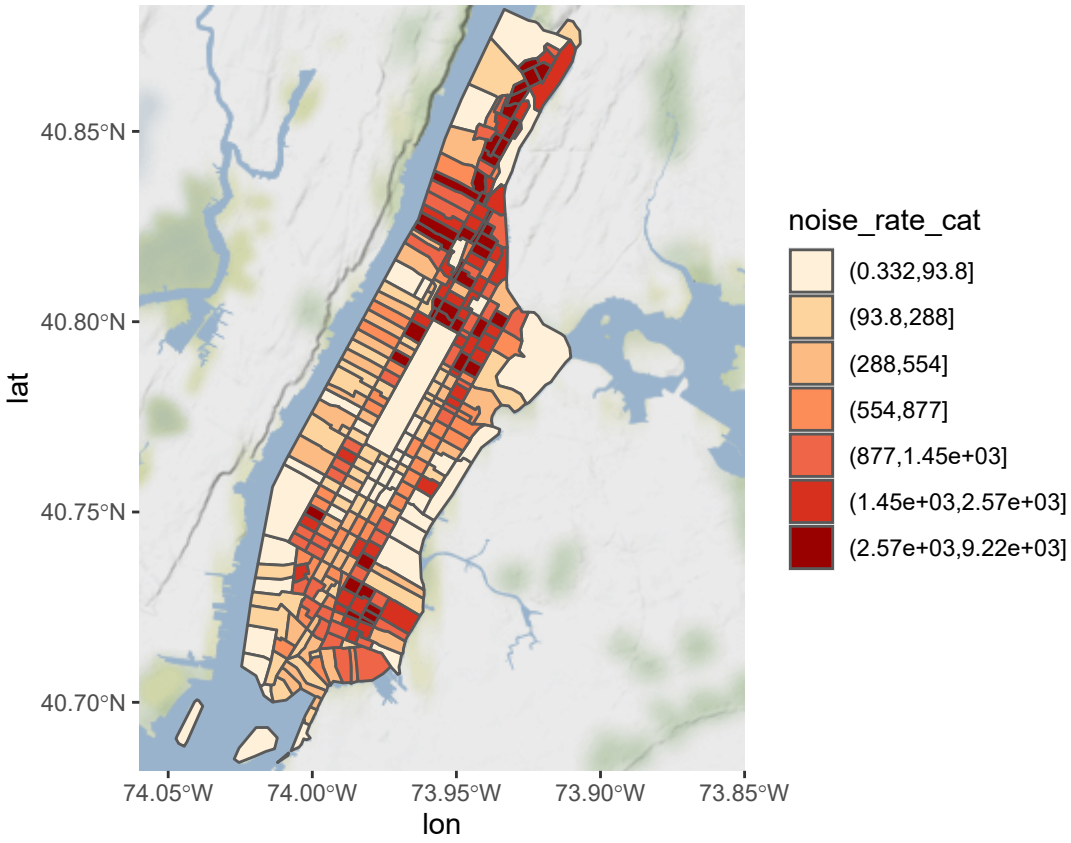
We also have to set `inherit.aes` to `FALSE`, so it overrides the default aesthetics (from the ggmap object).

```
ggmap(man_basemap) +
  geom_sf(data = man_noise_rate_sf, aes(fill=noise_rate_cat), inherit.aes = FALSE) +
  scale_fill_brewer(palette = "OrRd")
```



Note that if the data are in a local map projection, We need to set our CRS to WGS84, which is the one the tiles are downloaded in. We can add `coord_sf` to do this:

```
ggmap(man_basemap) +
    geom_sf(data = man_noise_rate_sf, aes(fill=noise_rate_cat), inherit.aes = FALSE) +
    scale_fill_brewer(palette = "OrRd") +
    coord_sf(crs = st_crs(4326))
```



The ggmap package also includes functions for distance calculations, geocoding, and calculating routes.

3.3.5 Arrange and export plots

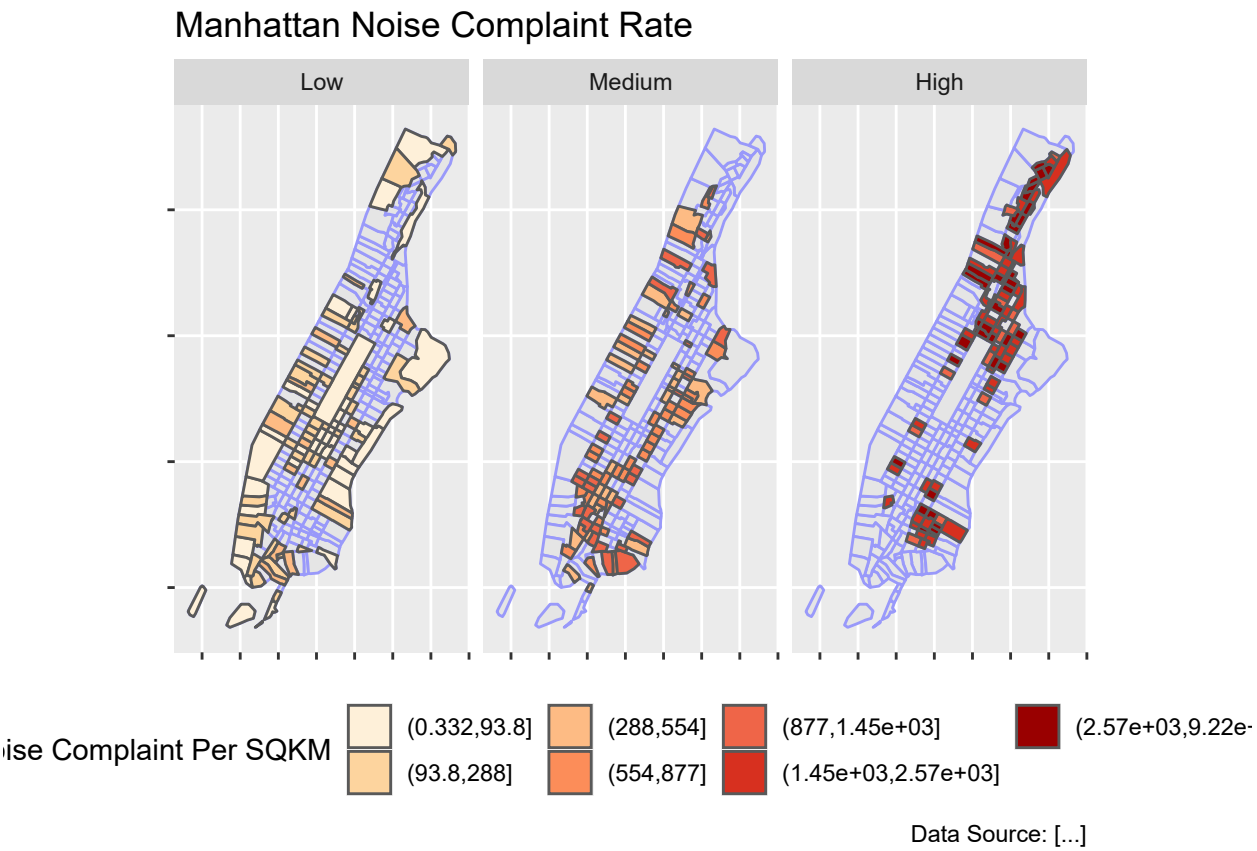
As we have learned before, we can plot these maps to an external file using ggsave or ggexport. With ggarrange, we can also make a nice layout of multiple maps. We can also create a better defined device first and the plot maps on that device.

```
g1 <- ggmap(man_basemap) +
    geom_sf(data = man_noise_rate_sf, aes(fill=noise_rate_cat), inherit.aes = FALSE) +
    scale_fill_brewer(palette = "OrRd") +
    coord_sf(crs = st_crs(4326))

g2 <- ggplot(man_noise_rate_sf) +
  geom_sf(aes(fill=noise_rate_cat)) +
  geom_sf(data=man_noises_pt, colour=ptColor, size=0.1) +
  scale_fill_brewer(palette = "OrRd", name='Noise Complaint Per SQKM') +
  coord_sf(xlim = c(-74.06, -73.85), default_crs = sf::st_crs(4326) ) +
  labs(x='Longitude', y='Latitude',
       title='Manhattan Noise Complaint Rate',
       caption = 'Data Source: [...]');

#============================================================
# does not run
pdf(file='./img/sf_plot_export.pdf', width = 6.5, height = 8.0);
```

```
#plot
g1; g2;
# Turn off the PDF device
dev.off()
#============================================================
require(ggpubr)

ggarrange(g1, g2, nrow = 2, ncol = 1) %>%
  ggexport(filename = file.path(getwd(), 'img/sf_plot_export_2.pdf'),
           width=6.50, height=8.00, # the unit is inch
           pointsize = 9);
```

For *sf* object, the facet also works quite well for comparison purposes.

```
breaks_lmh <- classIntervals(man_noise_rate_sf$noise_rate, n = 3, style = "quantile")
breaks_lmh$brks[1] %<>% magrittr::subtract(0.0001)
man_noise_rate_sf %<>% mutate(noiseLevel = cut(noise_rate,
                                               breaks_lmh$brks,
                                               labels = c('Low','Medium','High')))

ggplot(man_noise_rate_sf) +
  geom_sf(data = man_noise_rate_sf %>% st_geometry(), col = rgb(0.6,.6,.98)) +
  geom_sf(aes(fill=noise_rate_cat)) +
  #geom_sf(data=man_noises_pt, colour=ptColor, size=0.1) +
  scale_fill_brewer(palette = "OrRd", name='Noise Complaint Per SQKM') +
  labs(title='Manhattan Noise Complaint Rate',
       caption = 'Data Source: [...]') +
  facet_wrap(~noiseLevel)+
  theme(axis.text.x = element_blank(),
        axis.text.y = element_blank(),
        legend.position = 'bottom');
```



3.4 Choropleth with *tmap*

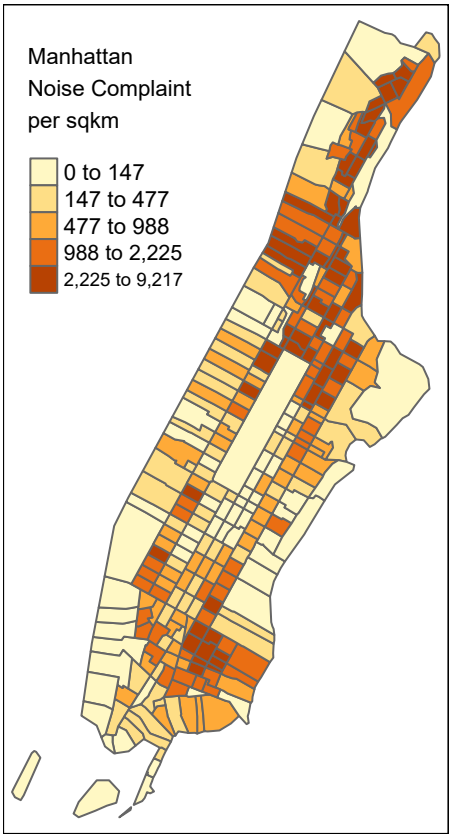
tmap is specifically designed to make creation of thematic maps more convenient. It borrows from the ggplot syntax and takes care of a lot of the styling and aesthetics. This reduces our amount of code significantly. We only need:

- `tm_shape()` where we provide
  - the *sf* object (we could also provide an `SpatialPolygonsDataframe`)
- `tm_polygons()` where we set
  - the attribute variable to map,
  - the break style, and
  - a title.

tmap is still not working well on Mac computers due to some incompatibility issues in some underlying packages/frameworks. Try to use a PC for this part. Or go to the next part.

```
library(tmap)
tm_shape(man_noise_rate_sf) +
  tm_polygons("noise_rate",
              style="quantile",
              title="Manhattan \nNoise Complaint \nper sqkm")
```

```
#> Some legend labels were too wide. These labels have been resized to 0.58. Increase legend.widt
```



tmap has a very nice feature that allows us to give basic interactivity to the map. We can switch from "plot" mode into "view" mode and call the last plot, like so:

```
tmap_mode("view")
tmap_last()
```

Cool huh?

The *tmap* library also includes functions for simple spatial operations, geocoding and reverse geocoding using OSM. For more check `vignette("tmap-getstarted")`.

3.5 Web mapping with *leaflet*

leaflet provides bindings to the 'Leaflet' JavaScript library, "the leading open-source JavaScript library for mobile-friendly interactive maps". We have already seen a simple use of leaflet in the *tmap* example.

The good news is that the *leaflet* library gives us loads of options to customize the web look and feel of the map.

The bad news is that the *leaflet* library gives us loads of options to customize the web look and feel of the map.

Let's build up the map step by step.

First we load the *leaflet* library. Use the `leaflet()` function with an *sp* or ___Spatial___ *object and pipe it to* `addPolygons()` *function. It is not required, but improves readability if you use the pipe operator* `%>%` *to chain the elements together when building up a map with* leaflet*.

And while *tmap* was tolerant about our AEA projection of `man_noise_rate_sf`, *leaflet* does require us to explicitly reproject the *sf* object.

```
library(leaflet)

st_crs(man_noise_rate_sf)
```

```
#> Coordinate Reference System:
#>   User input: WGS 84
#>   wkt:
#> GEOGCRS["WGS 84",
#>     DATUM["World Geodetic System 1984",
#>         ELLIPSOID["WGS 84",6378137,298.257223563,
#>             LENGTHUNIT["metre",1]]],
#>     PRIMEM["Greenwich",0,
#>         ANGLEUNIT["degree",0.0174532925199433]],
#>     CS[ellipsoidal,2],
#>         AXIS["latitude",north,
#>             ORDER[1],
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>         AXIS["longitude",east,
#>             ORDER[2],
#>             ANGLEUNIT["degree",0.0174532925199433]],
#>     ID["EPSG",4326]]
```

```
# reproject, if needed
# man_noise_rate_sf <- st_transform(man_noise_rate_sf, 4326)

leaflet(man_noise_rate_sf) %>%
  addPolygons()
```

To map the homicide density we use `addPolygons()` and:

- remove stroke (polygon borders)
- set a fillColor for each polygon based on `noise_rate` and make it look nice by adjusting fillOpacity and smoothFactor (how much to simplify the polyline on each zoom level). The fill color is generated using *leaflet*'s `colorQuantile()` function, which takes the color scheme and the desired number of classes. To constuct the color scheme `colorQuantile()` returns a function that we supply to `addPolygons()` together with the name of the attribute variable to map.
- add a popup with the `noise_rate` values. We will create as a vector of strings, that we then supply to `addPolygons()`.

```
pal_fun <- colorQuantile("YlOrRd", NULL, n = 5)

p_tip <- paste0("GEOID: ", man_noise_rate_sf$GEOID);

p_popup <- paste0("<strong>Noise Complaint Density: </strong>",
                  man_noise_rate_sf$noise_rate%>%round(3)%>%format(nsmall = 3),
                  " /sqkm",
```

```
                  " <br/>",
                  "<strong> Number of Noise Complaints: </strong>",
                  man_noise_rate_sf$n_noise,
                  sep="")

leaflet(man_noise_rate_sf) %>%
  addPolygons(
    stroke = FALSE, # remove polygon borders
    fillColor = ~pal_fun(noise_rate), # set fill color with function from above and value
    fillOpacity = 0.8, smoothFactor = 0.5, # make it nicer
    popup = p_popup)  # add popup
```

Here we add a basemap, which defaults to OSM, with `addTiles()`

```
leaflet(man_noise_rate_sf) %>%
  addPolygons(
    stroke = FALSE,
    fillColor = ~pal_fun(noise_rate),
    fillOpacity = 0.8, smoothFactor = 0.5,
    popup = p_popup) %>%
  addTiles()
```

Lastly, we add a legend. We will provide the `addLegend()` function with:

- the location of the legend on the map
- the function that creates the color palette
- the value we want the palette function to use
- a title

```
leaflet(man_noise_rate_sf) %>%
  addPolygons(
    stroke = FALSE,
    fillColor = ~pal_fun(noise_rate),
    label = p_tip,
    fillOpacity = 0.8, smoothFactor = 0.5,
    popup = p_popup) %>%
  addTiles() %>%
  addLegend("bottomright",  # location
            pal=pal_fun,    # palette function
            values=~noise_rate,  # value to be passed to palette function
            title = 'Manhattan Noise Complaints <br> per sqkm') # legend title
```

The labels of the legend show percentages instead of the actual value breaks[6].

To set the labels for our breaks manually we replace the `pal` and `values` with the `colors` and `labels`

[6]The formatting is set with `labFormat()` and in the documentation we discover that: "By default, `labFormat` is basically `format(scientific = FALSE,big.mark = ',')` for the numeric palette, `as.character()` for the factor palette, and a function to return labels of the form `x[i] - x[i + 1]` for bin and quantile palettes (**in the case of quantile palettes, x is the probabilities instead of the values of breaks**)."

arguments and set those directly using `brewer.pal()` and `breaks_qt` from an earlier section above.

```
leaflet(man_noise_rate_sf) %>%
  addPolygons(
    stroke = FALSE,
    fillColor = ~pal_fun(noise_rate),
    fillOpacity = 0.8, smoothFactor = 0.5,
    label = p_tip,
    popup = p_popup) %>%
  addTiles() %>%
  addLegend("bottomright",
            colors = brewer.pal(7, "YlOrRd"),
            labels = paste0("up to ", format(breaks_qt$brks[-1], digits = 2)),
            title =  'Manhattan Noise Complaints <br> per sqkm')
```

That's more like it. Finally, a layer control is added to switch to another basemap and turn the polygons off and on. Take a look at the changes in the code below. At the same time, the polygon layer and its legend have the group name. As such, both can be turned on and off at the same time.

```
polyHighlightOption <- leaflet::highlightOptions(opacity = 1.0, fillColor = 'black')
polyLabelOption <- leaflet::labelOptions(opacity = 0.6)

htmlMap <- leaflet(man_noise_rate_sf) %>%
  addPolygons(
    stroke = FALSE,
    fillColor = ~pal_fun(noise_rate),
    fillOpacity = 0.8, smoothFactor = 0.5,
    popup = p_popup,
    group = "manhattan",
    label = p_tip,
    highlightOptions = polyHighlightOption,
    labelOptions = polyLabelOption) %>%
  addTiles(group = "OSM") %>%
  addProviderTiles("CartoDB.DarkMatter", group = "Carto") %>%
  addLegend("bottomright",
            group = "nyc",
            colors = brewer.pal(7, "YlOrRd"),
            labels = paste0("up to ", format(breaks_qt$brks[-1], digits = 2)),
            title = "Manhattan <br> Noise Complaints <br>(/sqkm)") %>%
  addLayersControl(baseGroups = c("OSM", "Carto"),
                   overlayGroups = c("nyc"))

htmlMap
```

Such interactive maps can be saved using either `htmlwidgets::saveWidget()` or `mapview::mapshot()` The `htmlwidgets::saveWidget` method can also save *tmap* output.

```
# A single file
htmlwidgets::saveWidget(htmlMap, 'nyc_noise_leaflet_widget.html')
# A HTML file with underlying libraries
htmlwidgets::saveWidget(htmlMap, 'nyc_noise_leaflet.html',
                        selfcontained = FALSE, libdir = 'widget-lib')
```

```
# Using mapview: which is exactly as the self-contained html from htmlwidges.
# They use the same package and method under the hood.
mapview::mapshot(htmlMap, 'nyc_noise_mapview.html')
```

If you'd like to take this further here are a few pointers.

- Leaflet for R
- Creating maps in R
- Maps in R

Here is an example using `ggplot`, *leaflet*, `shiny`, and RStudio's flexdashboard template to bring it all together.

3.6 Animated maps

One useful column in the noise point data is the reported time, which can be used to generated an animated map to show the dynamics over time.

```
require(gganimate)

# Let's show one day's noise complaints
baseMapExt <- c(-74.06, 40.691, -73.85, 40.885)

# First we create a regular ggplot
ggmap::get_stamenmap(baseMapExt, zoom = 12, messaging = FALSE, maptype = 'terrain-background
```

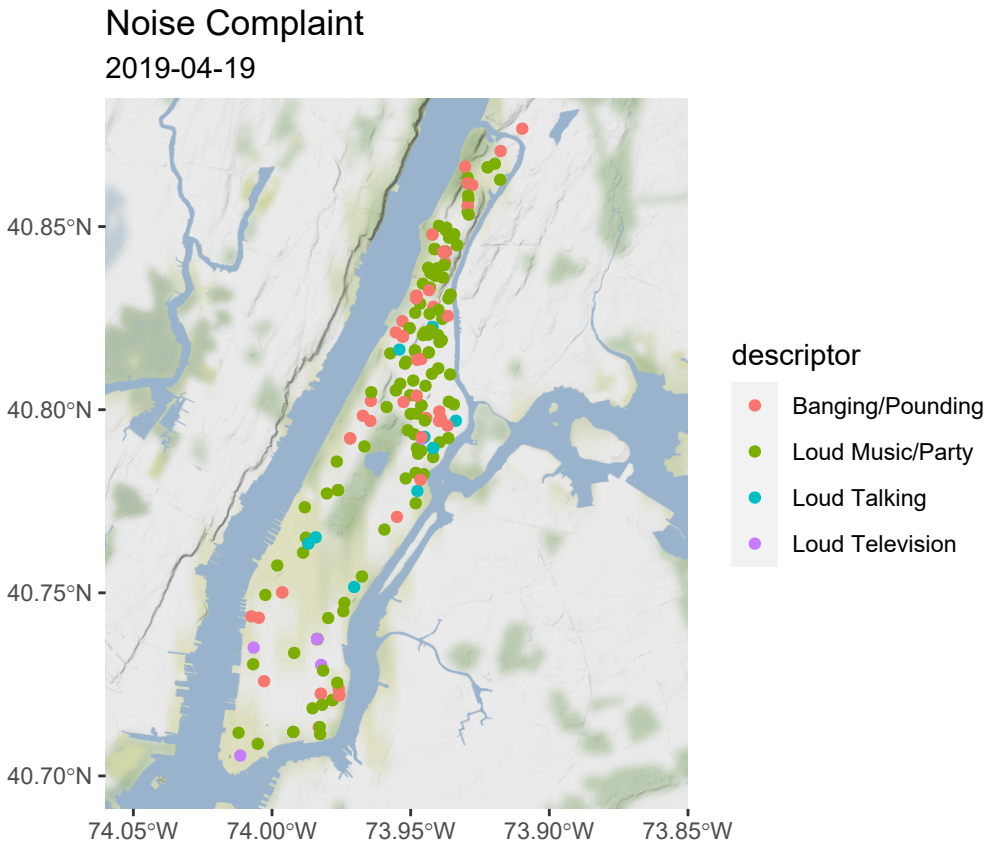
```
#> Source : http://tile.stamen.com/terrain-background/12/1205/1537.png
#> Source : http://tile.stamen.com/terrain-background/12/1206/1537.png
#> Source : http://tile.stamen.com/terrain-background/12/1207/1537.png
#> Source : http://tile.stamen.com/terrain-background/12/1205/1538.png
#> Source : http://tile.stamen.com/terrain-background/12/1206/1538.png
#> Source : http://tile.stamen.com/terrain-background/12/1207/1538.png
#> Source : http://tile.stamen.com/terrain-background/12/1205/1539.png
#> Source : http://tile.stamen.com/terrain-background/12/1206/1539.png
#> Source : http://tile.stamen.com/terrain-background/12/1207/1539.png
#> Source : http://tile.stamen.com/terrain-background/12/1205/1540.png
#> Source : http://tile.stamen.com/terrain-background/12/1206/1540.png
#> Source : http://tile.stamen.com/terrain-background/12/1207/1540.png
```

```
man_day_noise_sf <- st_read(dsn = './data/nyc/man_data.gpkg',
        layer='manhattan_noise_one_day')

man_noise_anim <- man_day_noise_sf %>%
                mutate(datetime = lubridate::ceiling_date(datetime, 'minutes')) %>%
                arrange(datetime)  # sort the data by time
```

```
ggmap(baseMap) +
      geom_sf(data = man_noise_anim,
              aes(color = descriptor, group = datetime),
              show.legend = TRUE,
              inherit.aes = FALSE) +
      labs(title='Noise Complaint', subtitle = '2019-04-19') +
      theme(axis.title.x=element_blank(),
            axis.title.y = element_blank()) -> p1
```

```
#> Coordinate system already present. Adding new coordinate system, which will replace the existi
```

```
p1
```




```
# use gganimate package to add transitions
p1 + transition_time(datetime) +
  shadow_mark() +
  #shadow_wake(wake_length = 0.18, size=NULL, alpha = 0, ) +
  #enter_fade() + enter_grow() + exit_fade() +
  ggtitle('Noise Complaint', subtitle = '{frame_time}') -> panim

# Now produce the animation
# define a device to specify the size of the layout
animate(panim, nframes = 180,
        device = 'png', width = 600, height = 600, units = 'px')
```

3.7 Lab Assignment

The third and last R-spatial lab is to visualize data that we assembled during the first two labs. As geovisualization is often exploratory, you are encouraged to be more creative.

Main tasks for the third lab are:

1. Plot at least two high-quality static maps with one using the COVID-19 data and one using a related factor. You can use either `plot` method for *sf* or `ggplot` method.

2. Use *ggplot2* and other ggplot-compatible packages to create a multi-map figure illustrating the possible relationship between COVID-19 confirmed cases or rate and another factor (e.g., the number of nursing homes, number of food stores, neighborhood racial composition, elderly population, etc.). The maps should be put side by side on one single page. Add graticule to at least one of those maps and label some of the feature on the map where applicable and appropriate.
3. Create a web-based interactive map for COIVD-19 data using *tmap* or *leaflet* package and save it as a HTML file.

Although data visualization has many subjective factors, try your best to make the visuals as appealing as possible.

```
#> Warning: package 'maptools' was built under R version 4.1.3
```

```
#> Warning: package 'Rcpp' was built under R version 4.1.3
```

Chapter 4

Spatial Regression in R

Learning Objectives

- Describe Spatial Autocorrelation
- Calculate Global and Local Indicators of Spatial Autocorrelation
- Apply Spatial Error and Spatial Lag Models to address spatial autocorrelation
- Explore Spatial Heterogeneity with Geographically Weighted Regression

---

4.1 Spatial Autocorrelation

Most statistical methods are based on certain assumptions such as that the samples are independent of each other. Geographic phenomena, however, are all related to each other as Waldo R. Tobler's *First Law of Geography* states: *Everything is related to everything else, but near things are more related than distant things.*

> "In spatial data, it is often the case that some or all outcome measures exhibit spatial autocorrelation. This occurs when the relative outcomes of two points is related to the distance between them or two polygons share boundaries. When analyzing spatial data, it is important to check for autocorrelation. If there is no evidence of spatial autocorrelation, then proceeding with a standard approach is acceptable. However, if there is evidence of spatial autocorrelation, then one of the underlying assumptions of standard non-spatial analyses may be violated and the results may not be valid."

"Spatial autocorrelation measures how similar or dissimilar objects are in comparison with close objects or neighbors." Spatial autocorrelation can be measured globally or locally.

4.1.1 Global Indicators of Spatial Autocorrelation

While there are many methods to indicate global spatial autocorrelation, the Moran's I is one of the most widely used.

Moran's I

- Positive spatial autocorrelation
  - values are similar to their neighbors or other close objects
  - clusters of similar values on the map
- Zero or no spatial autocorrelation

  - random values of close objects or neighbors
  - no clear pattern visually
- Negative spatial autocorrelation
  - values are dissimilar to their neighbors or close objects
  - dispersed patterns of values on the map
  - checker board style for polygons

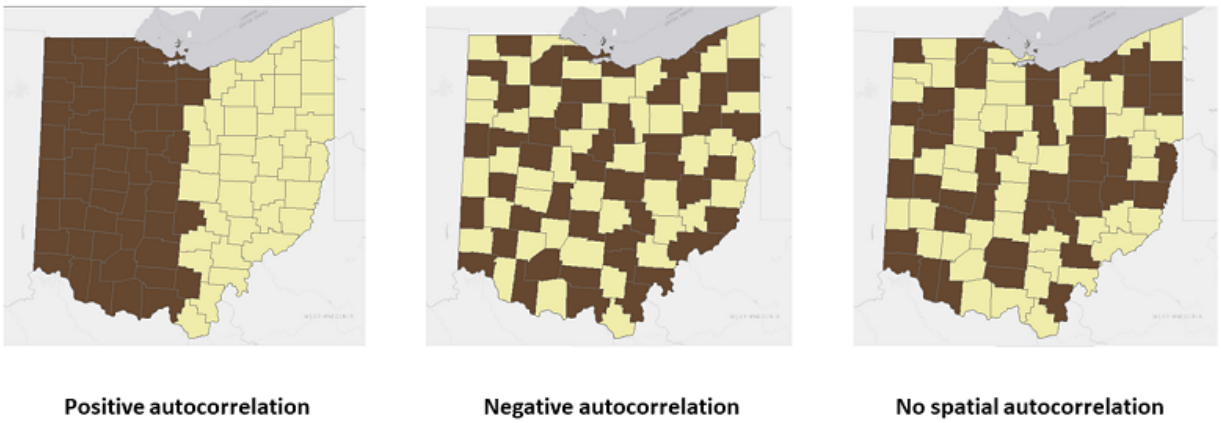
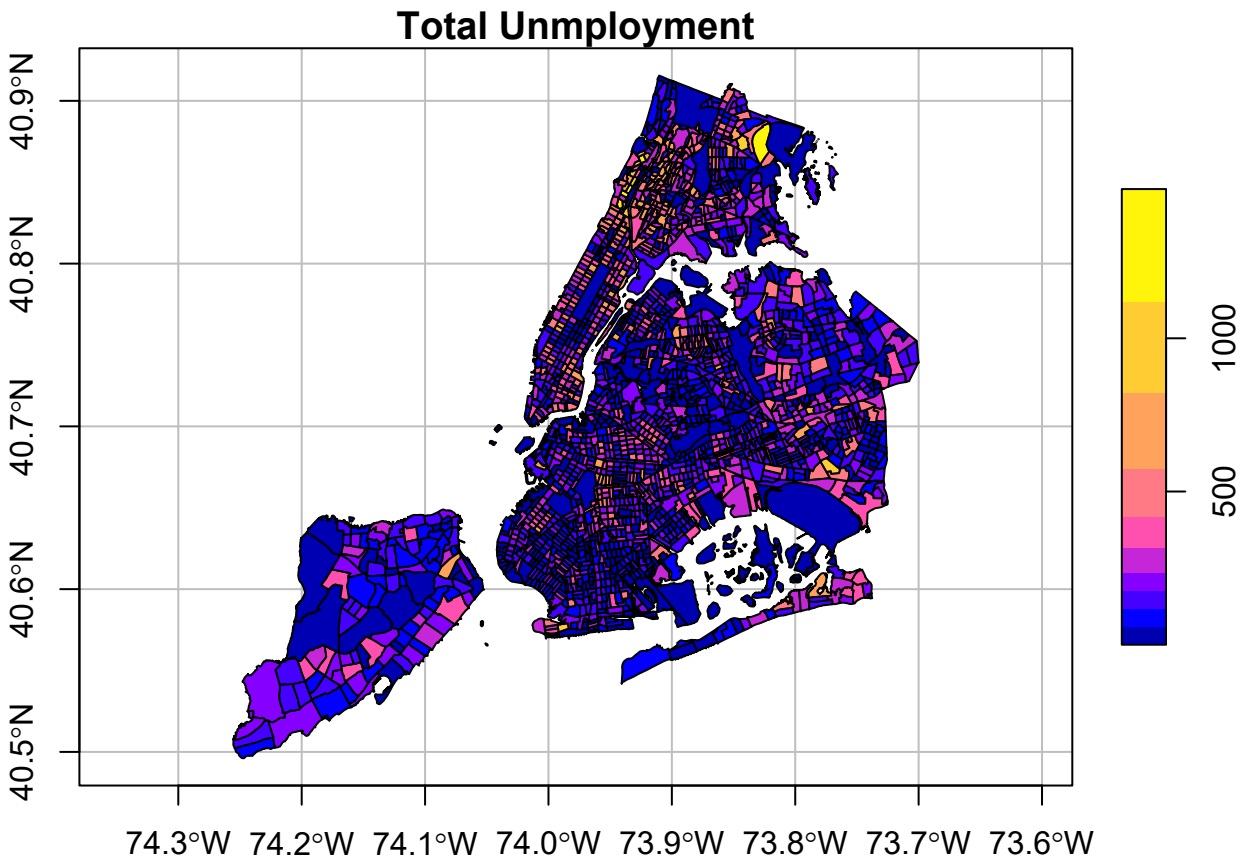


Figure 4.1: Global Spatial Autocorrelation

First, let us load the data into a ***sf*** object. The data is about education and socio-demographic characteristics for New York City Census tracts compiled from American Community Survey 2008-2012, US Census Bureau. It is available at GeoDa Data Repository.

```
# Download and unzip the data. Skip this if you downloaded the data from GeoDa.
#download.file("http://www.geo.hunter.cuny.edu/~ssun/R-Spatial/data/NYC.zip", "NYC.zip");
#unzip("NYC.zip", exdir = "data")

# Read the data
sf::st_read('./data/nyc/NYC_Tract_ACS2008_12.shp') -> nycDat
# Set the CRS ID
sf::st_crs(nycDat) <- 4326;

# Check data
head(nycDat);
str(nycDat);
names(nycDat)

# Simple plot to confirm
plot(nycDat['popunemplo'], graticule= sf::st_crs(4326),
     main='Total Unmployment', breaks="jenks", axes=TRUE)
```



With the spatial object, we can run the global Moran's I test. In order to run that, we need to prepare a list of weights based on neighboring relationships. Essentially, we want to see if the unemployment rate at a census tract is similar to or dissimilar to its neighbors. The `spdep::poly2nb` produces such neighboring relationships and then `spdep::nb2listw` turns them into weights. Depending on the parameters to those functions, neighbors get higher weights and non-neighbors get low or zero weights.

For point type of data, `spdep::knearneigh` and `spdep::knn2nb` can construct the weights for neighbors. And `spdep::grid2nb` can do so for raster data. Alternatively, `spdep::graph2nb` can produce neighbors for point data using graphs.

```
# use spdep package to test (global autocorrelation)
# spdep::moran, spdep::moran.test
# The zero.policy = TRUE allows zero-length weights vectors
nycDat %>% spdep::poly2nb(c('cartodb_id')) %>%
  spdep::nb2listw(zero.policy = TRUE) -> nycNbList

nycNbList %>%
  spdep::moran.test(nycDat$UNEMP_RATE, ., zero.policy = TRUE)
```

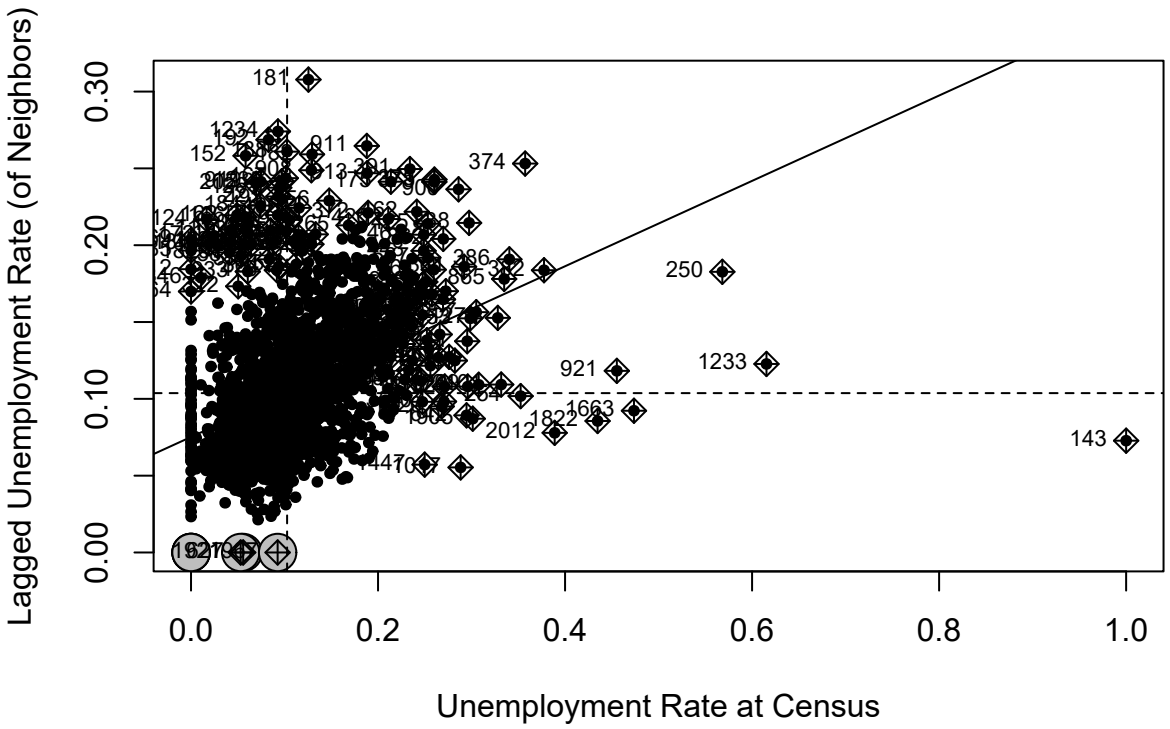
```
#>
#>  Moran I test under randomisation
#>
#> data:  nycDat$UNEMP_RATE
#> weights: .  n reduced by no-neighbour observations
#>
```

```
#>
#> Moran I statistic standard deviate = 22.049, p-value < 2.2e-16
#> alternative hypothesis: greater
#> sample estimates:
#> Moran I statistic       Expectation          Variance
#>      0.2748112287     -0.0004625347      0.0001558661
```

The Moran's I test shows a positive statistic of 0.275. And it is also statistically significant with a near zero p-value. Because the p-value < 0.01, we are 99% confident that the unemployment rates are not spatially independent from those in their neighboring census tracts. And a positive Moran's I implies the unemployment rate of a census tract tends to be similar to its neighboring tracts, which is what we expect.

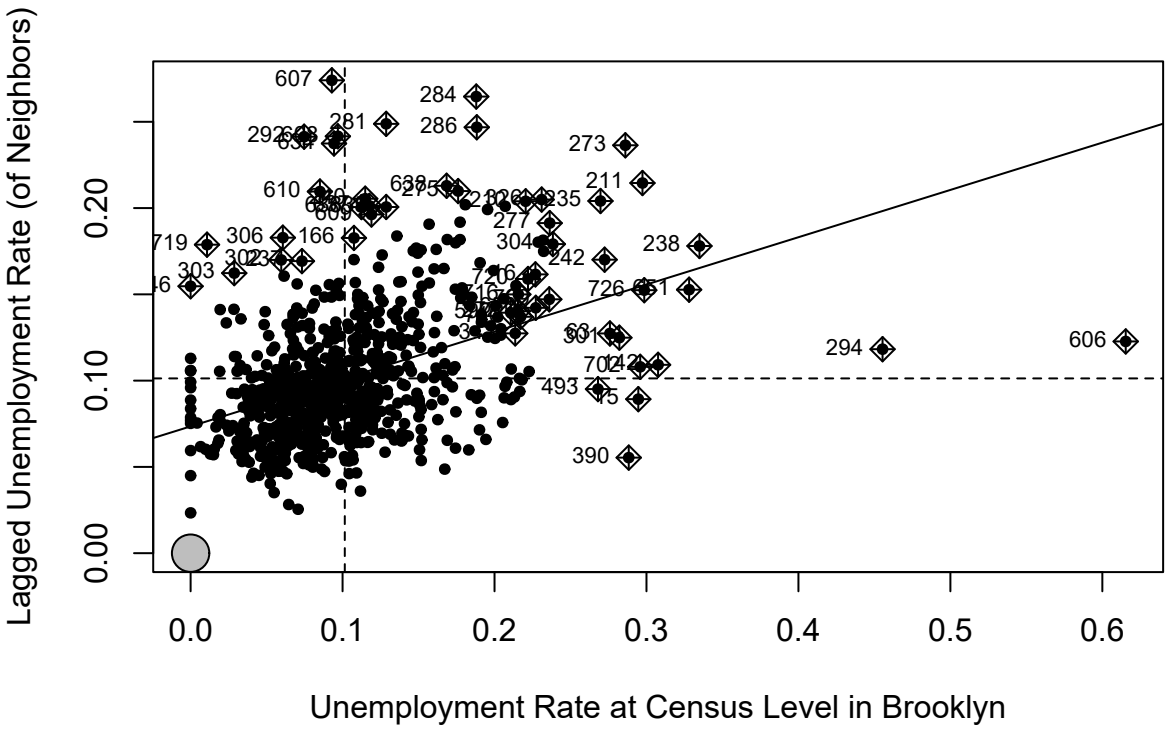
The Moran scatter-plot can offer more details than the single statistic, which also helps us understand the local Moran's I in the next section.

```
spdep::moran.plot(nycDat$UNEMP_RATE,
                  nycNbList,
                  zero.policy = TRUE,
                  xlab = 'Unemployment Rate at Census',
                  ylab = 'Lagged Unemployment Rate (of Neighbors)',
                  pch=20)
```



To better examine the concept, let us make a simpler case by filtering and choosing data in Brooklyn only.

```
nycDat %>% dplyr::filter(boroname == 'Brooklyn') -> brklnDat

brklnDat %>%
  spdep::poly2nb(c('cartodb_id')) %>%
  spdep::nb2listw(zero.policy = TRUE) %>%
  spdep::moran.plot(brklnDat$UNEMP_RATE, .,
                    zero.policy = TRUE,
                    xlab = 'Unemployment Rate at Census Level in Brooklyn',
                    ylab = 'Lagged Unemployment Rate (of Neighbors)',
                    pch=20)
```



On the X axis are the standardized unemployment rate, whose mean value is about 0.1. The Y axis is the spatially lagged unemployment rate, which we can interpret as the unemployment rate around a census tract. Each point in the scatter plot is a census tract. Its X value is its real unemployment rate and Y is the unemployment rate estimated from its neighbors.

The two dashed lines in the figure divide the plot into four quadrants. All the points in the upper right quadrant have higher-than-mean unemployment rates. Their neighbors also have higher-than-mean rates. The points in the bottom left quadrant have lower-than-mean values, both for census tracts and their neighbors. By contrast, the census tracts that fall into the bottom right quadrant have high-than-median values but their neighbors have lower-than-mean values. Similarly, those in the upper left quadrant have lower-than-mean values but their neighbors have high-than-mean values.

The solid line in the plot indicates in the Moran's I statistic. It goes though the mean values and its slope is exactly the Moran's I. Obviously, a upward slope means a positive autocorrelation and

downward slope means a negative autocorrelation.

4.1.2 Local Indicators of Spatial Autocorrelation

To examine spatial autocorrelation locally, we can use Local Moran's I or Local Indicators of Spatial Association (LISA), among others.

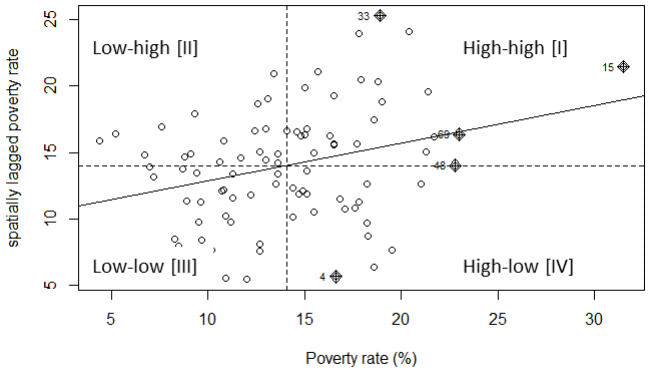


Figure 4.2: Global Spatial Autocorrelation

LISA can help identify clusters of high or low values as well as outliers that are surrounded by opposite values. As explained above, we could identify HH (high values surrounded by high values), LL (low values surrounded by low values), HL (high values surrounded by low values), and LH (Low values surrounded by high values).

```
# use spdep package to test (global autocorrelation)
# spdep::localmoran, spdep::localmoran.exact
# localmoran result:
##  High positive Ii means simlar values (either high or low clusters)
##  Low negative Ii means dissimilar values (outliers)
##

lisaRslt <- spdep::localmoran(nycDat$UNEMP_RATE, nycNbList,
                              zero.policy = TRUE, na.action = na.omit)

# The dimension of LISA result and the orignal sf object
dim(lisaRslt); dim(nycDat);
```

```
#> [1] 2166    5
```

```
#> [1] 2166  114
```

```
# some results
head(lisaRslt)
```

```
#>            Ii          E.Ii     Var.Ii       Z.Ii Pr(z != E(Ii))
#> 1  0.00000000  0.000000e+00 0.00000000        NaN            NaN
#> 2 -0.04952833 -5.118754e-05 0.03692135 -0.2574932     0.79679806
#> 3 -0.43545072 -5.322679e-04 0.23002975 -0.9068090     0.36450780
#> 4  0.65905181 -1.220423e-03 0.87925671  0.7041500     0.48133936
#> 5  0.09599460 -2.501648e-05 0.01352731  0.8255708     0.40904763
```

```
#> 6  0.54067850 -2.494425e-04 0.10783185  1.6472742     0.09950171
```

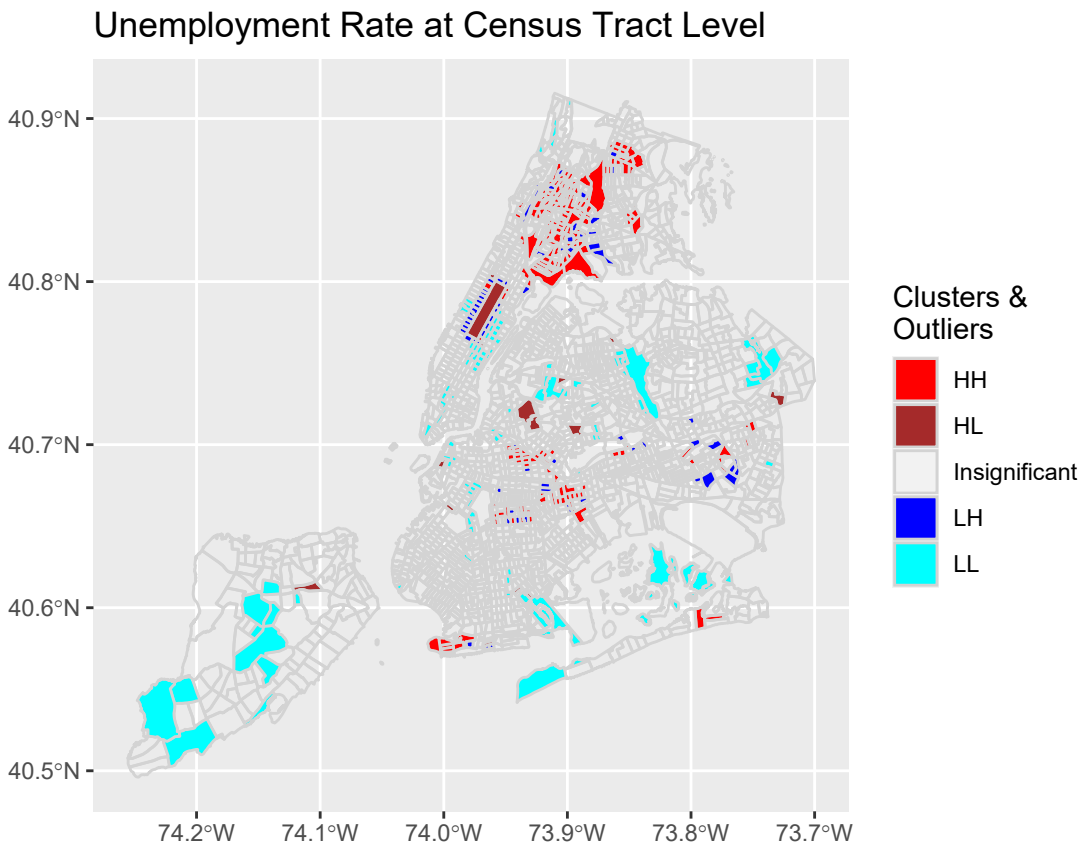
The columns in the LISA results have specific meanings.

Column	Meaning
Ii	local moran statistic
E.Ii	expectation of local moran statistic
Var.Ii	variance of local moran statistic
Z.Ii	standard deviate of local moran statistic
Pr(z > 0)	p-value of local moran statistic

> The combined information allows for a classification of the significant locations as high-high and low-low spatial clusters, and high-low and low-high spatial outliers. It is important to keep in mind that the reference to high and low is relative to the mean of the variable, and should not be interpreted in an absolute sense.
>
> — GeoDa

First, using the `p-value`, we can determine whether we can draw statistically significant conclusions for the existence of clusters or outliers around those spatial features. If the `p-vale >= signficance level` such as `p >= 0.05`, they could be labeled insignificant and we would not know if they are clusters or outliers. For those significant features, that is `p < 0.05`, we need to identify if they are clusters or outliers, which can be identified using `Ii`. If `Ii` is positive, it would be a cluster (similar to nearby or neighboring values); otherwise, it would be a outlier (very different from nearby or neighboring values). Then, it is very easy to determine if it is a high or low value relative to the mean.

```
# Now we can derive the cluster/outlier types (COType in ArcGIS term) for each spatial featu
significanceLevel <- 0.05; # 95% confidence
meanVal <- mean(nycDat$UNEMP_RATE);

lisaRslt %<>% tibble::as_tibble() %>%
  magrittr::set_colnames(c("Ii","E.Ii","Var.Ii","Z.Ii","Pr(z > 0)")) %>%
  dplyr::mutate(coType = dplyr::case_when(
  `Pr(z > 0)` > 0.05 ~ "Insignificant",
  `Pr(z > 0)` <= 0.05 & Ii >= 0 & nycDat$UNEMP_RATE >= meanVal ~ "HH",
  `Pr(z > 0)` <= 0.05 & Ii >= 0 & nycDat$UNEMP_RATE < meanVal ~ "LL",
  `Pr(z > 0)` <= 0.05 & Ii < 0 & nycDat$UNEMP_RATE >= meanVal ~ "HL",
  `Pr(z > 0)` <= 0.05 & Ii < 0 & nycDat$UNEMP_RATE < meanVal ~ "LH"
))

# Now add this coType to original sf data
nycDat$coType <- lisaRslt$coType %>% tidyr::replace_na("Insignificant")

ggplot(nycDat) +
  geom_sf(aes(fill=coType),color = 'lightgrey') +
  scale_fill_manual(values = c('red','brown','NA','blue','cyan'), name='Clusters & \nOutlier
  labs(title = "Unemployment Rate at Census Tract Level")
```



From the plot, it is clear that no outliers are present in the unemployment distribution at 95% confidence level. In other words, if a census tract has high or low unemployment rate, its neighborhoods are unlikely to have an opposite low or high rates. This is probably an indicator of socioeconomic segregation, which is not surprising to local residents.

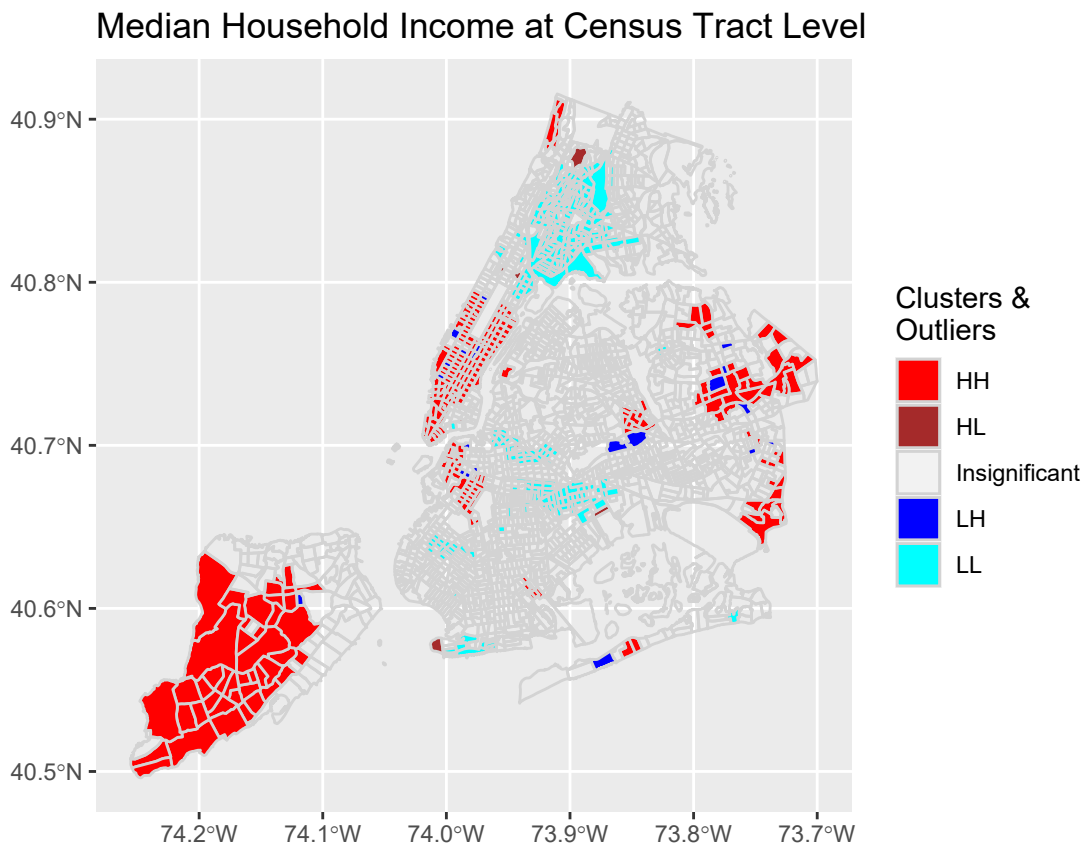
The HH, LL clusters suggest that there are local clusters of high unemployment areas like Bronx and Coney Island. Similarly, there are also clusters of low unemployment areas such as upper west and upper east areas in Manhattan.

Obviously, the local Moran's I or LISA provides more detailed and local information about spatial autocorrelation. More importantly, they can identify those local clusters (hotspots, coldspots) and outliers.

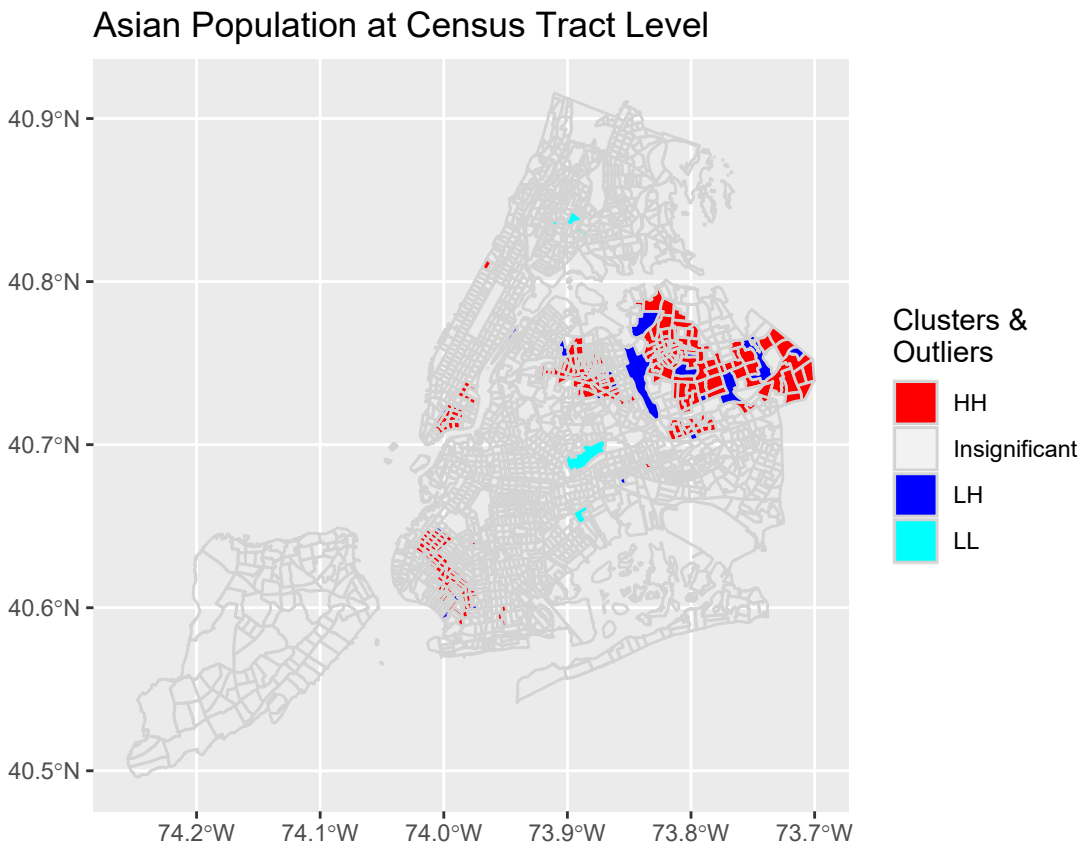
A few more examples from the same dataset.

```
# Define this as a function, which could save some space.
plotCOType <- function(varName, titleText, cols=1:5) {
  varVals <- nycDat[[varName]] %>% as.character() %>% as.numeric()
  lisaRslt <- spdep::localmoran(varVals, nycNbList,
                                zero.policy = TRUE, na.action = na.exclude)
  significanceLevel <- 0.05; # 95% confidence
  meanVal <- mean(varVals, na.rm=TRUE);

  lisaRslt %<>% tibble::as_tibble() %>%
    magrittr::set_colnames(c("Ii","E.Ii","Var.Ii","Z.Ii","Pr(z > 0)")) %>%
    dplyr::mutate(coType = dplyr::case_when(
    `Pr(z > 0)` > 0.05 ~ "Insignificant",
```

```
    `Pr(z > 0)` <= 0.05 & Ii >= 0 & varVals >= meanVal ~ "HH",
    `Pr(z > 0)` <= 0.05 & Ii >= 0 & varVals < meanVal ~ "LL",
    `Pr(z > 0)` <= 0.05 & Ii < 0 & varVals >= meanVal ~ "HL",
    `Pr(z > 0)` <= 0.05 & Ii < 0 & varVals < meanVal ~ "LH"
  ))

  # Now add this coType to original sf data
  nycDat$coType <- lisaRslt$coType %>% tidyr::replace_na("Insignificant")


  ggplot(nycDat) +
    geom_sf(aes(fill=coType),color = 'lightgrey') +
    scale_fill_manual(values = c('red','brown','NA','blue','cyan')[cols], name='Clusters & \
    labs(title = titleText)
}

plotCOType('medianinco', "Median Household Income at Census Tract Level")
```




```
plotCOType('asian', 'Asian Population at Census Tract Level', c(1,3,4,5))
```



Like many other spatial measures, global and local Moran's I are sensitive to geographic scales. While no outliers exist at census tract level for these variables, they may surface when being examined at a coarser scale such as the neighborhood level.

4.2 Spatial Error and Lag Models

With the presence of spatial autocorrelation, we need to address its possible influence on simple linear regression models that assume spatial independence. Spatial regression models are specifically developed to achieve this.

> Spatial regression models are statistical models that account for the presence of spatial effects, i.e., spatial autocorrelation (or more generally spatial dependence) and/or spatial heterogeneity. With R, we can run Ordinary Least Squares models using `lm` or other generalized linear regression using `glm`. Using `spdep` package, we can also obtain spatial diagnostics like Moran's I. While there are many other options, we can use `spdep` to estimate a model with a spatially lagged dependent variable (spatial lag model) and a spatial autoregressive process for the error term (spatial error model).
>
> — Adapted from GeoDa documentation

4.2.1 Spatial Error Model

Spatial error autocorrelation arises if error terms are correlated across observations, i.e., the error of an observation affects the errors of its neighbors. It is similar to serial correlation in time series analysis and leaves OLS coefficients unbiased but renders them inefficient. Because it's such a bothersome problem, spatial errors is also called "nuisance

> dependence in the error."
>
> There are a number of instances in which spatial error can arise. For example, similar to what can happen in time series, a source of correlation may come from unmeasured variables that are related through space. Correlation can also arise from aggregation of spatially correlated variables and systematic measurement error.
>
> So what to do if there is good reason to believe that there is spatial error? Maybe the most famous test is Moran's I which is based on the regression residuals and is also related to Moran's scatterplot of residuals which can be used to spot the problem graphically. There are other statistics like Lagrange multiplier and likelihood ratio tests, and each of them has different ways of getting at the same problem. If there is good reason to believe that spatial error is a problem, then the way forward is either model the error directly or to use autoregressive methods.
>
> In any case it's probably a good idea to assess whether spatial error might apply to the research problem. Because of it's effect on OLS, there might be a better way to estimate the coefficients that we are interested in, and the results might improve quite a bit.
>
> — James Greiner, Harvard Social Science Statistics Blog

The spatial error model handles the spatial autocorrelation in the residuals. The idea is that such errors (residuals from regression) are autocorrelated in that the error from one spatial feature can be modeled as a weighted average of the errors of its neighbors. In other words, such errors have spatial autocorrelation. This model can be expressed as:

$$\mathbf{y} = \mathbf{X} \ + \mathbf{u}, \qquad \mathbf{u} = \lambda_{\text{Err}} \mathbf{W}\mathbf{u} +$$

where $\mathbf{y}$ is an $(N \times 1)$ vector of observations on a response variable taken at each of N locations, $\mathbf{X}$ is an $(N \times k)$ matrix of covariates, is a $(k \times 1)$ vector of parameters, $\mathbf{u}$ is an $(N \times 1)$ spatially autocorrelated disturbance vector, is an $(N \times 1)$ vector of independent and identically distributed disturbances and λ_{Err} is a scalar spatial parameter.

The following code does an OLS and test whether spatial autocorrelation is present in the residuals.

```
# Prepare data: covert some factors to numeric values
nycDat %<>% dplyr::mutate(medianage = medianage %>% as.character() %>% as.numeric(),
                          househol_1 = househol_1 %>% as.character() %>% as.numeric())
# olsRslt <- lm(nycDat$popunemplo ~ nycDat$popinlabou +
#                 nycDat$onlylessth + nycDat$master +
#                 nycDat$africaninl + nycDat$asianinlab +
#                 nycDat$hispanicin + nycDat$medianage,
#               data = nycDat)
# summary(olsRslt)

# Create a simple linear regression on unemployed population
olsRslt <- lm(log(popunemplo +1) ~ log(1+popinlabou) +
                log(1+onlylessth) + log(1+master) +
                log(1+africaninl) + log(1+asianinlab) +
                log(1+hispanicin) + nycDat$medianage,
              data = nycDat)
summary(olsRslt)
```

```
#>
```

```
#> Call:
#> lm(formula = log(popunemplo + 1) ~ log(1 + popinlabou) + log(1 +
#>     onlylessth) + log(1 + master) + log(1 + africaninl) + log(1 +
#>     asianinlab) + log(1 + hispanicin) + nycDat$medianage, data = nycDat)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -4.6967 -0.2451  0.0518  0.3223  1.9265
#>
#> Coefficients:
#>                       Estimate Std. Error t value Pr(>|t|)
#> (Intercept)         -1.983558   0.148840 -13.327  < 2e-16 ***
#> log(1 + popinlabou)  0.731806   0.027352  26.755  < 2e-16 ***
#> log(1 + onlylessth)  0.185133   0.015073  12.283  < 2e-16 ***
#> log(1 + master)     -0.036226   0.013619  -2.660 0.007876 **
#> log(1 + africaninl)  0.082545   0.005890  14.015  < 2e-16 ***
#> log(1 + asianinlab) -0.015196   0.006458  -2.353 0.018709 *
#> log(1 + hispanicin)  0.040066   0.011579   3.460 0.000551 ***
#> nycDat$medianage     0.003001   0.001888   1.589 0.112124
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.5168 on 2116 degrees of freedom
#>   (42 observations deleted due to missingness)
#> Multiple R-squared:  0.6931, Adjusted R-squared:  0.6921
#> F-statistic: 682.8 on 7 and 2116 DF,  p-value: < 2.2e-16
```

```
# Derive the residuals from the regression. Need to handle those missed values
lmResiduals <- rep(0, length(nycDat$popunemplo))
resIndex <- olsRslt$residuals %>% names() %>% as.integer();
lmResiduals[resIndex] <- olsRslt$residuals

# Test if there is spatial autocorrelation in the regression residuals (errors).
nycNbList %>%
  spdep::moran.test(lmResiduals, ., zero.policy = TRUE)
```

```
#>
#>  Moran I test under randomisation
#>
#> data:  lmResiduals
#> weights: .  n reduced by no-neighbour observations
#>
#>
#> Moran I statistic standard deviate = 10.285, p-value < 2.2e-16
#> alternative hypothesis: greater
#> sample estimates:
#> Moran I statistic       Expectation          Variance
#>      0.1283881833     -0.0004625347      0.0001569522
```

The Moran's test shows that the residuals from linear regression have statistically significant positive spatial autocorrelation. The Moran's I is 0.128 and the p-value < 0.01. With that, we can run a

spatial error model to take such autocorrelation into account.

```
# use spdep package to run the spatial error model
# Use spatialreg::errorsarlm to run the same model
serrRslt <- spatialreg::errorsarlm(log(popunemplo +1) ~ log(1+popinlabou) +
                log(1+onlylessth) + log(1+master) +
                log(1+africaninl) + log(1+asianinlab) +
                log(1+hispanicin) + medianage,
              data = nycDat,
              listw = nycNbList,
              zero.policy = TRUE,
              na.action = na.omit);

summary(serrRslt)
```

```
#>
#> Call:spatialreg::errorsarlm(formula = log(popunemplo + 1) ~ log(1 +
#>     popinlabou) + log(1 + onlylessth) + log(1 + master) + log(1 +
#>     africaninl) + log(1 + asianinlab) + log(1 + hispanicin) +
#>     medianage, data = nycDat, listw = nycNbList, na.action = na.omit,
#>     zero.policy = TRUE)
#>
#> Residuals:
#>       Min        1Q    Median        3Q       Max
#> -4.526908 -0.244982  0.055031  0.300734  1.897656
#>
#> Type: error
#> Regions with no neighbours included:
#>  289 627 1967
#> Coefficients: (asymptotic standard errors)
#>                       Estimate Std. Error  z value  Pr(>|z|)
#> (Intercept)         -1.8431148  0.1519117 -12.1328 < 2.2e-16
#> log(1 + popinlabou)  0.7245485  0.0277532  26.1068 < 2.2e-16
#> log(1 + onlylessth)  0.1911611  0.0160471  11.9125 < 2.2e-16
#> log(1 + master)     -0.0399508  0.0139103  -2.8720  0.004078
#> log(1 + africaninl)  0.0761078  0.0068356  11.1341 < 2.2e-16
#> log(1 + asianinlab) -0.0115502  0.0070767  -1.6322  0.102646
#> log(1 + hispanicin)  0.0346482  0.0127560   2.7162  0.006603
#> medianage            0.0014076  0.0020147   0.6987  0.484770
#>
#> Lambda: 0.29439, LR test value: 85.367, p-value: < 2.22e-16
#> Asymptotic standard error: 0.03168
#>     z-value: 9.2926, p-value: < 2.22e-16
#> Wald statistic: 86.352, p-value: < 2.22e-16
#>
#> Log likelihood: -1564.967 for error model
#> ML residual variance (sigma squared): 0.25155, (sigma: 0.50155)
#> Number of observations: 2124
#> Number of parameters estimated: 10
#> AIC: 3149.9, (AIC for lm: 3233.3)
```

From the AIC, the spatial error model performs much better than the linear model (lower AIC basically means better fit). The

$$\lambda$$

Lambda value of 0.294 is also statistically significant, suggesting the error term is spatially autoregressive. While the model coefficients do not change much, most of them have higher absolute Z values, meaning more robust estimation with lower variance.

```
# Derive the residuals from the regression. Need to handle those missed values
seResiduals <- rep(0, length(nycDat$popunemplo))
resIndex <- serrRslt$residuals %>% names() %>% as.integer();
seResiduals[resIndex] <- serrRslt$residuals

# Test if there is spatial autocorrelation in the regression residuals (errors).
nycNbList %>%
  spdep::moran.test(seResiduals, ., zero.policy = TRUE)
```

```
#>
#>  Moran I test under randomisation
#>
#> data:  seResiduals
#> weights: .  n reduced by no-neighbour observations
#>
#>
#> Moran I statistic standard deviate = -0.54661, p-value = 0.7077
#> alternative hypothesis: greater
#> sample estimates:
#> Moran I statistic       Expectation          Variance
#>     -0.0073098388     -0.0004625347      0.0001569199
```

Now, it is rather clear that there is no spatial autocorrelation in the residuals anymore as the Moran's I is close to zero now. More importantly, the p-value is 0.71, meaning we cannot reject the hypothesis that the Moran's I is zero.

4.2.2 Spatial Lag Model

> A spatial lag is a variable that essentially averages the neighboring values of a location (the value of each neighboring location is multiplied by the spatial weight and then the products are summed). It can be used to compare the neighboring values with those of the location itself. Which locations are defined as neighbors in this process is specified through a row-standardized spatial weights matrix in GeoDa and a list of standardized weights in `spdep`. By convention, the location at the center of its neighbors is not included in the definition of neighbors and is therefore set to zero.
>
> — Adapted from GeoDa Documentation

For these spatial lags, we can use the spatial lag model to address the spatial autocorrelation in the dependent variable.

$$\mathbf{y} = \rho_{\text{Lag}}\mathbf{W}\mathbf{y} + \mathbf{X} \; + \; ,$$

where ρ_{Lag} is a scalar spatial parameter, indicating how much a spatial feature is influenced by its neighbors.

```
# use spdep package to run the spatial lag model
# spatialreg::lagsarlm
slmRslt <- spatialreg::lagsarlm(log(popunemplo +1) ~ log(1+popinlabou) +
                log(1+onlylessth) + log(1+master) +
                log(1+africaninl) + log(1+asianinlab) +
                log(1+hispanicin) + medianage,
              data = nycDat,
              listw = nycNbList,
              zero.policy = TRUE,
              na.action = na.omit);

summary(slmRslt)
```

```
#>
#> Call:spatialreg::lagsarlm(formula = log(popunemplo + 1) ~ log(1 +
#>     popinlabou) + log(1 + onlylessth) + log(1 + master) + log(1 +
#>     africaninl) + log(1 + asianinlab) + log(1 + hispanicin) +
#>     medianage, data = nycDat, listw = nycNbList, na.action = na.omit,
#>     zero.policy = TRUE)
#>
#> Residuals:
#>       Min        1Q    Median        3Q       Max
#> -4.591905 -0.248899  0.053451  0.306993  1.797936
#>
#> Type: lag
#> Regions with no neighbours included:
#>  289 627 1967
#> Coefficients: (asymptotic standard errors)
#>                       Estimate Std. Error  z value Pr(>|z|)
#> (Intercept)         -2.4911142  0.1626826 -15.3127  < 2e-16
#> log(1 + popinlabou)  0.7208913  0.0269525  26.7467  < 2e-16
#> log(1 + onlylessth)  0.1810895  0.0148579  12.1881  < 2e-16
#> log(1 + master)     -0.0314152  0.0134174  -2.3414  0.01921
#> log(1 + africaninl)  0.0713624  0.0059629  11.9678  < 2e-16
#> log(1 + asianinlab) -0.0150805  0.0063527  -2.3739  0.01760
#> log(1 + hispanicin)  0.0287700  0.0114775   2.5066  0.01219
#> medianage            0.0030659  0.0018591   1.6491  0.09912
#>
#> Rho: 0.14127, LR test value: 54.515, p-value: 1.5421e-13
#> Asymptotic standard error: 0.019469
#>     z-value: 7.2563, p-value: 3.979e-13
#> Wald statistic: 52.654, p-value: 3.979e-13
#>
#> Log likelihood: -1580.393 for lag model
#> ML residual variance (sigma squared): 0.25841, (sigma: 0.50834)
#> Number of observations: 2124
#> Number of parameters estimated: 10
#> AIC: 3180.8, (AIC for lm: 3233.3)
#> LM test for residual autocorrelation
#> test value: 31.043, p-value: 2.5237e-08
```

Similarly, the spatial lag model is much better than the linear mode, even though it is not as good as the spatial error model. The spatial lag term

$$\rho$$

or Rho is also statistically significant.

```
# Derive the residuals from the regression. Need to handle those missed values
slResiduals <- rep(0, length(nycDat$popunemplo))
resIndex <- slmRslt$residuals %>% names() %>% as.integer();
slResiduals[resIndex] <- slmRslt$residuals

# Test if there is spatial autocorrelation in the regression residuals (errors).
nycNbList %>%
  spdep::moran.test(slResiduals, ., zero.policy = TRUE)
```

```
#>
#>  Moran I test under randomisation
#>
#> data:  slResiduals
#> weights: .  n reduced by no-neighbour observations
#>
#>
#> Moran I statistic standard deviate = 4.7, p-value = 1.301e-06
#> alternative hypothesis: greater
#> sample estimates:
#> Moran I statistic       Expectation          Variance
#>      0.0584191208     -0.0004625347      0.0001569527
```

The under-performance of spatial lag model is also evident from the Moran's I test. Residuals from spatial lag model still have statistically significant ($p < 0.01$) spatial autocorrelation, although the Moran's I of 0.06 is much smaller than 0.27 of the dependent variable.

In fact, we can combine the spatial error and spatial lag models into a spatial simultaneous autoregression model or "SAC/SARAR" model.

$$\mathbf{y} = \rho_{\text{Lag}}\mathbf{W}\mathbf{y} + \mathbf{X} \ + \mathbf{u}, \qquad \mathbf{u} = \lambda_{\text{Err}}\mathbf{W}\mathbf{u} +$$

This model can be estimated using `spatialreg::sacsarlm` but will not be discussed here.

4.3 Spatial Heterogeneity and Spatially Varying Coefficients

> Spatial heterogeneity exists when structural changes related to location exist in a dataset. In such cases, spatial regimes might be present, which are characterized by differing parameter values or functional forms (e.g., crime in certain regions might be structurally different from crime in other regions). Spatial heterogeneity can result in non-constant error variance (heteroskedasticity) across areas, especially when scale-related measurement errors are present. It can be difficult to distinguish from spatial dependence.
>
> — GeoDa Documentation

There are two families of models that can nicely address spatial heterogeneity: geographically weighted regression (GWR) and Bayesian hierarchical spatial models. While GWR is more like an

exploratory tool than a rigorous statistical analysis, it is rather easy to understand and can produce insightful visualizations.

4.3.1 Geographically Weighted Regression (GWR)

> Geographically Weighted Regression (GWR) is one of several spatial regression techniques used in geography and other disciplines to address spatial heterogeneity. GWR evaluates a local model of the variable or process that we are trying to understand or predict by fitting a regression equation to every feature in the dataset. GWR constructs these separate equations by incorporating the dependent and explanatory variables of the features falling within the neighborhood of each target feature or the `k` nearest neighbors. GWR should be applied to datasets with over several hundred features. It is not an appropriate method for small datasets and does not work with multipoint data.
>
> Geographically Weighted Regression can be used for a variety of applications, including the following:
>
> - Is the relationship between educational attainment and income consistent across the study area?
> - Do certain illness or disease occurrences increase with proximity to water features?
> - What are the key variables that explain high forest fire frequency?
> - Which habitats should be protected to encourage the reintroduction of an endangered species?
> - Where are the districts where children are achieving high test scores? What characteristics seem to be associated? Where is each characteristic most important?
> - Are the factors influencing higher cancer rates consistent across the study area?
>
> — ArcGIS Documentation

Here we are going to do a basic GWR using the `GWmodel` package in R. First we need to estimate an optimal bandwidth that will define the local neighborhood for every spatial feature. Then spatial features within the neighborhood will be used to estimate a regression model for that particular feature. There are many different ways to define such "neighborhoods" or `bandwidth`. The code below is using an "adaptive" neighborhood, which is defined as the `k` nearest neighbors based on Euclidean distances. But then how do we know how many neighbors we should use for the local regression? Or what is best value for `k`? `GWmodel::bw.gwr` can help us search for such an optimal value based on the model fitness. With that bandwidth, we can conduct a basic GWR.

```
# Remove all the NAs in the dataset
nycDatNoNA <- nycDat %>% tidyr::drop_na()
# Estimate an optimal bandwidth
bwVal <- GWmodel::bw.gwr(log(popunemplo +1) ~ log(1+popinlabou) +
                           log(1+onlylessth) + log(1+master) +
                           log(1+africaninl) + log(1+asianinlab) +
                           log(1+hispanicin) + medianage,
                         data = nycDatNoNA %>% sf::as_Spatial(),
                         approach = 'AICc', kernel = 'bisquare',
                         adaptive = TRUE)
```

```
#> Take a cup of tea and have a break, it will take a few minutes.
#>           -----A kind suggestion from GWmodel development group
#> Adaptive bandwidth (number of nearest neighbours): 1314 AICc value: 3056.162
#> Adaptive bandwidth (number of nearest neighbours): 820 AICc value: 3010.612
```

```
#> Adaptive bandwidth (number of nearest neighbours): 513 AICc value: 2977.403
#> Adaptive bandwidth (number of nearest neighbours): 325 AICc value: 2943.909
#> Adaptive bandwidth (number of nearest neighbours): 207 AICc value: 2925.297
#> Adaptive bandwidth (number of nearest neighbours): 136 AICc value: 2944.405
#> Adaptive bandwidth (number of nearest neighbours): 253 AICc value: 2930.353
#> Adaptive bandwidth (number of nearest neighbours): 180 AICc value: 2923.699
#> Adaptive bandwidth (number of nearest neighbours): 162 AICc value: 2928.638
#> Adaptive bandwidth (number of nearest neighbours): 190 AICc value: 2925.533
#> Adaptive bandwidth (number of nearest neighbours): 172 AICc value: 2925.419
#> Adaptive bandwidth (number of nearest neighbours): 183 AICc value: 2923.988
#> Adaptive bandwidth (number of nearest neighbours): 176 AICc value: 2924.437
#> Adaptive bandwidth (number of nearest neighbours): 180 AICc value: 2923.699
```

```
# Perform a basic GWR
gwr.res <- gwr.basic(log(popunemplo +1) ~ log(1+popinlabou) +
                           log(1+onlylessth) + log(1+master) +
                           log(1+africaninl) + log(1+asianinlab) +
                           log(1+hispanicin) + medianage,
                         data = nycDatNoNA %>% sf::as_Spatial(),
                     bw = bwVal, kernel = "bisquare", adaptive = TRUE)
```

```
#> Warning in proj4string(data): CRS object has comment, which is lost in output; in tests, see
#> https://cran.r-project.org/web/packages/sp/vignettes/CRS_warnings.html
```

```
# Show the results
print(gwr.res)
```

```
#>    ***********************************************************************
#>    *                       Package   GWmodel                             *
#>    ***********************************************************************
#>    Program starts at: 2022-03-24 18:14:25
#>    Call:
#>    gwr.basic(formula = log(popunemplo + 1) ~ log(1 + popinlabou) +
#>     log(1 + onlylessth) + log(1 + master) + log(1 + africaninl) +
#>     log(1 + asianinlab) + log(1 + hispanicin) + medianage, data = nycDatNoNA %>%
#>     sf::as_Spatial(), bw = bwVal, kernel = "bisquare", adaptive = TRUE)
#>
#>    Dependent (y) variable:  popunemplo
#>    Independent variables:  popinlabou onlylessth master africaninl asianinlab hispanicin media
#>    Number of data points: 2115
#>    ***********************************************************************
#>    *                    Results of Global Regression                     *
#>    ***********************************************************************
#>
#>    Call:
#>     lm(formula = formula, data = data)
#>
#>    Residuals:
#>     Min      1Q  Median      3Q     Max
#> -4.7298 -0.2514  0.0565  0.3249  2.0892
#>
```

```
#>    Coefficients:
#>                          Estimate Std. Error t value Pr(>|t|)
#>    (Intercept)         -2.621486   0.172640 -15.185  < 2e-16 ***
#>    log(1 + popinlabou)  0.856573   0.032845  26.079  < 2e-16 ***
#>    log(1 + onlylessth)  0.167166   0.015982  10.459  < 2e-16 ***
#>    log(1 + master)     -0.061666   0.014385  -4.287 1.89e-05 ***
#>    log(1 + africaninl)  0.078297   0.005837  13.415  < 2e-16 ***
#>    log(1 + asianinlab) -0.018332   0.006388  -2.870  0.00415 **
#>    log(1 + hispanicin)  0.030994   0.011509   2.693  0.00714 **
#>    medianage            0.003609   0.001902   1.898  0.05788 .
#>
#>    ---Significance stars
#>    Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>    Residual standard error: 0.5098 on 2107 degrees of freedom
#>    Multiple R-squared: 0.6757
#>    Adjusted R-squared: 0.6746
#>    F-statistic: 627.1 on 7 and 2107 DF,  p-value: < 2.2e-16
#>    ***Extra Diagnostic information
#>    Residual sum of squares: 547.5001
#>    Sigma(hat): 0.5090288
#>    AIC:  3161.799
#>    AICc:  3161.884
#>    BIC:  1166.621
#>    ***********************************************************************
#>    *          Results of Geographically Weighted Regression              *
#>    ***********************************************************************
#>
#>    ********************Model calibration information*********************
#>    Kernel function: bisquare
#>    Adaptive bandwidth: 180 (number of nearest neighbours)
#>    Regression points: the same locations as observations are used.
#>    Distance metric: Euclidean distance metric is used.
#>
#>    ****************Summary of GWR coefficient estimates:******************
#>                               Min.    1st Qu.     Median    3rd Qu.   Max.
#>    Intercept           -7.2771344 -3.2662540 -2.3904552 -1.5407623 0.4646
#>    log(1 + popinlabou)  0.0227865  0.7145161  0.8723999  1.0315822 1.7145
#>    log(1 + onlylessth) -0.3053821  0.0612402  0.1315848  0.2488254 0.6085
#>    log(1 + master)     -0.3135134 -0.1358077 -0.0505235 -0.0144470 0.1334
#>    log(1 + africaninl) -0.1071455  0.0301272  0.0555869  0.1098297 0.3374
#>    log(1 + asianinlab) -0.1755278 -0.0413494 -0.0182863  0.0149573 0.2309
#>    log(1 + hispanicin) -0.5906099 -0.0249766  0.0281980  0.0997731 0.4744
#>    medianage           -0.0522653 -0.0114841 -0.0010550  0.0082988 0.0810
#>    ************************Diagnostic information*************************
#>    Number of data points: 2115
#>    Effective number of parameters (2trace(S) - trace(S'S)): 288.7402
#>    Effective degrees of freedom (n-2trace(S) + trace(S'S)): 1826.26
#>    AICc (GWR book, Fotheringham, et al. 2002, p. 61, eq 2.33): 2923.699
#>    AIC (GWR book, Fotheringham, et al. 2002,GWR p. 96, eq. 4.22): 2651.793
#>    BIC (GWR book, Fotheringham, et al. 2002,GWR p. 61, eq. 2.34): 1992.695
```

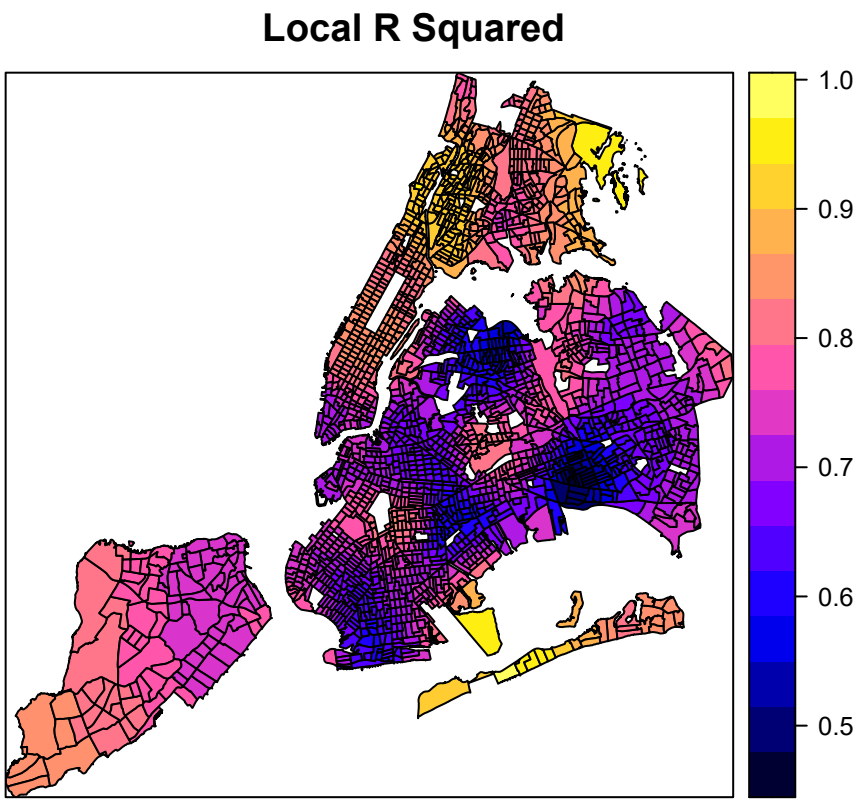
```
#>    Residual sum of squares: 391.2425
#>    R-square value:  0.768232
#>    Adjusted R-square value:  0.7315684
#>
#>    ************************************************************************
#>    Program stops at: 2022-03-24 18:14:26
```

While the pseudo-R square values improve a lot from GWR, the most valuable application of GWR outputs is to examine the spatial variations of the estimated model coefficients.

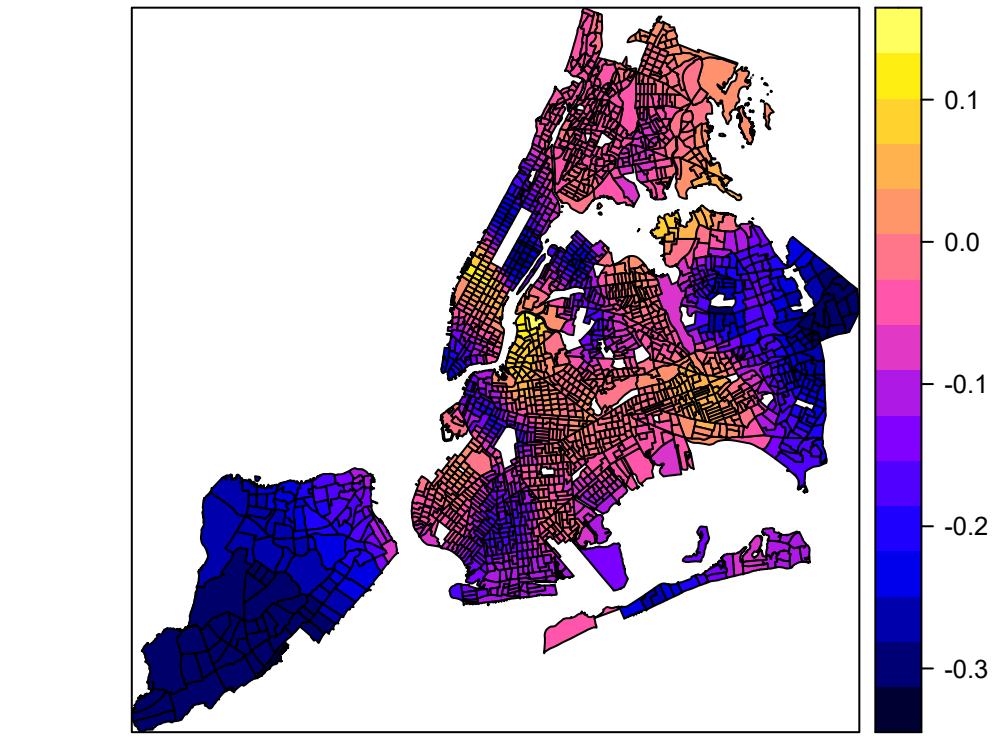
```
# The gwr.res$SDF is a Spatial*DataFrame that contain all the coefficients estimated
# at each sptial feature. We can simply map it out as a sp object
names(gwr.res$SDF)
```

```
#>  [1] "Intercept"              "log(1 + popinlabou)"    "log(1 + onlylessth)"
#>  [4] "log(1 + master)"        "log(1 + africaninl)"    "log(1 + asianinlab)"
#>  [7] "log(1 + hispanicin)"    "medianage"              "y"
#> [10] "yhat"                   "residual"               "CV_Score"
#> [13] "Stud_residual"          "Intercept_SE"           "log(1 + popinlabou)_SE"
#> [16] "log(1 + onlylessth)_SE" "log(1 + master)_SE"     "log(1 + africaninl)_SE"
#> [19] "log(1 + asianinlab)_SE" "log(1 + hispanicin)_SE" "medianage_SE"
#> [22] "Intercept_TV"           "log(1 + popinlabou)_TV" "log(1 + onlylessth)_TV"
#> [25] "log(1 + master)_TV"     "log(1 + africaninl)_TV" "log(1 + asianinlab)_TV"
#> [28] "log(1 + hispanicin)_TV" "medianage_TV"           "Local_R2"
```

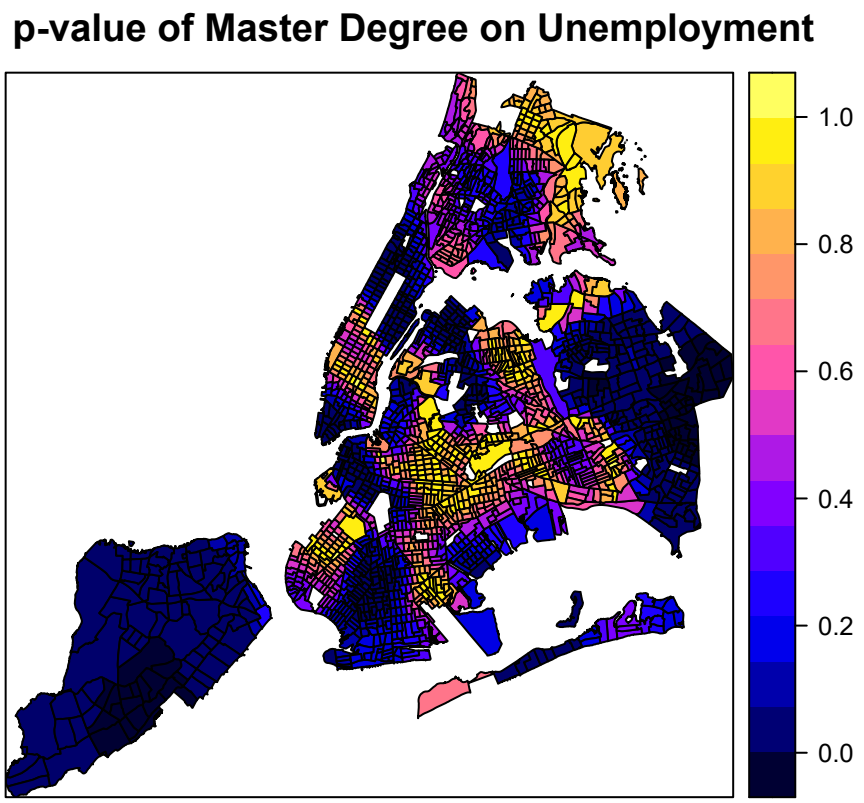
```
# Because the spplot cannot handle those unconventional column names, we need to do extra works.
spGWRData <- gwr.res$SDF@data;
spGWRData$coefMaster <- spGWRData$`log(1 + master)`;
# Calculate the p value from (student) t value
spGWRData$pMaster <- 2*pt(-abs(spGWRData$`log(1 + master)_TV`), df = dim(spGWRData)[1] -1)

spGWRData$coefAfrican <- spGWRData$`log(1 + africaninl)`;
spGWRData$coefHighschool <- spGWRData$`log(1 + onlylessth)`

spGWR <- gwr.res$SDF;
spGWR@data <- spGWRData;

# This show how the coefficients of the master degree on unemplyment change over space.
spplot(spGWR, 'Local_R2', main="Local R Squared")
```



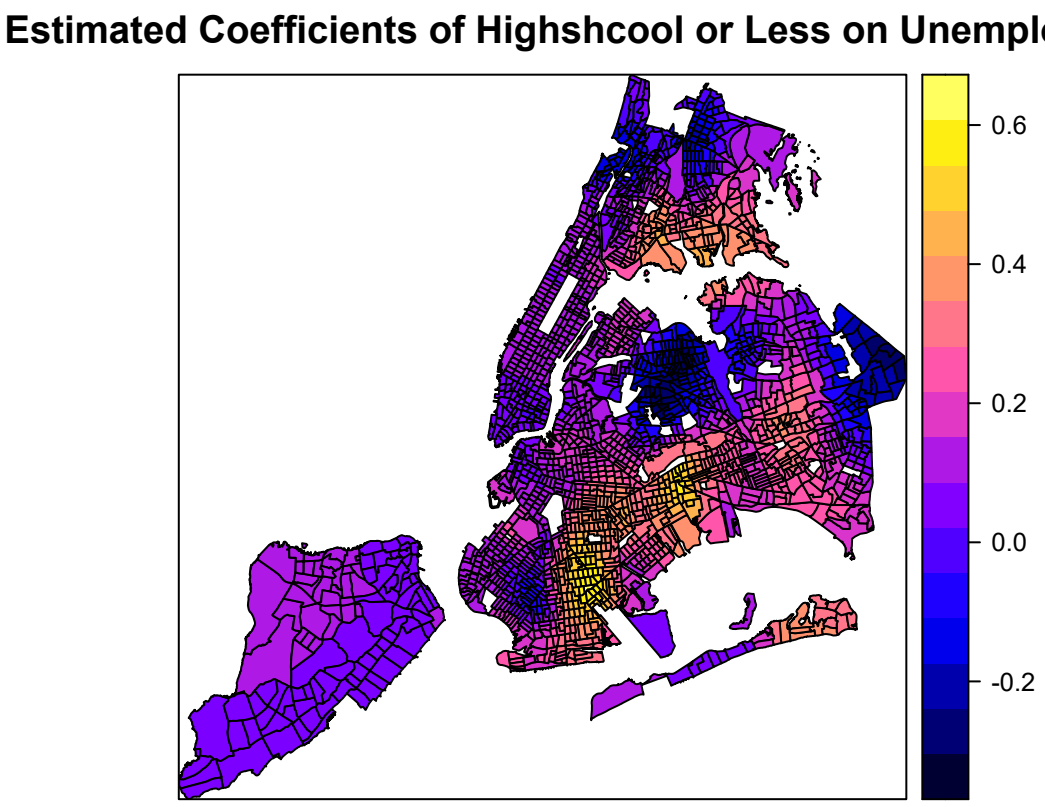

```
# This show how the coefficients of the master degree on unemplyment change over space.
spplot(spGWR, 'coefMaster', main="Estimated Coefficients of Master Degree on Unemployment")
```




```
# But this variable is not signficiant everywhere.
spplot(spGWR, 'pMaster', main="p-value of Master Degree on Unemployment")
```




```
# Another factor of highschool
spplot(spGWR, 'coefHighschool', main="Estimated Coefficients of Highshcool or Less on Unempl
```



The `SDF` object in the `gwr.basic` resutls contains estimated coefficients, predicted Y calues, coefficient standard errors, and t-values. All these values are organized in a Spatial*DataFrame that can be directly mapped.

From these plots, it seems like a master degree, while helping reduce unemployment overall with statistical significance, may not matter at some places (where p-value is high). Additionally, it can also positively contribute to unemployment at some areas, even though the master degree factor is mostly not significant at those areas. Similarly, the number of population with an education of high school or less in the labor force may positively or negatively contribute to the unemployment, depending on the locations. Clearly, GWR is a powerful exploratory tool as we instantly want to know why such spatial variation exists and what might explain it. Such exploration would continue in advanced R topics, but the book has to stop here.