Building Spatial Databases based on Location

HES 505 Fall 2024: Session 15

Carolyn Koehn

Objectives

By the end of today you should be able to:

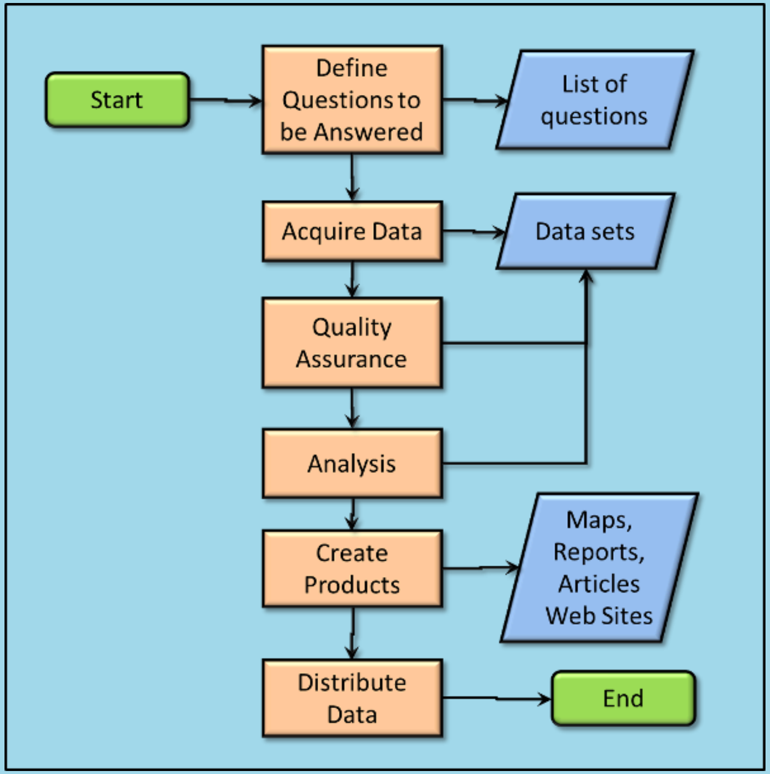
- Create new features based on topological relationships
- Use topological subsetting to reduce features
- Use spatial joins to add attributes based on location

Revisiting Spatial Analysis

What is spatial analysis?

> "The process of examining the locations, attributes, and relationships of features in spatial data through overlay and other analytical techniques in order to address a question or gain useful knowledge. Spatial analysis extracts or creates new information from spatial data".
> — ESRI Dictionary

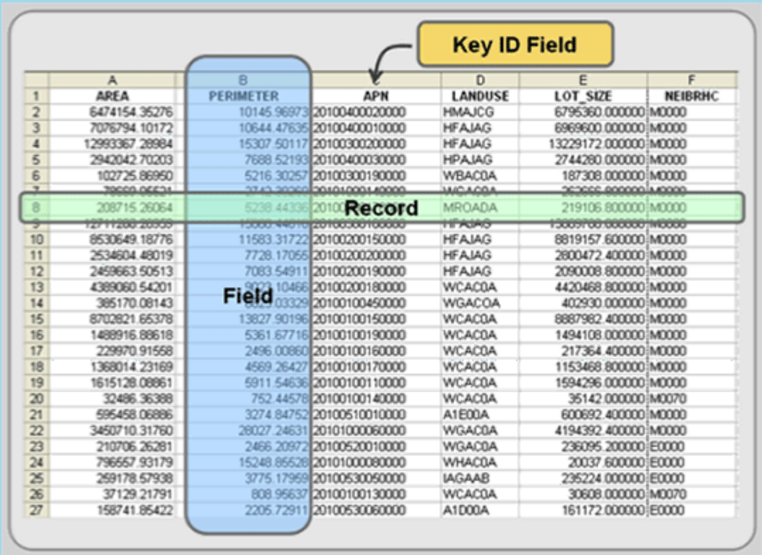
Workflows for spatial analysis



courtesy of Humboldt State University

- Align processing with objectives
- Imagining the visualizations and analysis clarifies file formats and variables
- Helps build reproducibility

Databases and Attributes



courtesy of Giscommons

- Attributes: Information that further describes a spatial feature
- Attributes → predictors for analysis
- Monday's focus on thematic relations between datasets
  - Shared 'keys' help define linkages between objects
- Sometimes we are interested in attributes that describe location (overlaps, contains, distance)
- Sometimes we want to join based on location rather than thematic connections
  - **Must have the same CRS**

Calculating New Attributes

Attributes based on geometry and location (`measures`)

- Attributes like area and length can be useful for a number of analyses
  - Estimates of 'effort' in sampling designs
  - Offsets for modeling rates (e.g., Poisson regression)
- Need to assign the result of the function to a column in data frame (e.g., `$`, `mutate`, and `summarize`)
- Often useful to test before assigning

Estimating area

- `sf` bases area (and length) calculations on the map units of the CRS
- the `units` library allows conversion into a variety of units

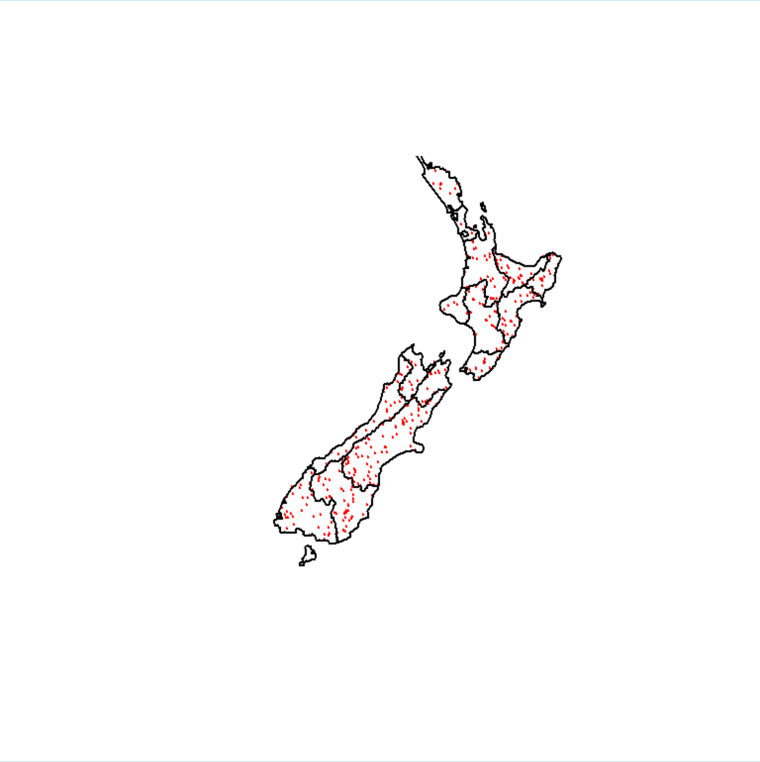
```
nz.sf <- nz %>%
  mutate(area = st_area(nz
head(nz.sf$area, 3)
```

```
Units: [m^2]
[1] 12890576439  4911565037
24588819863
```

```
nz.sf$areakm <- units::set
head(nz.sf$areakm, 3)
```

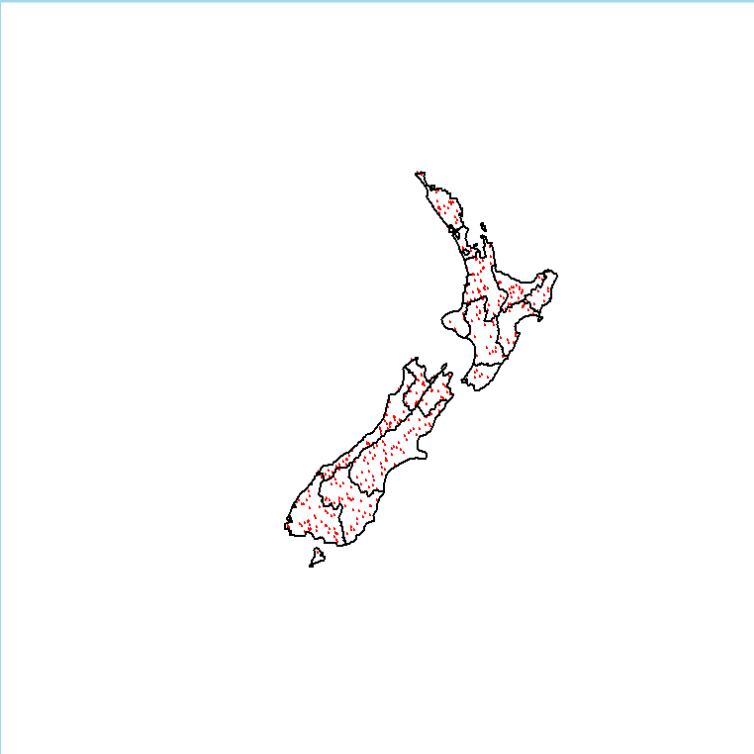
```
Units: [km^2]
[1] 12890.576  4911.565
24588.820
```

Estimating Density in Polygons



- Creating new features based on the frequency of occurrence
- Clarifying graphics
- Underlies quadrat sampling for point patterns
- Two steps: count and area

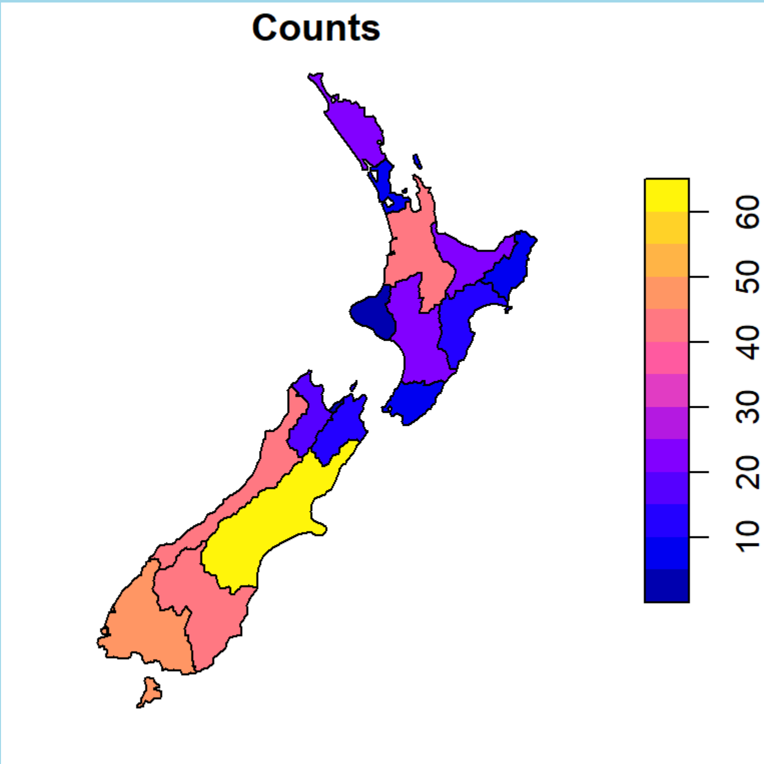
Estimating Density in Polygons

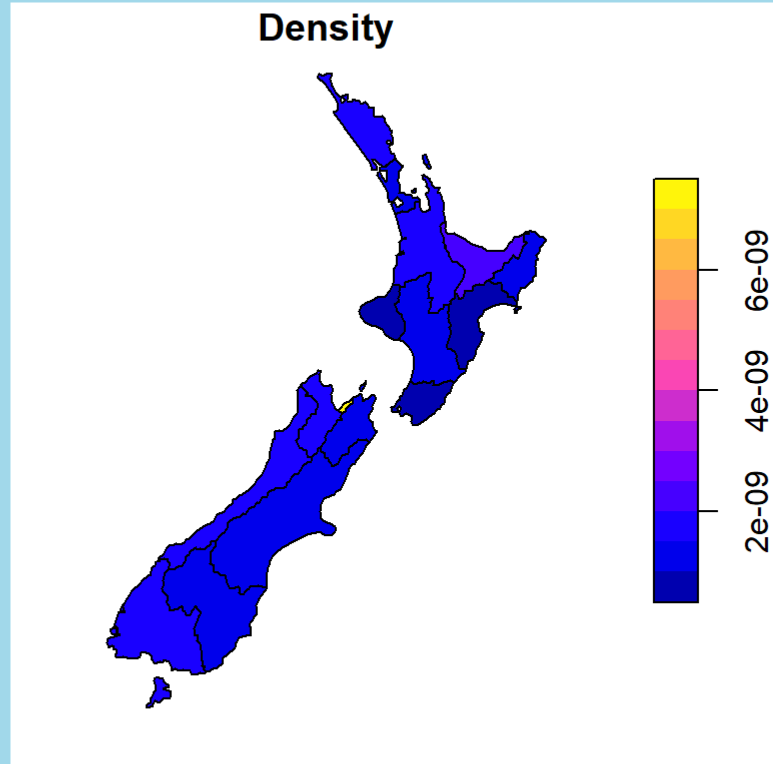


```
nz.df <- nz %>%
mutate(counts = lengths(st_intersects(., r
          area = st_area(nz),
          density = counts/area)
head(st_drop_geometry(nz.df[,7:10]))
```

```
  counts               area
density
1      24 12890576439 [m^2] 1.861825e-09
[1/m^2]
2       7  4911565037 [m^2] 1.425208e-09
[1/m^2]
3      42 24588819863 [m^2] 1.708093e-09
[1/m^2]
4      25 12271015945 [m^2] 2.037321e-09
[1/m^2]
5      10  8364554416 [m^2] 1.195521e-09
[1/m^2]
6      14 14242517871 [m^2] 9.829723e-10
[1/m^2]
```

Estimating Density in Polygons



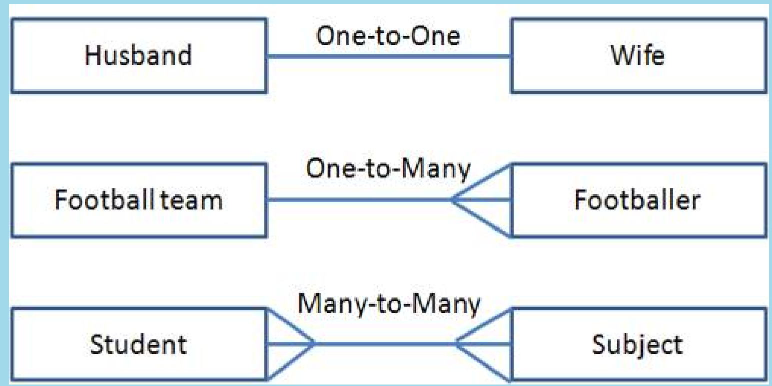


Estimating Distance

- As a covariate
- For use in covariance matrices
- As a means of assigning connections in networks

Estimating Single Point Distance

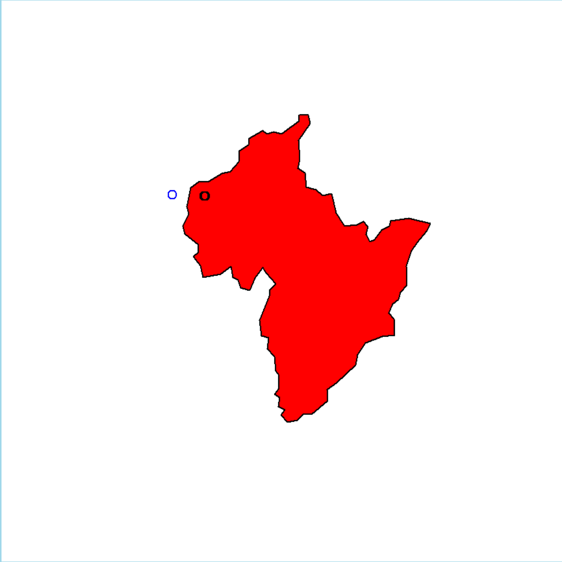
- `st_distance` returns distances between all features in `x` and all features in `y`
- One-to-One relationship requires choosing a single point for `y`



Estimating Single Point Distance

- Subsetting y into a single feature

```
canterbury = nz %>% filter(Name == "Canterbury")
canterbury_height = nz_height[canterbury, ]
co = filter(nz, grepl("Canter|Otag", Name))
st_distance(nz_height[1:3, ], co)
```



```
Units: [m]
          [,1]     [,2]
[1,] 123537.16 15497.72
[2,]  94282.77     0.00
[3,]  93018.56     0.00
```

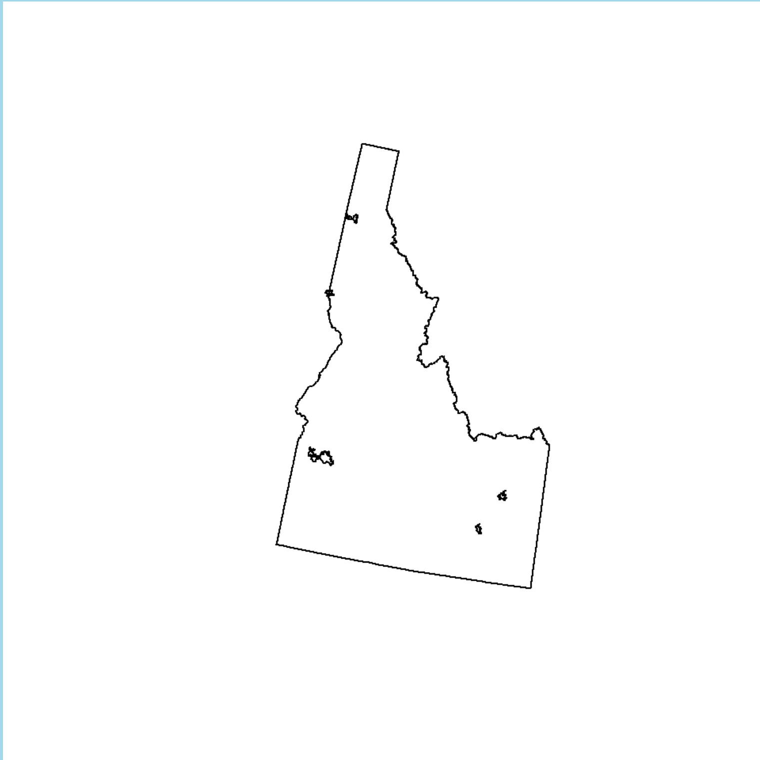
Estimating Single Point Distance

- Using nearest neighbor distances

```
ua <- urban_areas(cb = FALSE, progress_bar
  filter(., UATYP10 == "U") %>%
  filter(., str_detect(NAME10, "ID")) %>%
  st_transform(., crs=2163)

#get index of nearest ID city
nearest <-  st_nearest_feature(ua)
#estimate distance
(dist = st_distance(ua, ua[nearest,], by_e
```

```
Units: [m]
[1]  61373.575  61373.575   1647.128
1647.128 136917.546 136917.546
```

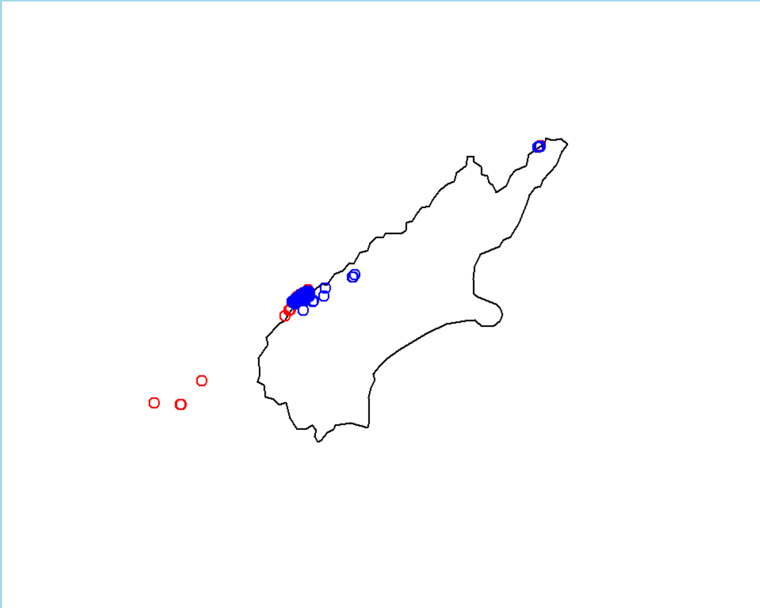


Topological Subsetting

Topological Subsetting

- Topological relations describe the spatial relationships between objects
- We can use the overlap (or not) of vector data to subset the data based on topology
- Need *valid* geometries
- Easiest way is to use [notation, but also most restrictive

```
ctby_height <-  nz_height[canterbury, ]
```



Topological Subsetting

- Lots of verbs in `sf` for doing this (e.g., `st_intersects`, `st_contains`, `st_touches`)
- see `?geos_binary_pred` for a full list
- Creates an **implicit** attribute (the *records* in `x` that are "in" `y`)

Using `sparse=TRUE`

```
st_intersects(nz_height, co,
                         sparse = TRUE)[1:3]
```

```
[[1]]
integer(0)

[[2]]
[1] 2

[[3]]
[1] 2
```

```
lengths(st_intersects(nz_height,
                               co, sparse =
```

```
[1] FALSE  TRUE  TRUE
```

Topological Subsetting

- The `sparse` option controls how the results are returned
- We can then find out if one or more elements satisfies the criteria

Using `sparse=FALSE`

```
st_intersects(nz_height, co, sparse = FALSE)[1:3,]
```

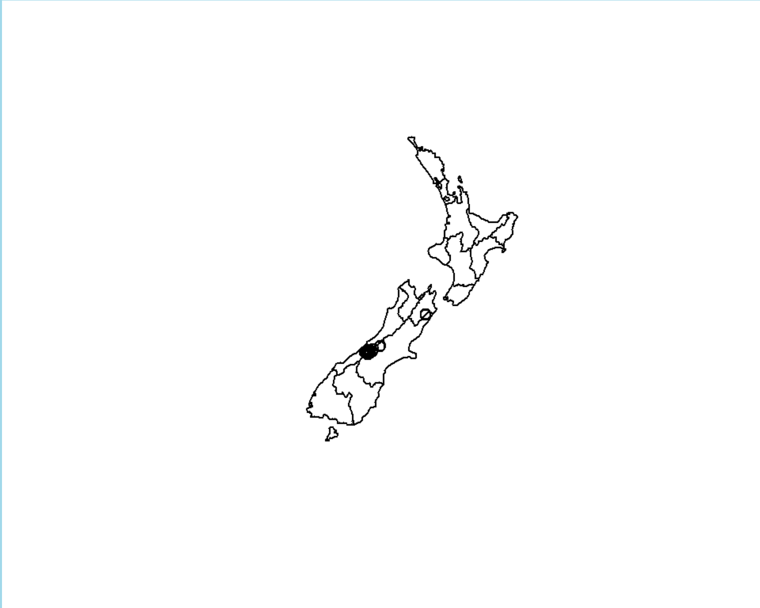
```
      [,1]  [,2]
[1,] FALSE FALSE
[2,] FALSE  TRUE
[3,] FALSE  TRUE
```

```
apply(st_intersects(nz_height, co, sparse = FALSE), 1,any)[1:3]
```

```
[1] FALSE  TRUE  TRUE
```

Topological Subsetting

```
canterbury_height3 = nz_height %>%
  filter(st_intersects(x = ., y = canterbu
```



Spatial Joins

Spatial Joins

- `sf` package provides `st_join` for vectors
- Allows joins based on the predicates (`st_intersects`, `st_touches`, `st_within_distance`, etc.)
- Default is a left join

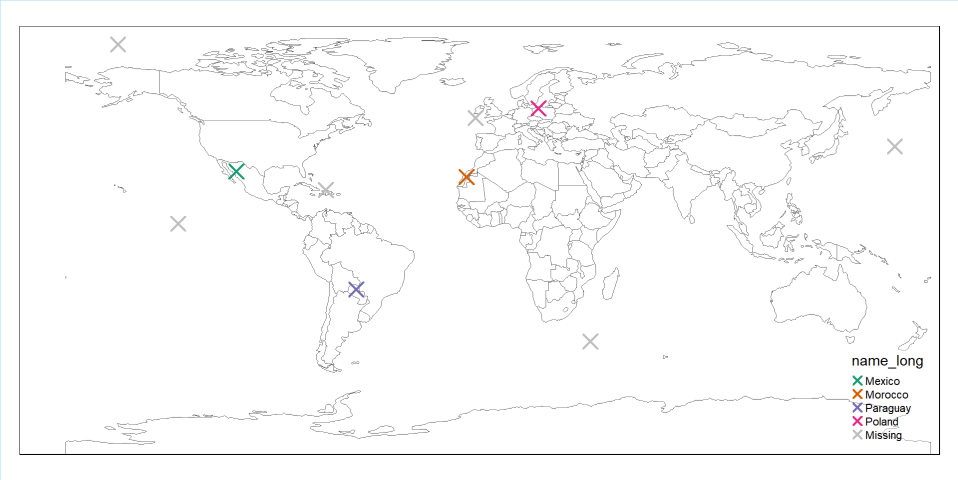
Spatial Joins

```
set.seed(2018)
(bb = st_bbox(world)) # the world'
```

```
      xmin       ymin       xmax
ymax
-180.00000  -89.90000  179.99999
83.64513
```

```
#>    xmin    ymin    xmax    ymax
#> -180.0   -89.9   180.0    83.6
random_df = data.frame(
  x = runif(n = 10, min = bb[1], m
  y = runif(n = 10, min = bb[2], m
)
random_points <- random_df %>%
  st_as_sf(coords = c("x", "y")) %
  st_set_crs("EPSG:4326") # set ge

random_joined = st_join(random_poi
```



Spatial Joins

- Sometimes we may want to be less restrictive
- Just because objects don't touch doesn't mean they don't relate to each other
- Can use `predicates` in `st_join`
- Remember that default is `left_join` (so the number of records can grow if multiple matches)

Spatial Joins

```r
any(st_touches(cycle_hire, cycle_hire_osm, sparse
```

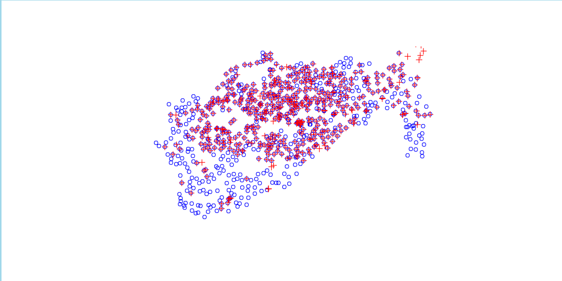
```
[1] FALSE
```

```r
z = st_join(cycle_hire, cycle_hire_osm, st_is_with
nrow(cycle_hire)
```

```
[1] 742
```

```r
nrow(z)
```

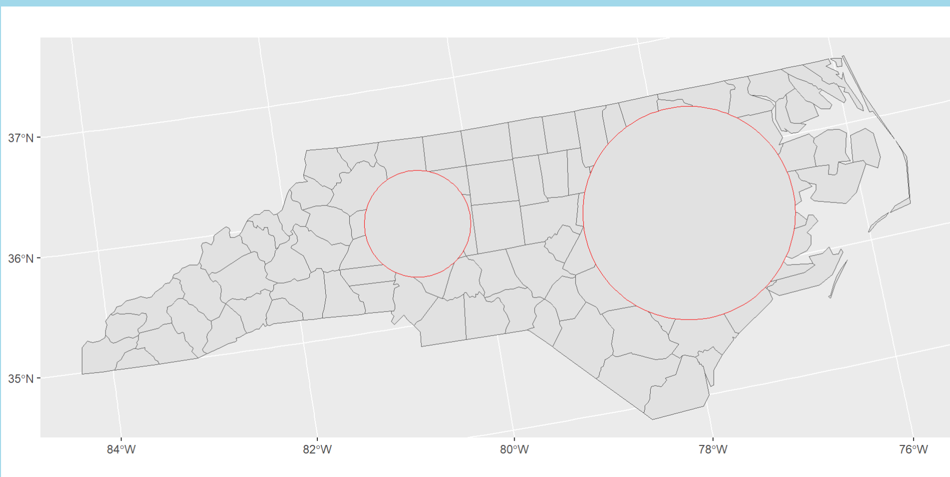
```
[1] 762
```



Extending Joins

Extending Joins

- Sometimes we are interested in analyzing locations that contain the overlap between two vectors
  - How much of home range *a* occurs on soil type *b*
  - How much of each Census tract is contained with a service provision area?
- `st_intersection`, `st_union`, and `st_difference` return new geometries that we can use as records in our spatial database




```
intersect_pct <- st_intersection(n
   mutate(intersect_area = st_area
   dplyr::select(NAME, intersect_a
   st_drop_geometry()

nc <- mutate(nc, county_area = st_

# Merge by county name
nc <- merge(nc, intersect_pct, by

# Calculate coverage
nc <- nc %>%
   mutate(coverage = as.numeric(in
```

Extending Joins