EDS 223: week 3 lab

Ruth Oliver

2022-10-17

introduction

In this lab, we'll explore the basics of spatial and geometry operations on vector data in R using the **sf** package. The following materials are modified from Chapter 4 and Chapter 5of Geocomputation with R by Rovin Lovelace.

prerequisites

```
rm(list = ls())
library(sf)
library(spData)
library(tmap)
library(tidyverse)
library(rmapshaper)
library(smoothr)
```

spatial subsetting

Spatial subsetting is the process of converting a spatial object into a new object containing only the features that *relate* in space to another object. This is analogous the attribute subsetting that we covered last week. There are many ways to spatially subset in R, so we will explore a few.

Let's start by going back to the New Zealand datasets and find all the high points in the state of Canterbury.

The command

```
canterbury <- nz %>%
  filter(Name == "Canterbury")

# subsets nz_heights to just the features that intersect Canterbury
c_height <- nz_height[canterbury, ]

tm_shape(nz) +
  tm_polygons() +
  tm_shape(nz_height) +
  tm_dots(fill = "red")

tm_shape(nz) +
  tm_polygons() +
```

```r
  tm_shape(canterbury) +
  tm_polygons(fill = "blue") +
  tm_shape(c_height) +
  tm_dots(fill = "red")
```

The default is to subset to features that intersect, but we can use other operations, including finding features that do not intersect.

```r
outside_height <- nz_height[canterbury, , op = st_disjoint]

tm_shape(nz) +
  tm_polygons() +
  tm_shape(canterbury) +
  tm_polygons(fill = "blue") +
  tm_shape(outside_height) +
  tm_dots(fill = "red")
```

We can perform the same operations using topological operators. These operators return matrices testing the relationship between features.

```r
sel_sgbp <- st_intersects(x = nz_height, y = canterbury)
sel_sgbp
sel_logical <- lengths(sel_sgbp) > 0

c_height2 <- nz_height[sel_logical, ]

tm_shape(nz) +
  tm_polygons() +
  tm_shape(canterbury) +
  tm_polygons(fill = "blue") +
  tm_shape(c_height2) +
  tm_dots(fill = "red")
```

We can also use the **st_filter** function in *sf*

```r
c_height3 <- nz_height %>%
  st_filter(y = canterbury, .predicate = st_intersects)

tm_shape(nz) +
  tm_polygons() +
  tm_shape(canterbury) +
  tm_polygons(fill = "blue") +
  tm_shape(c_height3) +
  tm_dots(fill = "red")
```

We can change the predicate option to test subset to features that don't intersect

```r
outside_height2 <- nz_height %>%
  st_filter(y = canterbury, .predicate = st_disjoint)

tm_shape(nz) +
  tm_polygons() +
  tm_shape(canterbury) +
  tm_polygons(fill = "blue") +
  tm_shape(outside_height2) +
  tm_dots(fill = "red")
```

buffers

Buffers create polygons representing a set distance from a feature.

```r
seine_buffer <- st_buffer(seine, dist = 5000)

tm_shape(seine_buffer) +
  tm_polygons()
```

unions

As we saw in the last lab, we can spatially aggregate without explicitly asking R to do so.

```r
world %>%
  group_by(continent) %>%
  summarize(population = sum(pop, na.rm = TRUE))
```

What is going on here? Behind the scenes, **summarize()** is using **st_union()** to dissolve the boundaries.

```r
us_west <- us_states %>%
  filter(REGION == "West")

us_west_union <- st_union(us_west)

tm_shape(us_west_union) +
  tm_polygons()
```

st_union() can also take 2 geometries and unite them.

```r
texas <-  us_states %>%
  filter(NAME == "Texas")

texas_union = st_union(us_west_union, texas)

tm_shape(texas_union) +
  tm_polygons()
```

spatial joining

Where attribute joining depends on both data sets sharing a ‘key’ variable, spatial joining uses the same concept but depends on spatial relationships between data sets.

Let’s test this out by creating 50 points randomly distributed across the world and finding out what countries they call in.

```r
set.seed(2018)
bb <- st_bbox(world)

random_df <- data.frame(
  x = runif(n = 10, min = bb[1], max = bb[3]),
  y = runif(n = 10, min = bb[2], max = bb[4])
)

random_points <- random_df %>%
  st_as_sf(coords = c("x", "y")) %>%
  st_set_crs("EPSG:4326")

tm_shape(world) +
  tm_fill() +
  tm_shape(random_points) +
  tm_dots(fill = "red")
```

Let's first use spatial subsetting to find just the countries that contain random points.

```r
world_random <- world[random_points, ]

tm_shape(world) +
  tm_fill() +
  tm_shape(world_random) +
  tm_fill(fill = "red")
```

Now let's perform a spatial join to add the info from each country that a point falls into onto the point dataset.

```r
random_joined  <- st_join(random_points, world)

tm_shape(world) +
  tm_fill() +
  tm_shape(random_joined) +
  tm_dots(fill = "name_long")
```

By default, **st_join** performs a left join. We change this and instead perform an inner join.

```r
random_joined_inner <- st_join(random_points, world, left = FALSE)
```

non-overlapping joins

Sometimes we might want join geographic datasets that are strongly related, but do not have overlapping geometries. To demonstrate this, let's look at data on cycle hire points in London.

```r
tmap_mode("view")

tm_shape(cycle_hire) +
  tm_dots(col = "blue", alpha = 0.5) +
  tm_shape(cycle_hire_osm) +
  tm_dots(col = "red", alpha = 0.5)
```

We can check if any of these points overlap.

```r
any(st_touches(cycle_hire, cycle_hire_osm, sparse = FALSE))
```

Let's say we need to join the 'capacity' variable in 'cycle_hire_osm' onto the official 'target' data in 'cycle_hire'. The simplest method is using the topological operator **st_is_within_distance()**.

```r
sel <- st_is_within_distance(cycle_hire, cycle_hire_osm, dist = 20)

summary(lengths(sel) > 0) #summarizes the number of points within 20 meters
```

Now, we'd like to add the values from 'cycle_hire_osm' onto the 'cycle_hire' points.

```r
z <- st_join(cycle_hire, cycle_hire_osm, st_is_within_distance, dist = 20)

nrow(cycle_hire)
nrow(z)
```

Note: the number of rows of the join is larger than the number of rows in the original dataset. Why? Because some points in 'cycle_hire' were within 20 meters of multiple points in 'cycle_hire_osm'. If we wanted to aggregate so we have just one value per original point, we can use the aggregation methods from last week.

```r
z <- z %>%
  group_by(id) %>%
  summarise(capacity = mean(capacity))
```

spatial aggregation

Similar to attribute data aggregation, spatial data aggregation condenses data (we end up with fewer rows than we started with).

Let's say we wanted to find the average height of high point in each region of New Zealand. We could use the **aggregate** function in base R.

```r
nz_agg = aggregate(x = nz_height, by = nz, FUN = mean)
```

The result of this is an object with the same geometries as the aggregating feature data set (in this case 'nz').

```r
nz_agg
```

We could also use a **sf**/**dplyr** approach.

```r
nz_agg <- st_join(nz, nz_height) %>%
  group_by(Name) %>%
  summarise(elevation = mean(elevation, na.rm = TRUE))
nz_agg
```

joining incongruent layers

We might want to aggregate data to geometries that are not congruent (i.e. their boundaries don't line up). This causes issues when we think about how to summarize associated values.

```r
head(incongruent)
head(aggregating_zones)

tm_shape(incongruent) +
  tm_polygons() +
  tm_shape(aggregating_zones) +
  tm_borders(col = "red")
```

The simplest method for dealing with this is using area weighted spatial interpolation which transfers values from the 'incongruent' object to a new column in 'aggregating_zones' in proportion with the area of overlap.

```r
iv <- incongruent["value"]
agg_aw <- st_interpolate_aw(iv, aggregating_zones, extensive = TRUE)

tm_shape(agg_aw) +
  tm_fill(fill = "value")
```

distance relationships

While topological relationships are binary (features either intersect or don't), distance relationships are continuous.

We can use the following to find the distance between the highest point in NZ and the centroid of the Canterbury region.

```r
nz_highest <- nz_height %>%
  slice_max(n = 1, order_by = elevation)

canterbury_centroid = st_centroid(canterbury)
st_distance(nz_highest, canterbury_centroid)
```

Note: this function returns distances with units (yay!) and as a matrix, meaning we could find the distance between many locations at once.

simplification

Simplification generalizes vector data (polygons and lines) to assist with plotting and reducing the amount of memory, disk space, and network bandwidth to handle a dataset.

Let's try simplifying the US states using the Douglas-Peucker algorithm. GEOS assumes a projected CRS, so we first need to project the data, in this case into the US National Atlas Equal Area (epsg = 2163)

```r
us_states2163 = st_transform(us_states, "EPSG:2163")
us_states2163 = us_states2163

us_simp1 = st_simplify(us_states2163, dTolerance = 100000)  # 100 km
```

```r
tm_shape(us_states_simp1) +
  tm_polygons()
```

To preserve the states' topology let's use a simplify function from **rmapshaper** which uses Visalingam's algorithm.

```r
#install.packages('rmapshaper')
library(rmapshaper)

# proportion of points to retain (0-1; default 0.05)
us_states_simp2 = rmapshaper::ms_simplify(us_states2163, keep = 0.01,
                                          keep_shapes = TRUE)

tm_shape(us_states_simp2) +
  tm_polygons()
```

Instead of simplifying, we could try smoothing using Gaussian kernel regression.

```r
us_states_simp3 = smoothr::smooth(us_states2163, method = 'ksmooth', smoothness = 6)

tm_shape(us_states_simp3) +
  tm_polygons()
```

centroids

Centroids identify the center of a spatial feature. Similar to taking an average, there are many ways to compute a centroid. The most common is the *geographic* centroid.

```r
nz_centroid <- st_centroid(nz)

tm_shape(nz) +
  tm_fill() +
  tm_shape(nz_centroid) +
  tm_dots()
```

Sometimes centroids fall outside of the boundaries of the objects they were created from. In the case where we need them to fall inside of the feature, we can use *point on surface* methods.

```r
nz_pos <- st_point_on_surface(nz)

tm_shape(nz) +
  tm_fill() +
  tm_shape(nz_centroid) +
  tm_dots() +
  tm_shape(nz_pos) +
  tm_dots(fill = "red")
```