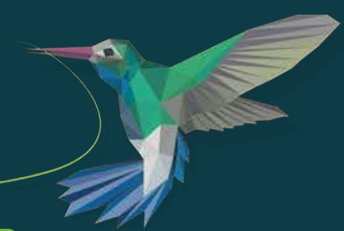
Geoprocessing for Vectors and Rasters



ZevRoss
Know Your Data

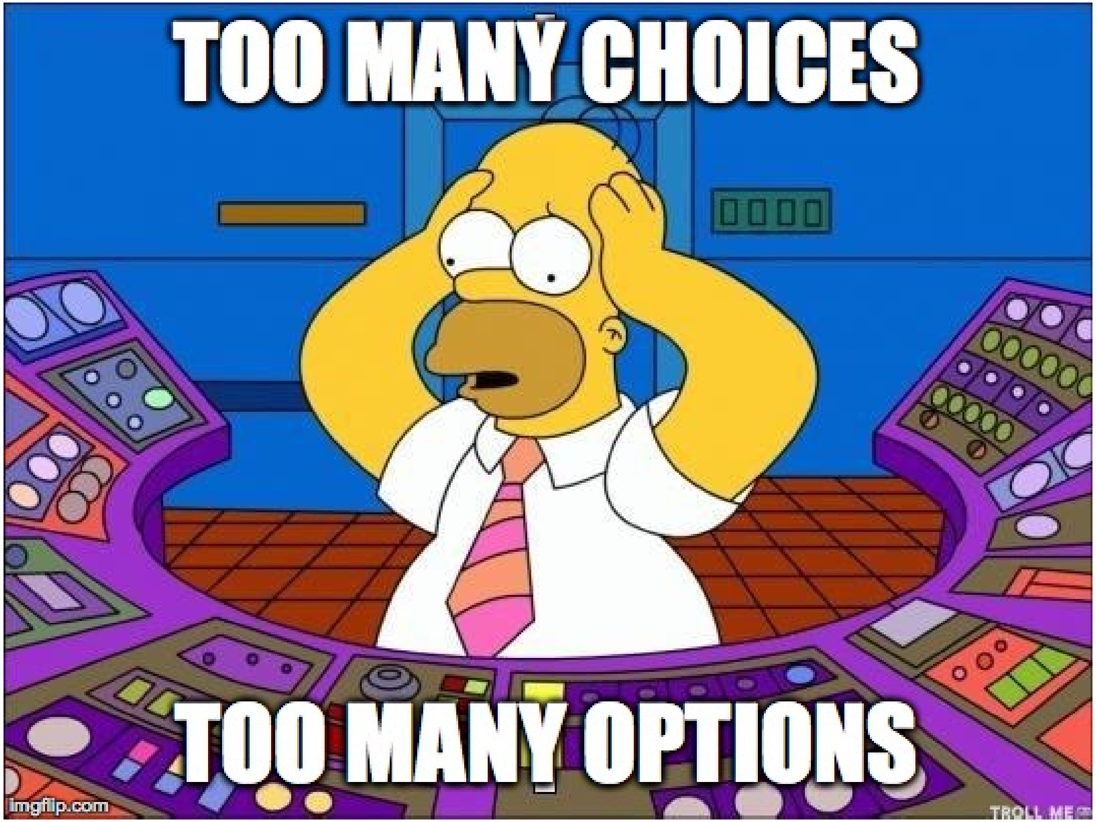
Putting all the pieces together for the fun stuff

Meaningfully manipulating your geospatial data

A three-part section:

- Single vector layers
- Multiple vector layers
- Raster layers

There are several dozen functions in each category



The plan

- Demonstrate some of the most important functions using a real-world, mini-analysis
- Provide a quick demonstration of important functions that are not included in the mini-analysis

Our mini-analysis

What influences air quality in New York City?

A common air quality modeling approach

- Collect measurements at air monitors
- Compute road density, landuse and other variables near each monitor
- Look at the relationship between concentrations and road density etc.

Start with the air quality monitors

Image source

New York City has one of the largest urban air monitoring networks in the world

Data from the NYC Dept. of Health, New York City Community Air Survey (NYCCAS).

More detail can be found here.

Take a look at our air quality data

$PM_{2.5}$ refers to particles in the air (soot)

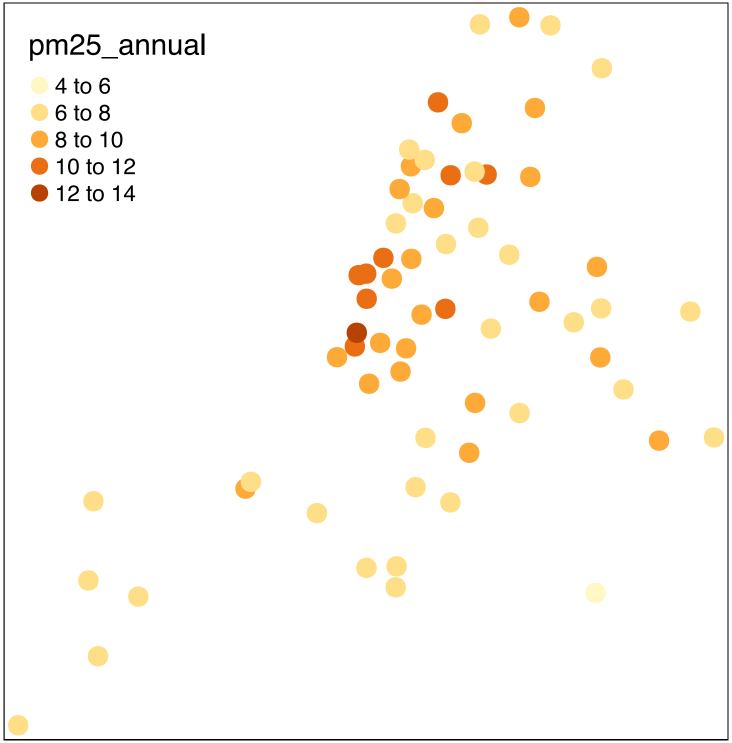
```
monitors <- read_sf("monitors.gpkg")
```

```
library(dplyr)
glimpse(monitors)
```

```
## Observations: 64
## Variables: 4
## $ site_id     <dbl> 228, 952, 2269, 2496, 2596, 2818…
## $ reference   <dbl> 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,…
## $ pm25_annual <dbl> 6.473097, 6.591441, 6.107921, 5.…
## $ geom        <POINT [US_survey_foot]> POINT (918300…
```

Map the air quality data

```
tm_shape(monitors) +
  tm_dots("pm25_annual", size = 0.5)
```



Let's add a little context by mapping the counties with the monitors

Read the county/borough data directly from nyc.gov

```
counties <- read_sf("http://bit.ly/39MxcnC")
```

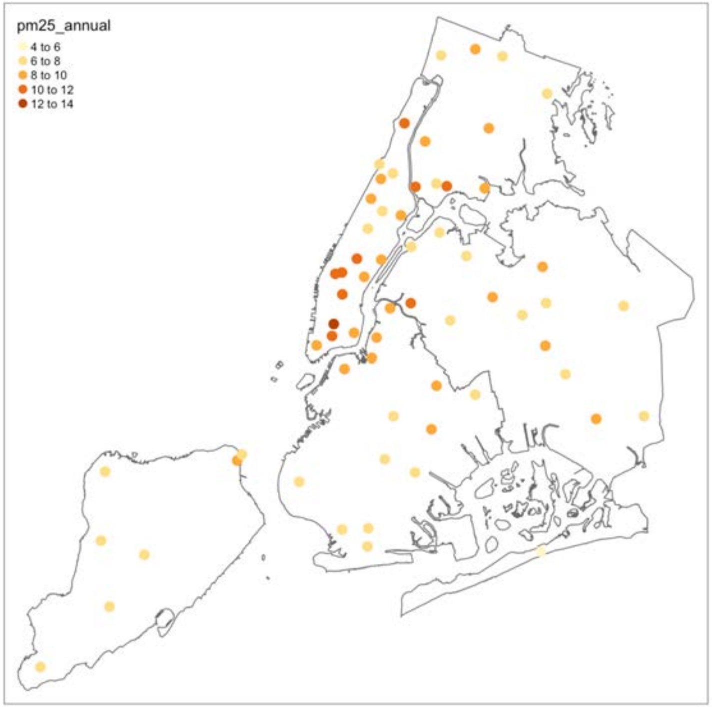
Data is here.

Here are our counties (also referred to as "boroughs")

```
tm_shape(counties) + tm_polygons() +
  tm_text("BoroName", size = 1)
```

Map the air quality data and the counties together

```
tm_shape(counties) + tm_borders() +
  tm_shape(monitors) + tm_dots("pm25_annual", size = 0.5)
```



By the way, that last map worked but why doesn't this?

```
plot(st_geometry(counties))
plot(st_geometry(monitors), add = TRUE)
```

CRS mismatch!

```
st_crs(monitors)
```

```
## Coordinate Reference System:
##   No EPSG code
##   proj4string: "+proj=lcc +lat_1=41.03333333333333 +lat_2=40.6666
```

```
st_crs(counties)
```

```
## Coordinate Reference System:
##   EPSG: 4326
##   proj4string: "+proj=longlat +datum=WGS84 +no_defs"
```

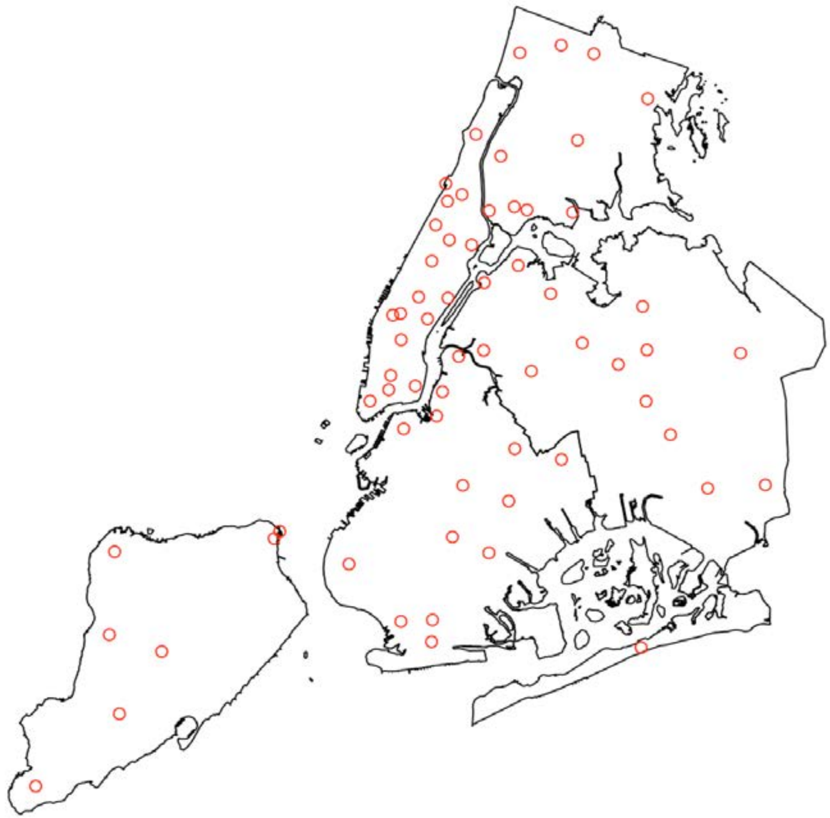
We will use a consistent, projected CRS

Long Island State Plane, EPSG 2908

```
counties <- counties %>%
  st_transform(crs = st_crs(monitors))
```

Try the map again

```
plot(st_geometry(counties))
plot(st_geometry(monitors), add = TRUE)
```



Introducing our candidate "predictor" variables

Road layer lines

```
roads <- read_sf("roads.gpkg")
```

Data from here.

Road map

```
tm_shape(counties) + tm_borders(col = "red") +
  tm_shape(roads) + tm_lines(col = "grey")
```

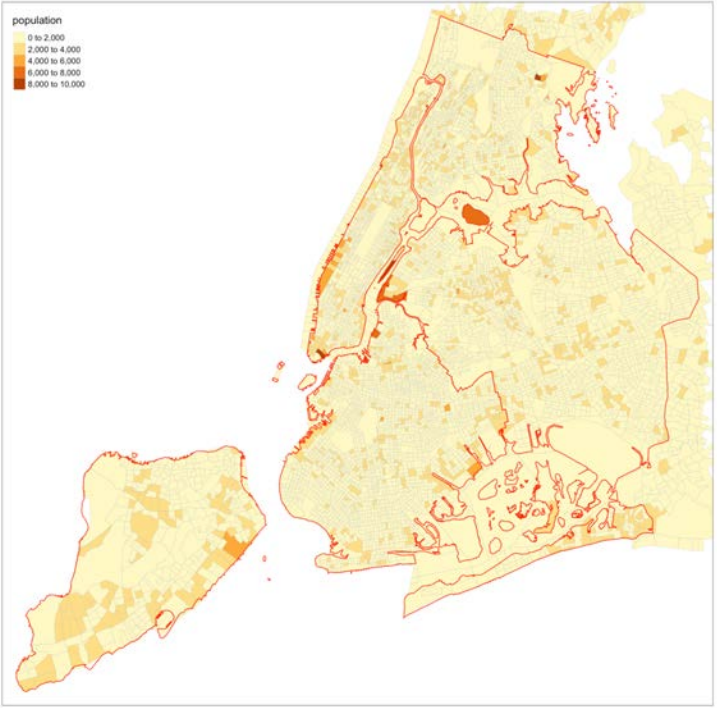
Census data (population polygons)

```
population <- read_sf("population.shp")
```

Data collected with {tidycensus}.

Census data map

```
tm_shape(population) +
  tm_polygons("population", border.col = "grey", lwd = 0.25)
  tm_shape(counties, is.master = TRUE) +
  tm_borders(col = "red")
```



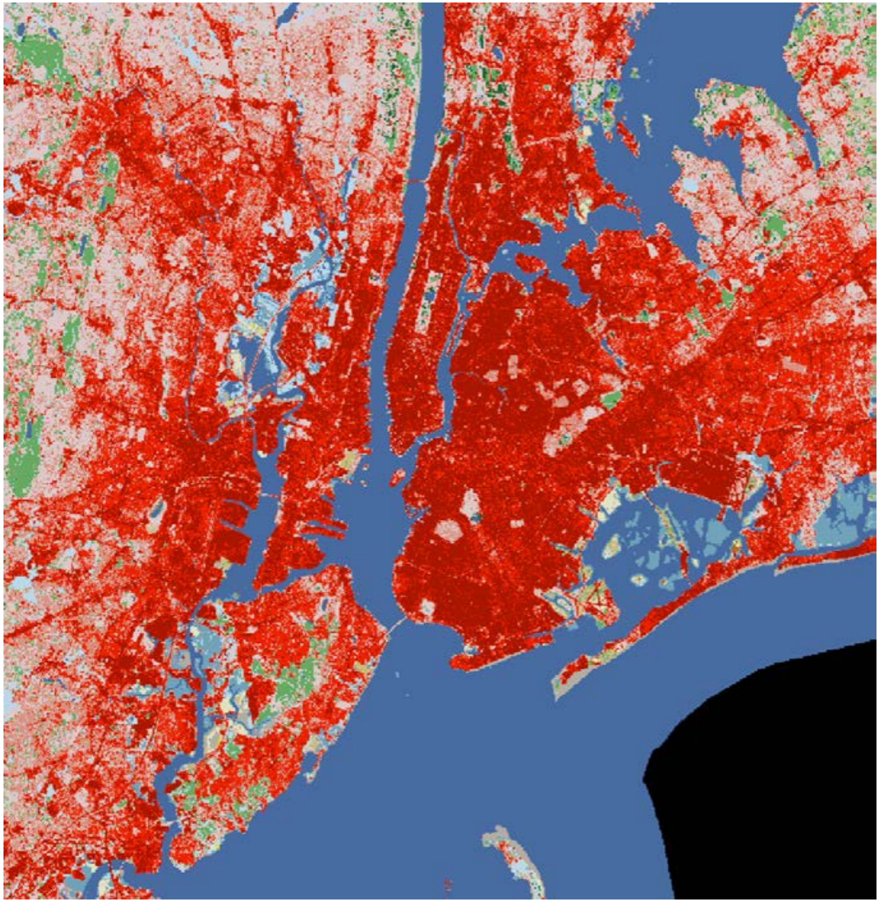
Land use raster

```r
landuse <- raster("landuse.tif")
```

Data collected using {FedData}.

Land use raster map

```
plot(landuse)
```



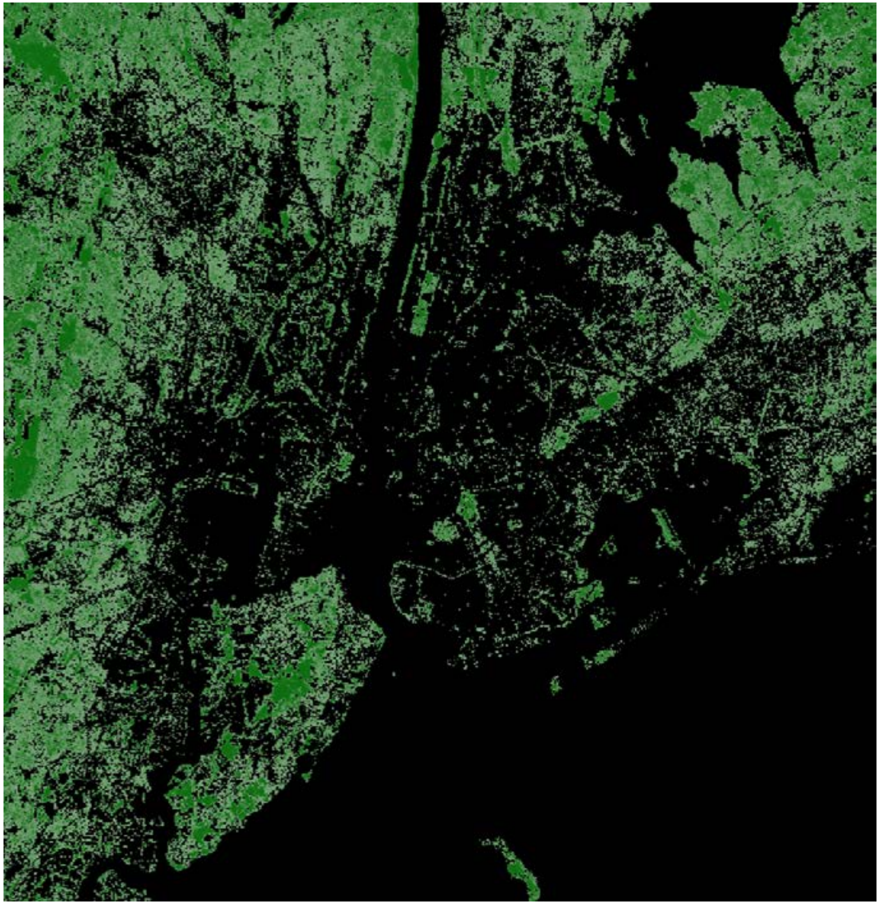
Canopy raster

```
canopy <- raster("canopy.tif")
```

Data collected using {FedData}.

Canopy raster map

```
plot(canopy)
```



Goal: characterize areas around monitors

- Compute road density
- Compute distance to the nearest road
- Compute population
- Compute the amount of "high intensity" developed land
- Compute average tree canopy

Single-layer geoprocessing for vectors

Examples of available functions

```
st_union()
st_centroid()
st_convex_hull()
st_buffer()
st_cast()
st_simplify()
```

Favorites that are not part of our mini-analysis...

Get the centroids with `st_centroid()`

```
cent <- st_centroid(counties)
```

```
tm_shape(counties) + tm_borders() +
  tm_shape(cent) + tm_dots(size = 0.5, col = "red")
```

Put a "hull" around geometries with `st_convex_hull()`

```
hull <- st_convex_hull(counties)
```

```
tm_shape(counties) + tm_polygons() + tm_shape(hull) +
  tm_polygons("BoroName", alpha = 0.3) + tm_layout(frame = F
```

Combine multiple geometries into one

```
counties_as_one <- st_union(counties)
```

```
counties_as_one
```

```
## Geometry set for 1 feature
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 913174.7 ymin: 120124.9 xmax: 1067382 ymax:
## epsg (SRID):    NA
## proj4string:    +proj=lcc +lat_1=41.03333333333333 +lat_2=40.6666
```

Reapply hull

```
hull <- st_convex_hull(counties_as_one)
```

```
tm_shape(counties) + tm_polygons() + tm_shape(hull) +
  tm_borders(col = "orange") + tm_layout(frame = FALSE)
```

For the analysis of air quality the function we need is `st_buffer()`

Create a 500 meter buffer around the monitors

Then compute, for example, road density within the buffer

The basic syntax is

```
st_buffer(geo, distance)
```

The distance units come from the geography CRS

In our case this is feet

```
st_crs(monitors)
```

proj4string: "+proj=lcc +lat_1=41.03333333333333 +lat_2=40.66666666666666 +lat_0=40.16666666666666 +lon_0=-74 +x_0=300000 +y_0=0 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=us-ft +no_defs"

You can access this directly with:
`sf::crs_parameters(st_crs(schools))$ud_unit`

To get meters from feet multiple by 3.28

```
monitor_buffers <- st_buffer(monitors, 500 * 3.28)
```

Our monitor buffers

```
tm_shape(counties) + tm_borders() +
  tm_shape(monitor_buffers) + tm_polygons(col = "red") +
  tm_shape(monitors) + tm_dots(size = 0.1, col = "yellow")
```

By the way, in terms of naming objects for this section

Final tables will be prefixed with `monitor_` (e.g., `monitor_roads`, `monitor_canopy` etc) open_exercise(7) and do activities 1-3 only

Geoprocessing with multiple vector layers

Tons of great functions for geoprocessing with two layers

```
st_join()
st_distance()
st_nearest_feature()
st_nearest_points()
st_combine()
st_intersection()
st_union()
st_crop()
st_intersects()
st_contains()
st_touches()
```

Some of these functions return a geometry

```
st_join()
st_nearest_points()
st_combine()
st_intersection()
st_union()
st_crop()
```

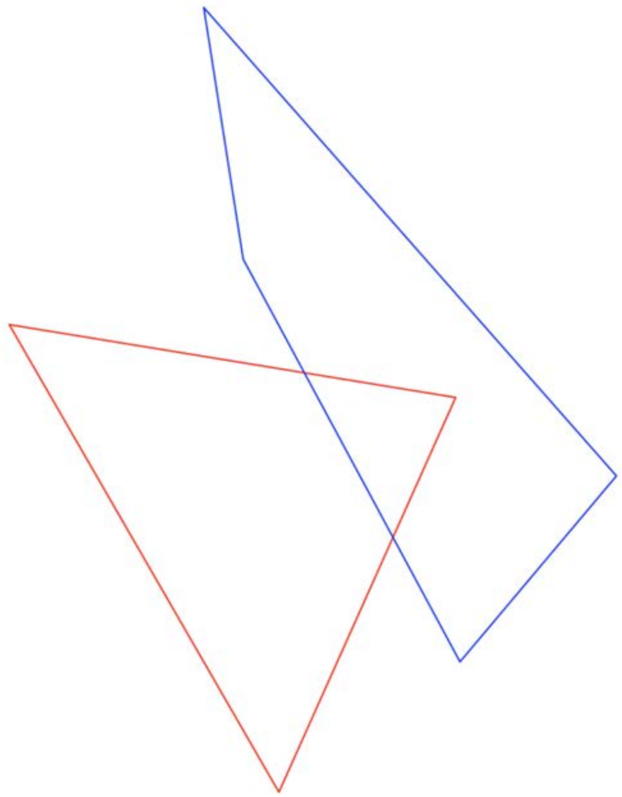
And some return an object describing relationships

```
st_intersects()
st_contains()
st_touches()
st_crosses()
st_distance()
st_nearest_feature()
```

Examples of functions that return a geometry

For illustration, start with two rectangles

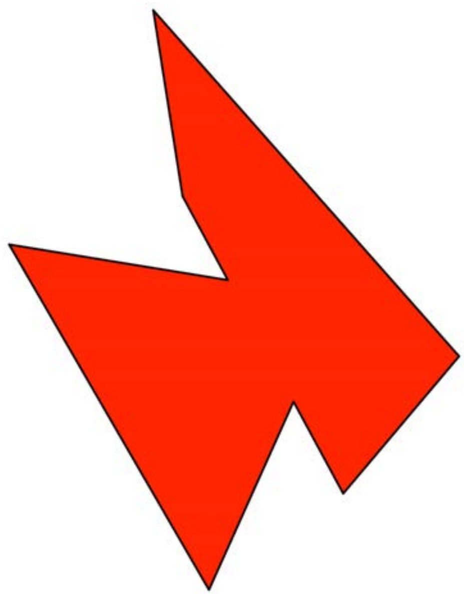
```
plot(polys, border = "grey")
plot(st_geometry(poly1), add = TRUE, border = "red")
plot(st_geometry(poly2), add = TRUE, border = "blue")
```



Combine multiple geometries into one, dissolved, geometry with `st_union()`

```
union <- st_union(poly1, poly2)
```

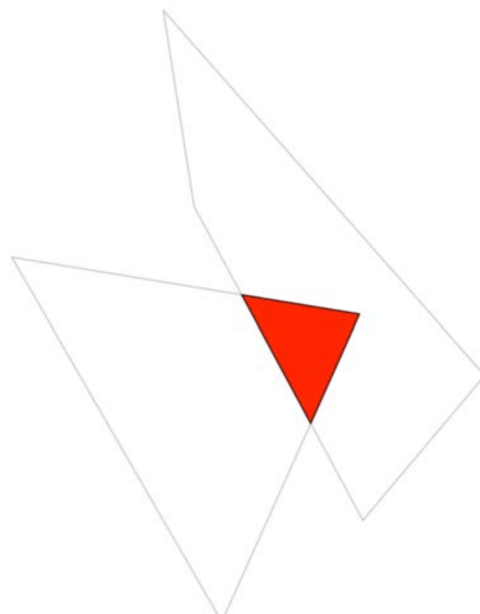
```
plot(polys, border = "grey")
plot(st_geometry(union), add = TRUE, col = "red", lwd = 2)
```



Compute the intersection between geometries with `st_intersection()`

```
intersection <- st_intersection(poly1, poly2)
```

```
plot(polys, border = "grey")
plot(st_geometry(intersection), add = TRUE, col = "red")
```



Examples of functions that return an object describing relationships

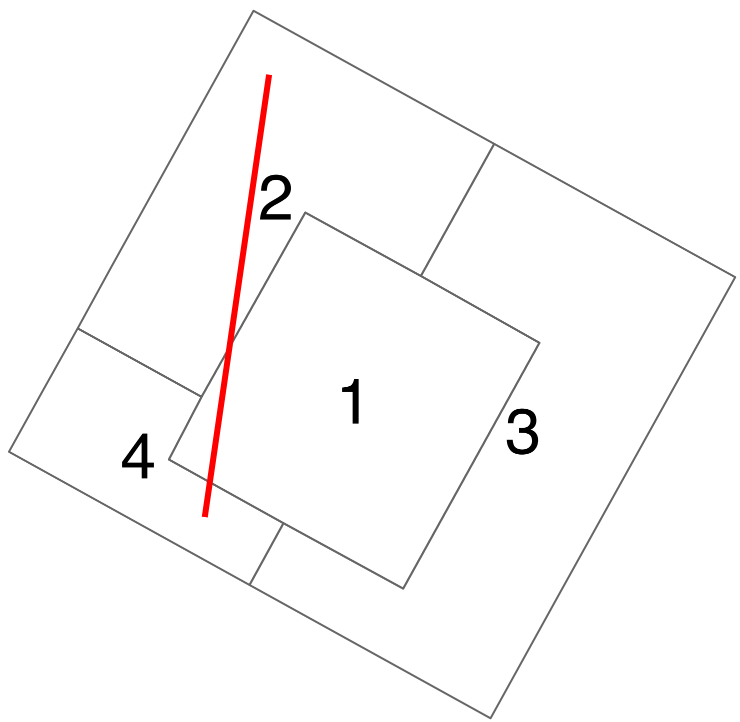
```
st_intersects()
st_contains()
st_touches()
st_crosses()
st_distance()
st_nearest_feature()
```

You can find visual descriptions of the relationships...

Here or here.

For our examples, we'll use two objects

- An {sf} object with four polygons (`poly`)
- An {sf} object with one line (`line`)



Most of these functions can return either a …

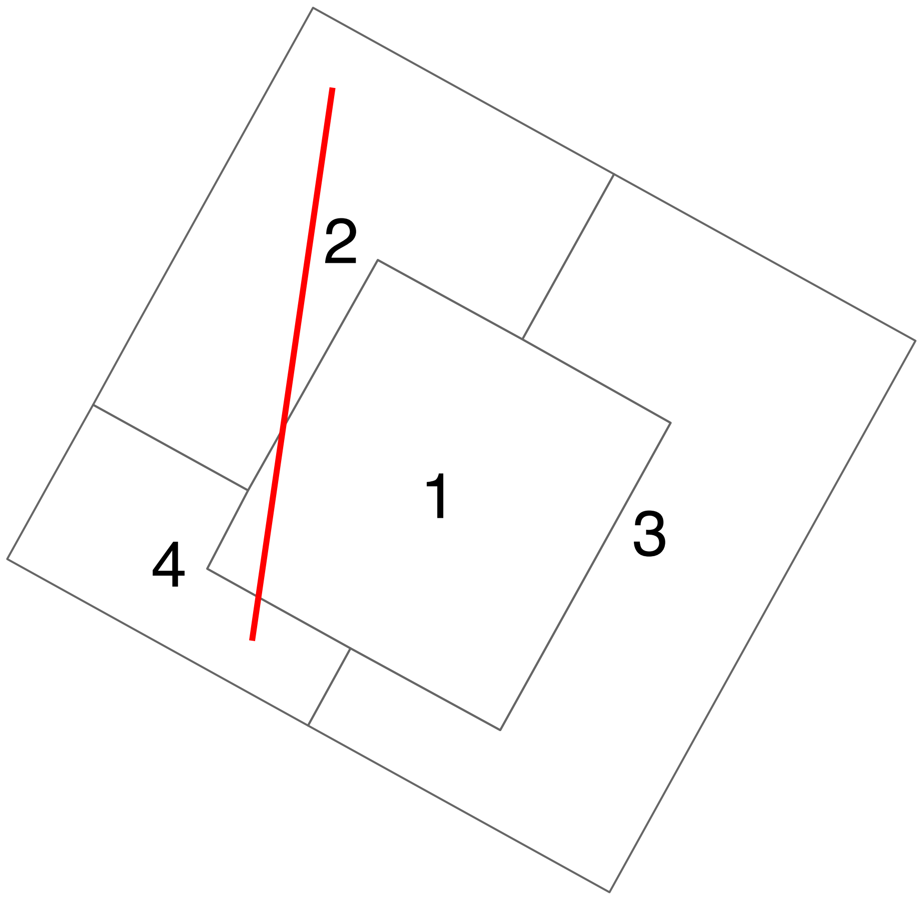
A sparse index list or

A dense logical matrix

These are called "binary logical operations"

```
st_intersects()
st_touches()
st_crosses()
st_within()
st_contains()
st_overlaps()
# And more!
```

Back to our example geometries



Test if features cross with `st_crosses()`

```
st_crosses(line, poly)
```

```
## Sparse geometry binary predicate list of length 1, where the pred
##  1: 1, 2, 4
```

```
st_crosses(poly, line)
```

```
## Sparse geometry binary predicate list of length 4, where the pred
##  1: 1
##  2: 1
##  3: (empty)
##  4: 1
```

Test if the features intersect with `st_intersects()`

By the way, note...

- `st_intersection()` returns a geometry
- `st_intersects()` returns an object of relationships

Test if the features intersect with `st_intersects()`

```
st_intersects(line, poly)
```

```
## Sparse geometry binary predicate list of length 1, where the pred
##  1: 1, 2, 4
```

```
st_intersects(poly, line)
```

```
## Sparse geometry binary predicate list of length 4, where the pred
##  1: 1
##  2: 1
##  3: (empty)
##  4: 1
```

Let's make this a little more interesting with the roads and population

```
road_pop_index <- st_intersects(roads, population)
```

The default for these functions is to return a sparse list

```
road_pop_index
```

```
## Sparse geometry binary predicate list of length 21464, where the
## first 10 elements:
##  1: 6454, 6457, 6458, 6467, 6468
##  2: 1526
##  3: 968
##  4: 1999
##  5: 3195
##  6: 1526
##  7: 6368
##  8: 1941, 1999
##  9: 4923
##  10: 3095, 3222
```

The list length is the same as the number of features (in the first object)

```
nrow(roads)
```

```
## [1] 21464
```

```
length(road_pop_index)
```

```
## [1] 21464
```

You can extract pieces like an R list

```r
# Results for polygon 1
road_pop_index[[1]]
```

```
## [1] 6454 6457 6458 6467 6468
```

```r
# Results for polygon 3
road_pop_index[[3]]
```

```
## [1] 968
```

Use `lengths()` to count how many intersections in this case

Zero means no intersection

For example...

```
number_of_intersections <- lengths(road_pop_index)
```

```
head(number_of_intersections)
```

```
## [1] 5 1 1 1 1 1
```

Are there roads that don't intersect the census polygons?

```
roads_no_intersect <- filter(roads,
                             number_of_intersections == 0)
```

```
nrow(roads_no_intersect)
```

```
## [1] 38
```

Where are these roads that don't intersect?

If you prefer, you can return dense logical matrix from binary logical operations

```
mat <- st_intersects(poly, line, sparse = FALSE)
mat
```

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

Back to our mini-analysis

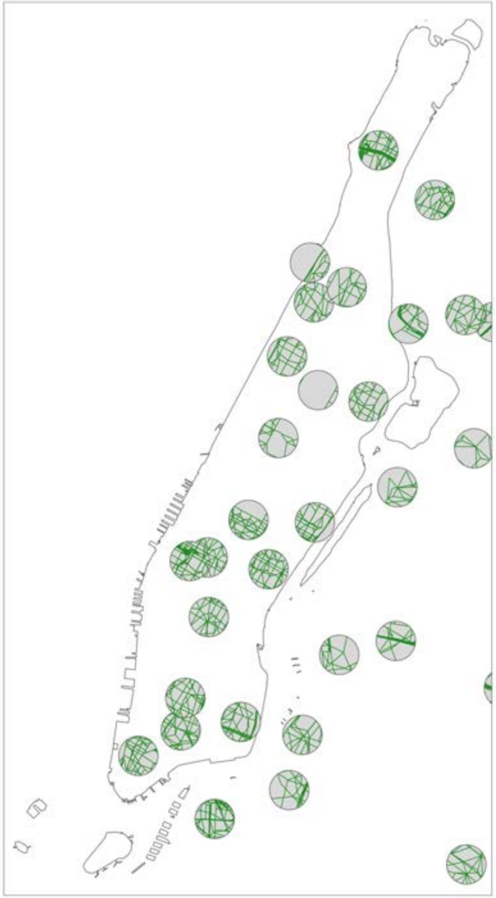
How might we compute road density in the monitor buffers?

Compute the intersection between the lines and polygons

```
roads_in_buffer <- st_intersection(monitor_buffers, roads)
```

Map of the roads in the buffers

Zoomed in to Manhattan



The resulting geometry is lines and includes attributes from both tables

```
roads_in_buffer[,1:8] %>%
  glimpse()
```

```
## Observations: 2,316
## Variables: 9
## $ site_id     <dbl> 11389, 2496, 6689, 2496, 11389, …
## $ reference   <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
## $ pm25_annual <dbl> 9.024395, 5.873043, 7.525750, 5.…
## $ Year_Recor  <dbl> 2017, 2017, 2017, 2017, 2017, 20…
## $ State_Code  <dbl> 36, 36, 36, 36, 36, 36, 36, 36, …
## $ Route_ID    <chr> "300258011", "257268011", "25625…
## $ Begin_Poin  <dbl> 3.58, 3.10, 1.40, 3.48, 3.60, 1.…
## $ End_Point   <dbl> 3.60, 3.20, 1.50, 3.50, 3.70, 1.…
## $ geom        <LINESTRING [US_survey_foot]> LINESTRI…
```

So the final step would be to add the road length and sum by site ID

```
roads_in_buffer <- roads_in_buffer %>%
  mutate(length = st_length(geom))
```

```
monitor_roads <- roads_in_buffer %>%
  group_by(site_id) %>%
  summarise(total_roads = sum(length)) %>%
  st_drop_geometry()
```

Road length/density in the buffers

```
glimpse(monitor_roads)
```

```
## Observations: 64
## Variables: 2
## $ site_id     <dbl> 228, 952, 2269, 2496, 2596, 2818…
## $ total_roads [US_survey_foot] 2772.254 [US_survey_f…
```

Compute distance to the nearest road

We could use `st_distance()`

```
dist <- st_distance(monitors, roads)
```

But `st_distance()` computes a matrix of distances from all features to all features

```r
dim(dist)
```

```
## [1]    64 21464
```

For speedier results you can:

- Find the nearest road first
- Then compute the distance to **just this road**

First use `st_nearest_feature()`

```
feat <- st_nearest_feature(monitors, roads)
```

```
# Index of nearest feature
feat
```

```
##  [1]  6041 13014  9259 17955 11994  4240 12642 18540 20322  3584
## [12]  9806 13617  8865  7471  2585 10303 12548  3242 16978 19174
## [23]  7501  7350  3244 15063 19556  3914 17863  5847  8783 18704
## [34] 18431  1085 10104  7640  5472 21371  4829    81  6201  5720
## [45] 18125 14457 13931 14542 15712  2354  2444  7390 17884 15208
## [56] 13927 12453  6955  2316 10638  5805  2340  7142 12690
```

And then compute the distance from each monitor to its nearest road

Use the `by_element = TRUE` argument so that the distance is only measured from the 1st monitor to the 1st road, 2nd to 2nd and so on.

```
min_dist <- st_distance(monitors, roads[feat,],
                        by_element = TRUE)
```

Create the minimum distance to road table

```
monitor_road_mindist <- monitors %>%
  mutate(road_mindist = min_dist) %>%
  select(-pm25_annual, -reference) %>%
  st_drop_geometry()
```

```
head(monitor_road_mindist)
```

```
## # A tibble: 6 x 2
##   site_id     road_mindist
##     <dbl> [US_survey_foot]
## 1     228        838.19670
## 2     952         16.11883
## 3    2269         96.42880
## 4    2496        125.59963
## 5    2596        252.24296
## 6    2818       1250.33448
```

How would we compute total population?

Remember that our census areas are polygons

```
st_geometry(population) %>%
  plot()
```

Easiest solution would be to simply use the population from the underlying census polygon

To do this you can use a "spatial" join with `st_join()`

```
st_join(monitors, population) %>%
  glimpse()
```

```
## Observations: 64
## Variables: 9
## $ site_id     <dbl> 228, 952, 2269, 2496, 2596, 2818…
## $ reference   <dbl> 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,…
## $ pm25_annual <dbl> 6.473097, 6.591441, 6.107921, 5.…
## $ geom        <POINT [US_survey_foot]> POINT (918300…
## $ GEOID       <chr> "360850244013", "360850170083", …
## $ NAME        <chr> "Block Group 3, Census Tract 244…
## $ variable    <chr> "B01001_001", "B01001_001", "B01…
## $ population  <dbl> 2816, 2899, 817, 1733, 1194, 200…
## $ moe         <dbl> 544, 587, 135, 326, 311, 243, 31…
```

The default for `st_join()` is to join if they intersect

Scientifically there is a problem with this approach, though

Census areas vary in size due to population

Lower population density in an area results in a larger census area

This means that if you use only the underlying polygon the "population" will be essentially the same wherever you are!

A better approach is to use the buffers

- Sum the population from all census geography in the buffer
- But do it proportionally by area. In other words, if 10% of a polygon is in the buffer then include 10% of the population

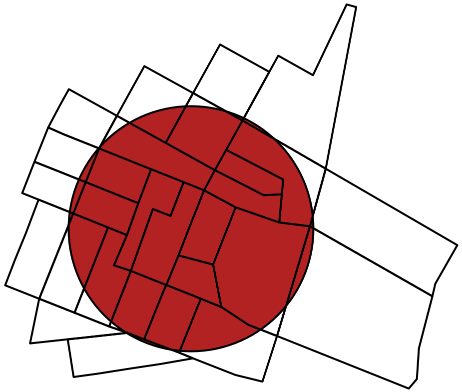
Add the full area as a variable to census polygons

```
population <- population %>%
  mutate(full_area = st_area(geom))
```

Do the intersection

```
population_buffer <- st_intersection(monitor_buffers,
                                     population)
```

The polygons are clipped to the buffers



Add the new area (since some areas get clipped)

```
population_buffer <- population_buffer %>%
  mutate(part_area = st_area(geom))
```

Compute the area proportion and proportional population

```
population_buffer <- population_buffer %>%
  mutate(
    prop_area = part_area/full_area,
    buffer_pop = population * prop_area
  )
```

Sum the population by buffer

```
monitor_population <- population_buffer %>%
  group_by(site_id) %>%
  summarise(population = sum(buffer_pop)) %>%
  st_drop_geometry()
```

open_exercise(7) and do activities 4-10 only

Geoprocessing with rasters

Lots of great functions for rasters as well!

```
reclassify()
extract()
calc()
crop()
mask()
trim()
overlay()
clump()
terrain()
zonal()
focal()
layerize()
aggregate()
```

Nearly all of these functions return a raster

For our analysis we'll introduce `layerize()` and `extract()`, `mask()`, `crop()` and `calc()`

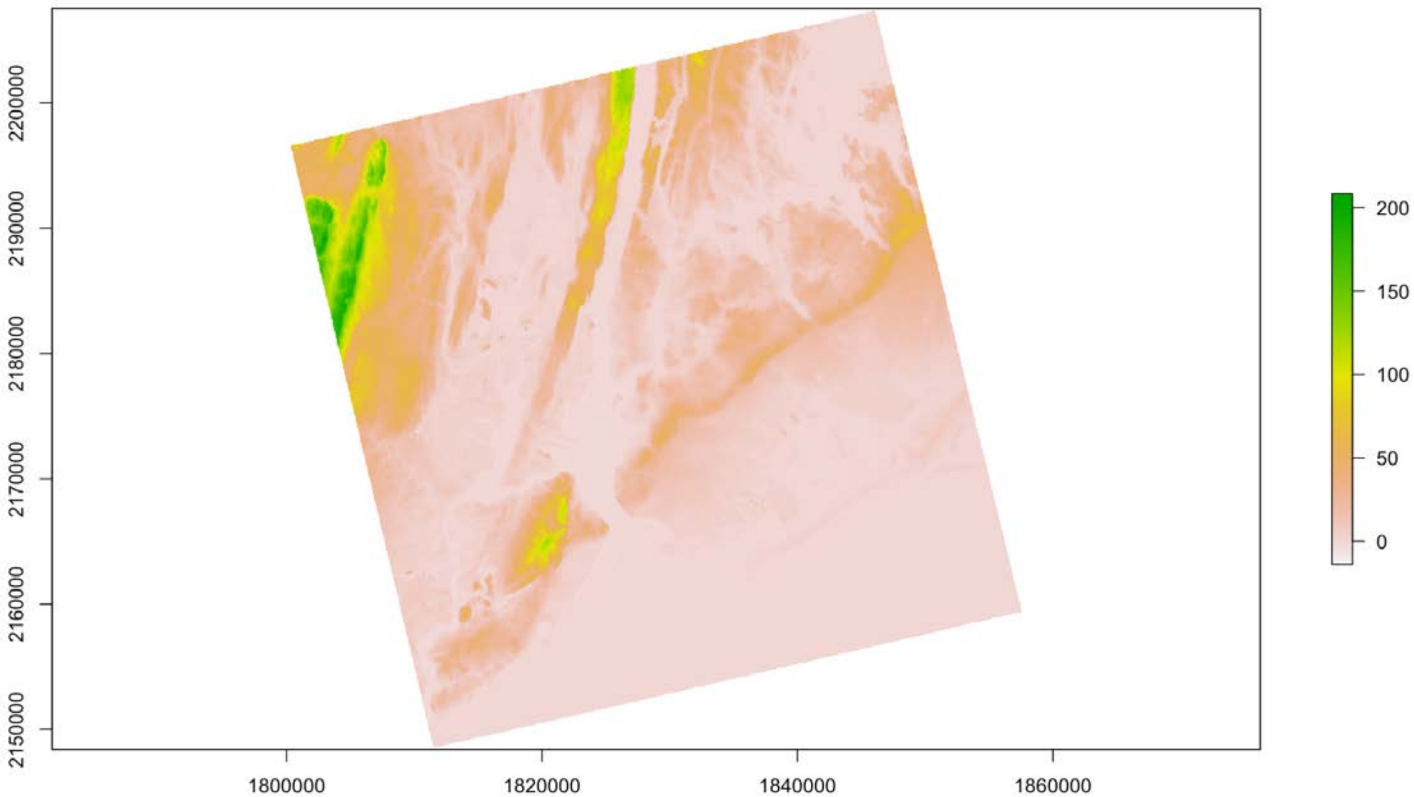
First, a few favorite functions that are not part of the analysis

For "bonus" functions we will use elevation data

```
elevation <- raster("elevation.tif")
```

Plot elevation

```
plot(elevation)
```



Bonus functions: Raster math

There are three approaches you can use to recalculate values

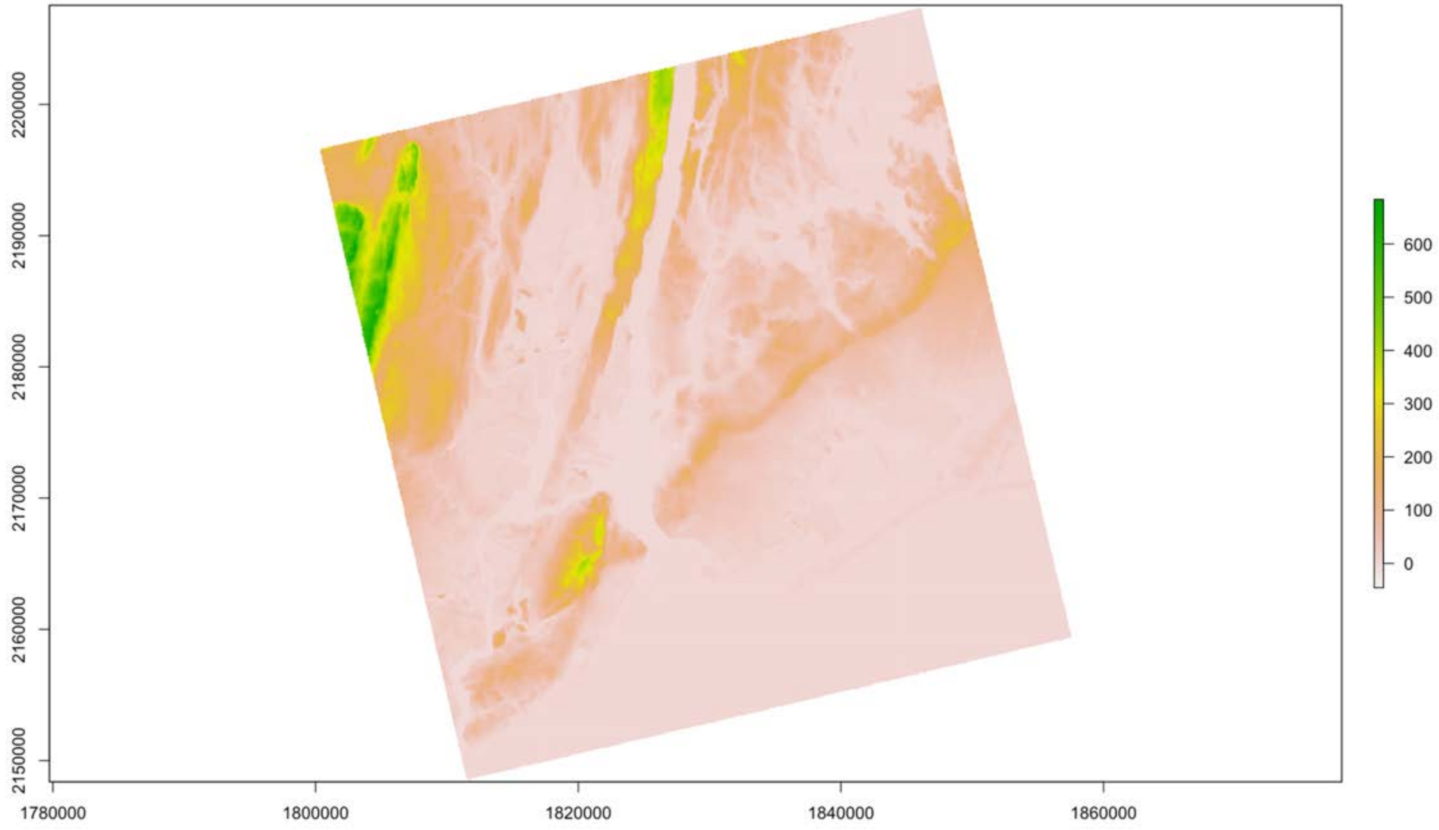
For simple raster arithmetic

For example, convert meters to feet by multiplying by 3.28

```
elevation_feet <- elevation * 3.28
```

Plot of raster in feet

```
plot(elevation_feet)
```



To apply a function `calc()`

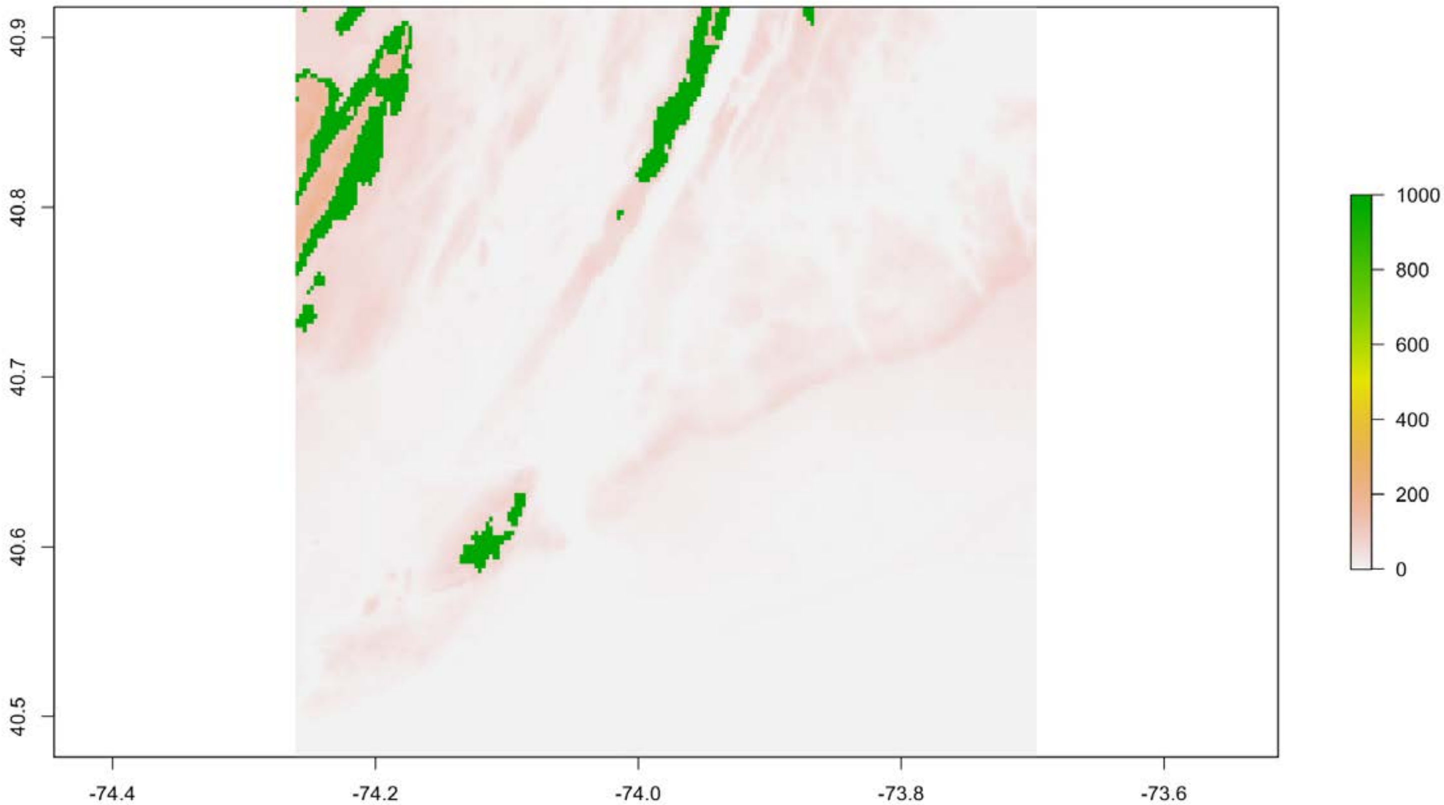
Can be faster with complex formulas and large datasets

```r
f <- function(x) {x[x>75 & x<125] <- 1000; return(x)}
```

```r
elevation_odd <- calc(elevation, fun = f)
```

Plot odd raster

```
plot(elevation_odd)
```



For raster calculations with multiple rasters use `overlay()`

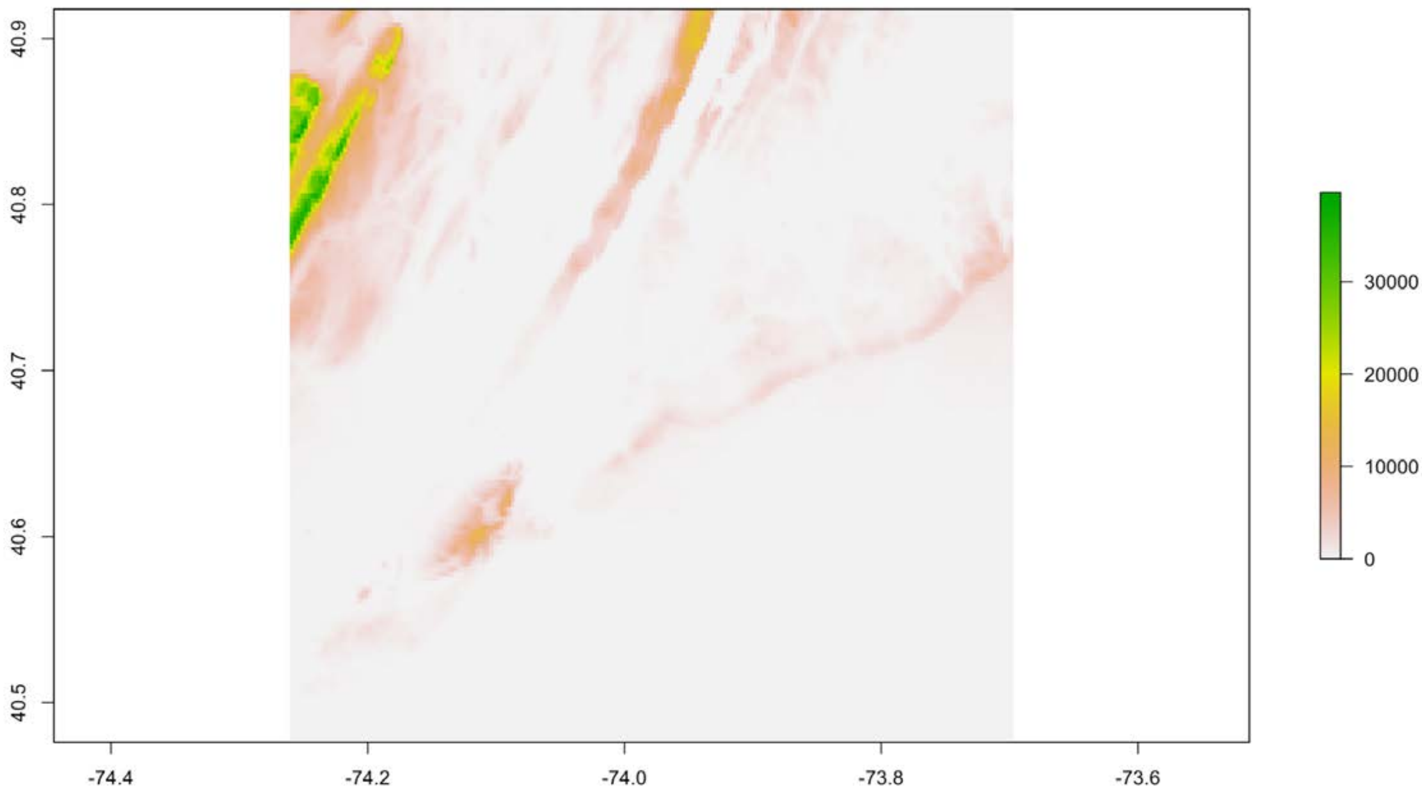
For simplicity, I'm cheating and using the same raster twice

```
f <- function(x,y){return(x * y)}
```

```
elevation_squared <- overlay(elevation, elevation, fun = f)
```

Plot the overlay result

```
plot(elevation_squared)
```



All three approaches can also be applied to a `RasterBrick` or `RasterStack`

Bonus functions: `aggregate()` to reduce resolution

Elevation layer has nearly 5 million cells

```
ncell(elevation)%>%
  format(big.mark   = ",") # format the number
```

```
## [1] "4,952,808"
```

```
# Cells are not square because the raster was projected
res(elevation) # meters
```

```
## [1] 22.7 30.3
```

Reduce resolution by factor of 10

```
lowres <- aggregate(elevation, fact = 10, fun = mean)
```

Lower resolution canopy is less than 50 thousand cells

```
ncell(lowres) %>%
  format(big.mark   = ",")
```

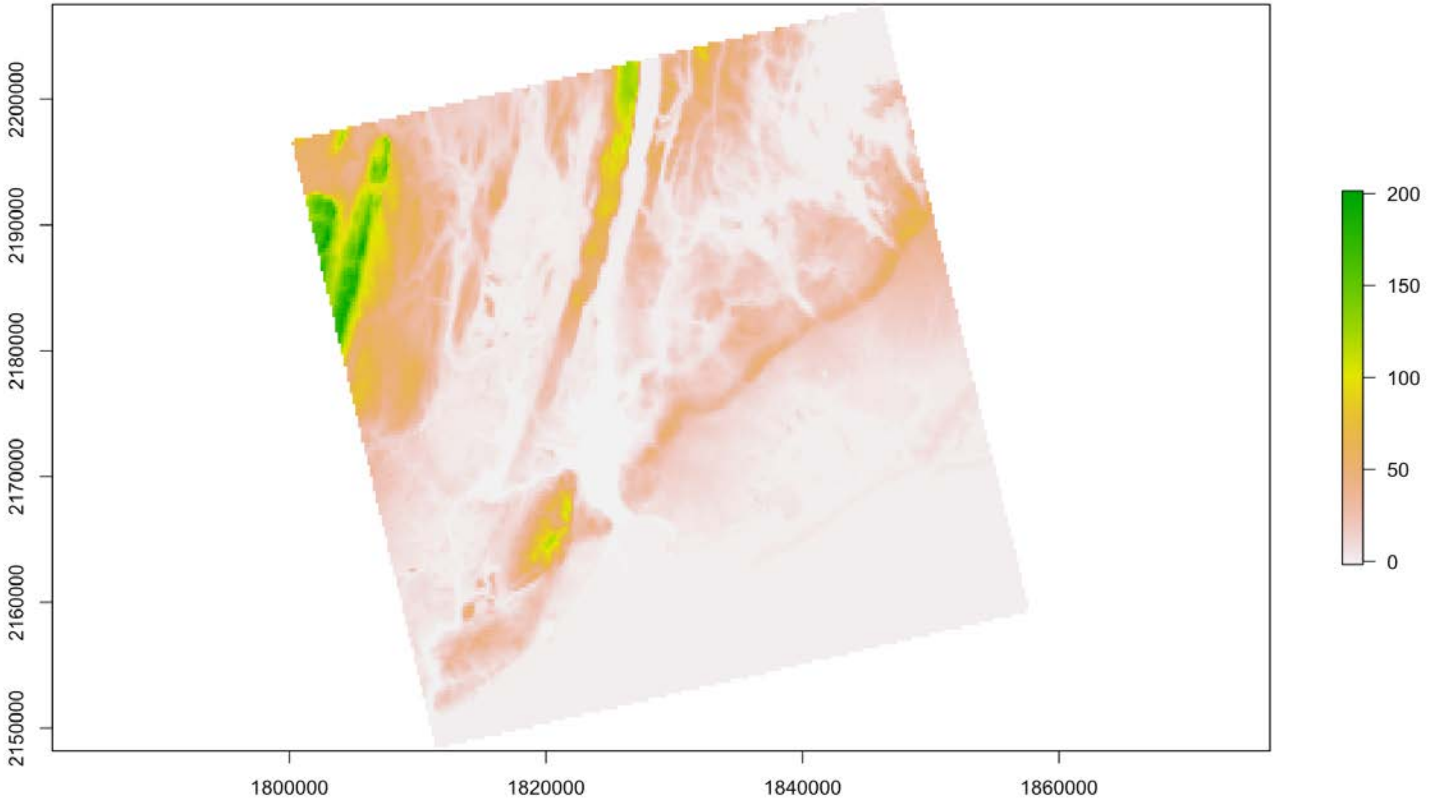
```
## [1] "49,784"
```

```
res(lowres)
```

```
## [1] 227 303
```

Lower resolution elevation

```
plot(lowres)
```



Note there is also a `disaggregate()` function

```r
r <- raster(nrow = 2, ncol = 2, vals = rnorm(4))
ncell(r)
```

```
## [1] 4
```

```r
disaggregate(r, fact = 10) %>%
  ncell()
```

```
## [1] 400
```

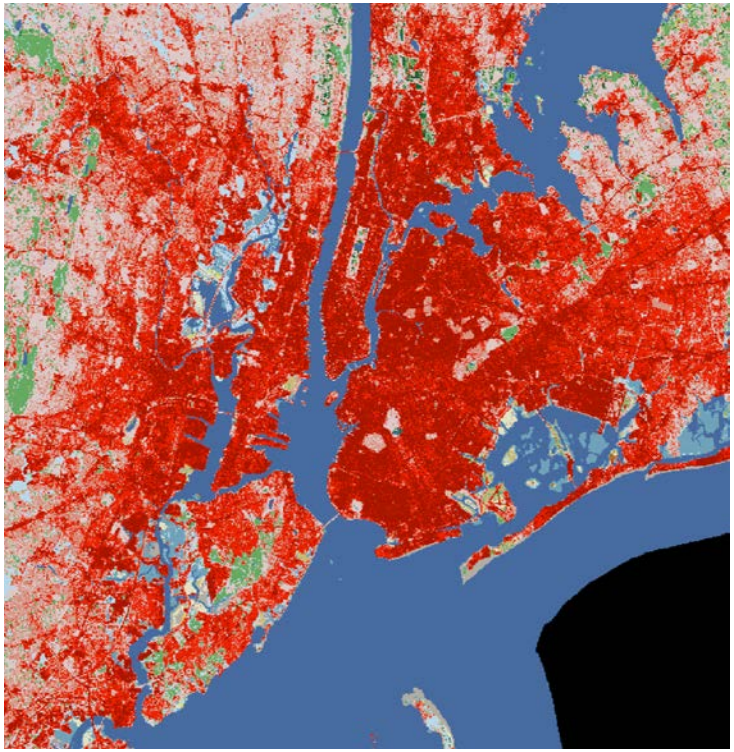
For our mini-analysis we have two raster-based variables to create

- Compute the amount of high-intensity, developed land within the buffers
- Compute the average tree canopy within the buffers

Start with computing the proportion of high intensity, developed land within the buffers

Here is the land use layer

```
plot(landuse)
```



Note that it is a categorical raster

The category definitions can be viewed using `levels()`

This is true because the original raster came with a metadata file (suffix `.tfw`)

Take a look at the levels limited to categories with at least one cell

We're only interested in cells with a value of 24

```
levels(landuse)[[1]] %>%
  filter(Count != 0) %>%
  select(Value, Count, NLCD.2011.Land.Cover.Class) %>%
  slice(1:10)
```

```
##    Value      Count  NLCD.2011.Land.Cover.Class
## 1      0 7854240512                Unclassified
## 2     11  469012527                  Open Water
## 3     12    1599206         Perennial Snow/Ice
## 4     21  292251633       Developed, Open Space
## 5     22  131633826    Developed, Low Intensity
## 6     23   59456652 Developed, Medium Intensity
## 7     24   21426522   Developed, High Intensity
## 8     31  110507264                 Barren Land
## 9     41  973617734            Deciduous Forest
## 10    42 1037912310            Evergreen Forest
```

In each buffer we will want to count the number of grid cells with a value/code of 24

Perhaps easiest to create a layer with 1 for developed and 0 otherwise

There are a couple of options

- `layerize()`
- Our friend from before, `calc()`

`layerize()` is a magical function

Create a binary layer for each category with `layerize()`

Creates a `RasterBrick`

Apply `layerize()`

```
landuse_layers <- layerize(landuse)
```

The result is a raster brick with 16 layers

```
class(landuse_layers)
```

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

```
nlayers(landuse_layers)
```

```
## [1] 16
```

The names of the layers start with an "X" followed by the value

```
names(landuse_layers)
```

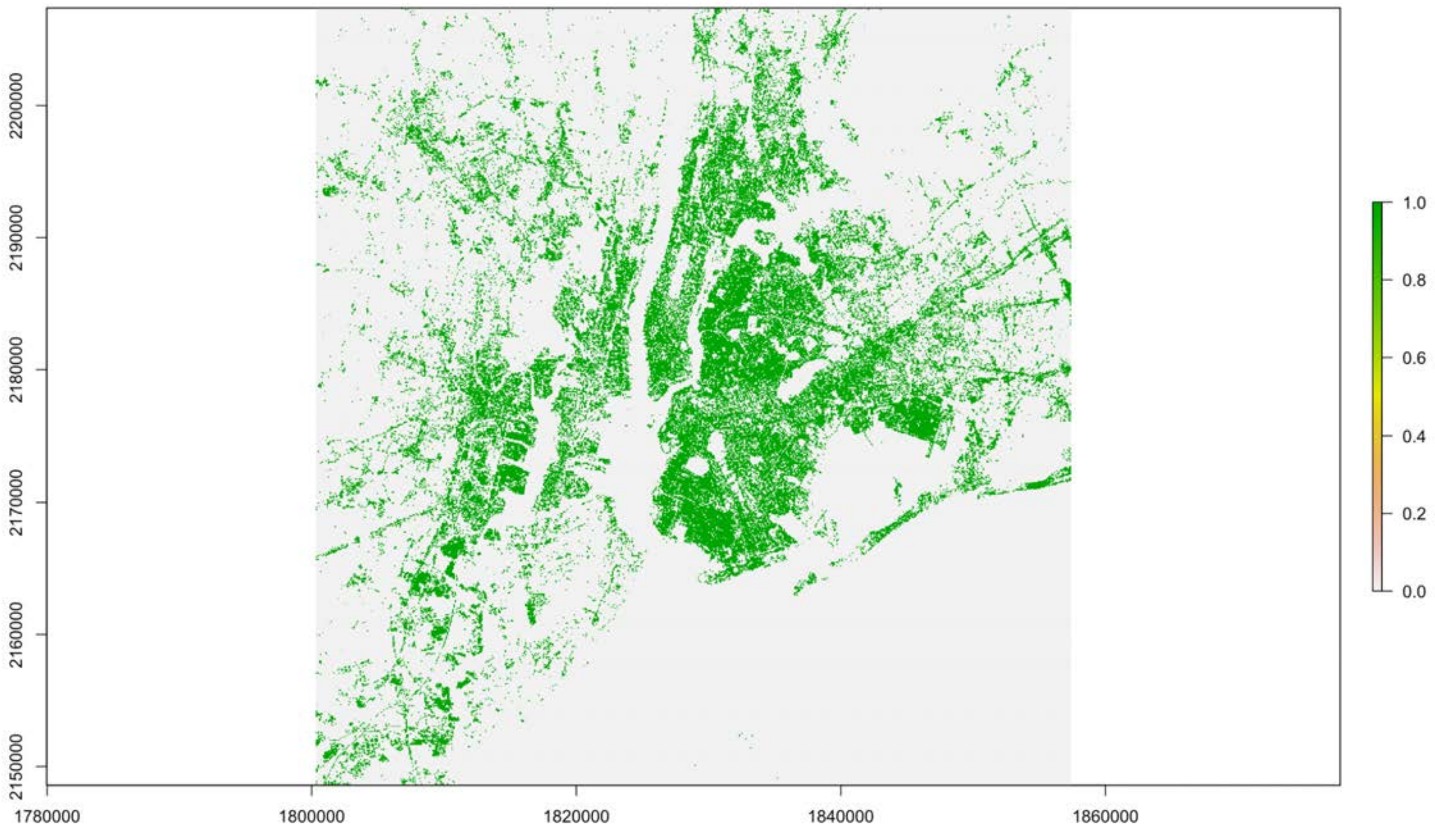
```
##  [1] "X0"  "X11" "X21" "X22" "X23" "X24" "X31" "X41" "X42" "X43"
## [12] "X71" "X81" "X82" "X90" "X95"
```

We can pull out the layer of interest with `subset()`

```
developed <- subset(landuse_layers, "X24")
```

A plot of high-intensity, developed grid cells

```
plot(developed)
```



`layerize()` works great but is more computation than needed

There is a simpler way to assign values of 24 to 1 and others to 0

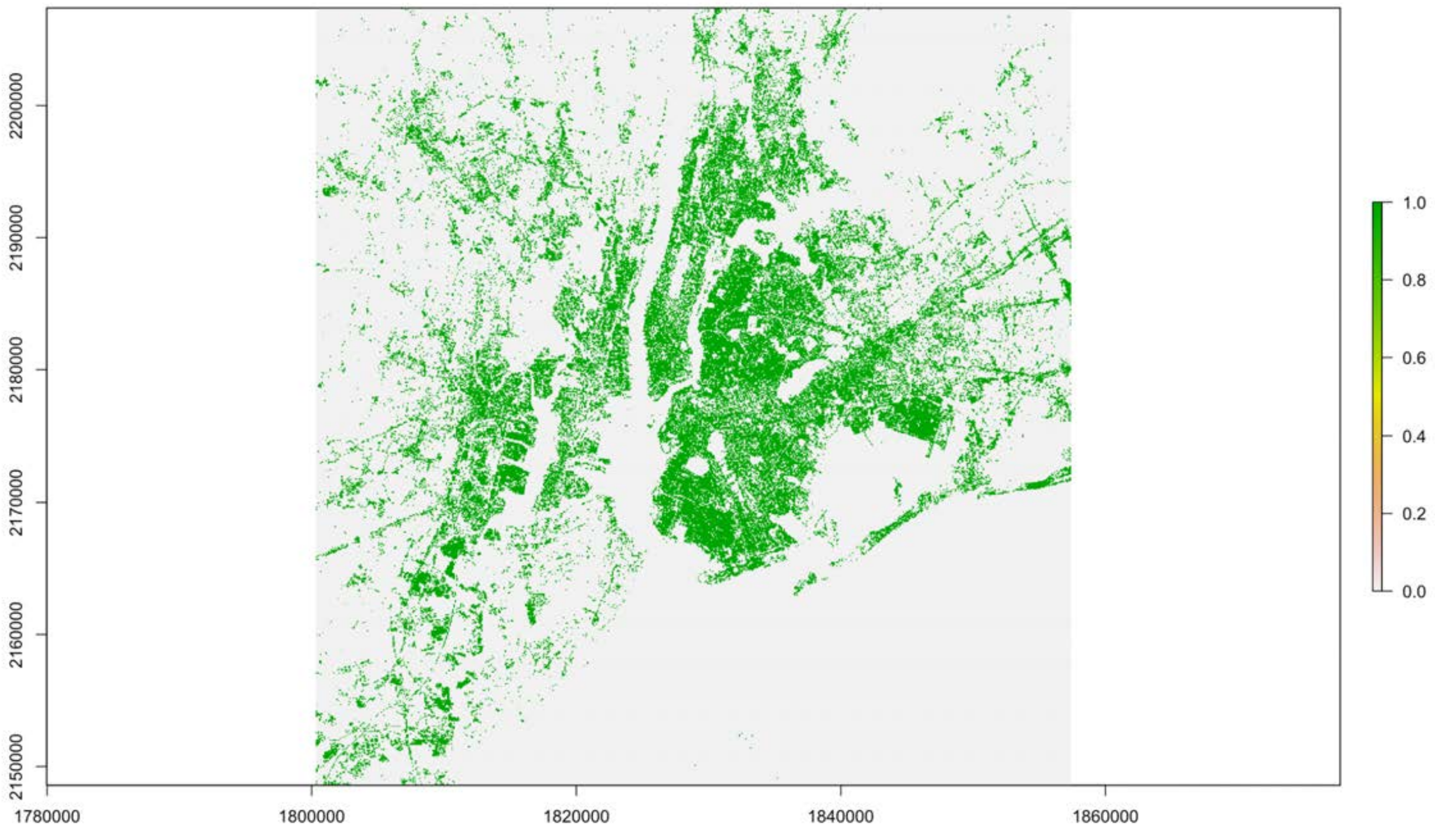
Use `calc()`

```
# Our function
f <- function(x){
  x[x != 24] <- 0
  x[x == 24] <- 1
  x
}
```

```
developed <- calc(landuse, f)
```

A plot of high-intensity, developed grid cells

```
plot(developed)
```



We have the raster layer we need...

Now we need to sum the cells by buffer

If you have zones as vectors you can use `extract()`

`extract()` pulls values from the raster at points or within polygons

And `extract()` can use {sf} objects!

Though the documentation does not mention this 🤔

`extract()` is particularly easy to apply if you only need the cell value under each point

```
extract(developed, monitors)
```

```
##  [1] 0 0 0 1 0 0 0 1 1 0 1 1 1 1 1 1 1 0 1 0 0 1 1 0 0 1 1 0 1 1
## [36] 1 1 1 1 1 0 1 1 0 0 1 1 0 1 1 1 1 0 1 1 1 1 1 1 0 0 0 0 0
```

But we want the total developed land in the buffers (polygons)

With polygons `extract()` returns all values in a list by default

```
raw_vals <- extract(developed, monitor_buffers)
```

```
raw_vals[[1]][1:5]
```

```
## [1] 0 0 0 0 1
```

```
raw_vals[[20]][1:5]
```

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

You could sum the values yourself or...

You can provide a summary function to `extract()`

```
developed_count  <- extract(
  developed,
  monitor_buffers,
  fun = sum
)
```

And here is our final result

```
head(developed_count)
```

```
##      [,1]
## [1,]   31
## [2,]   17
## [3,]   47
## [4,]  422
## [5,]  386
## [6,]   84
```

Add the result to the original buffer file

```
monitor_developed <- monitor_buffers %>%
  mutate(developed_count = c(developed_count)) %>%
  select(site_id, developed_count) %>%
  st_drop_geometry()
```

Final computation in our mini-analysis!

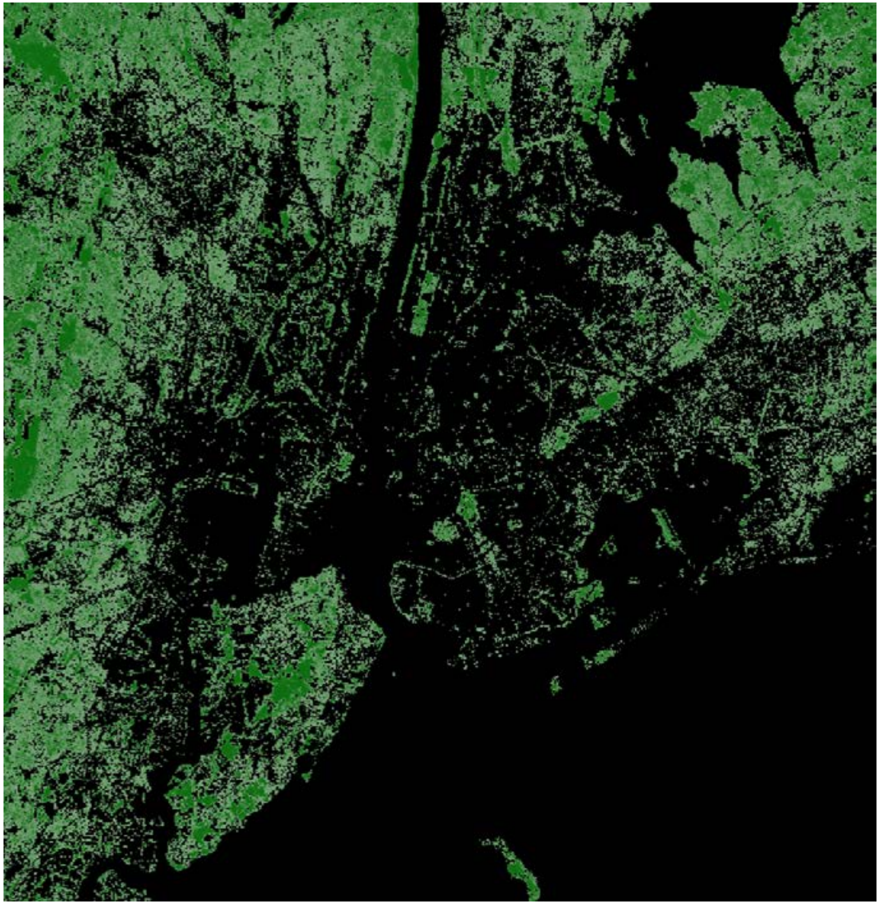
Average tree canopy in the buffer

Tree canopy is a numeric raster

```
canopy <- raster("canopy.tif")
```

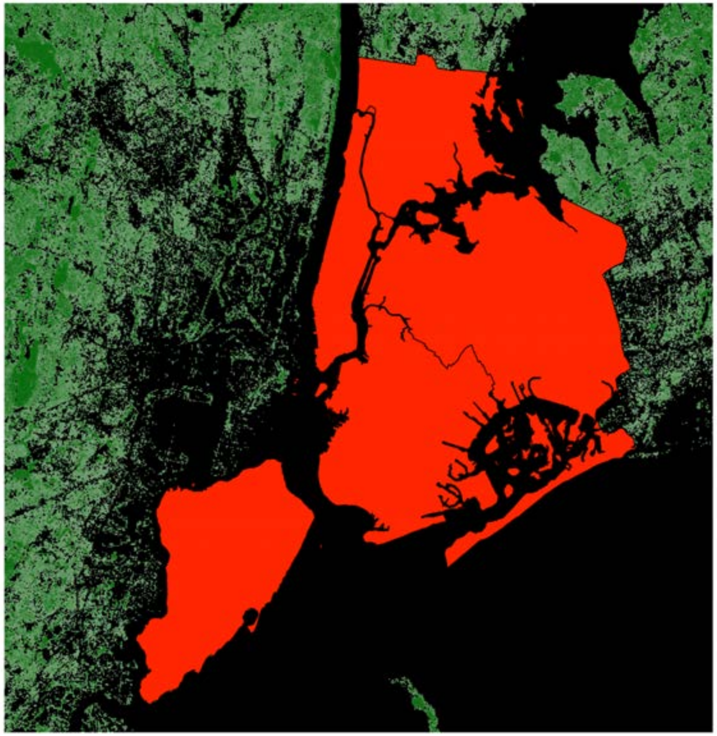
Values are percent of tree canopy

```
plot(canopy)
```



Before using `extract()` to grab the values

The raster extent is bigger than we need



Let's crop and mask so we keep only raster values in the counties

`crop()` will clip the raster to the (square) extent of another layer

`crop()` the canopy layer to the counties

```
cropped <- crop(canopy, counties)
```

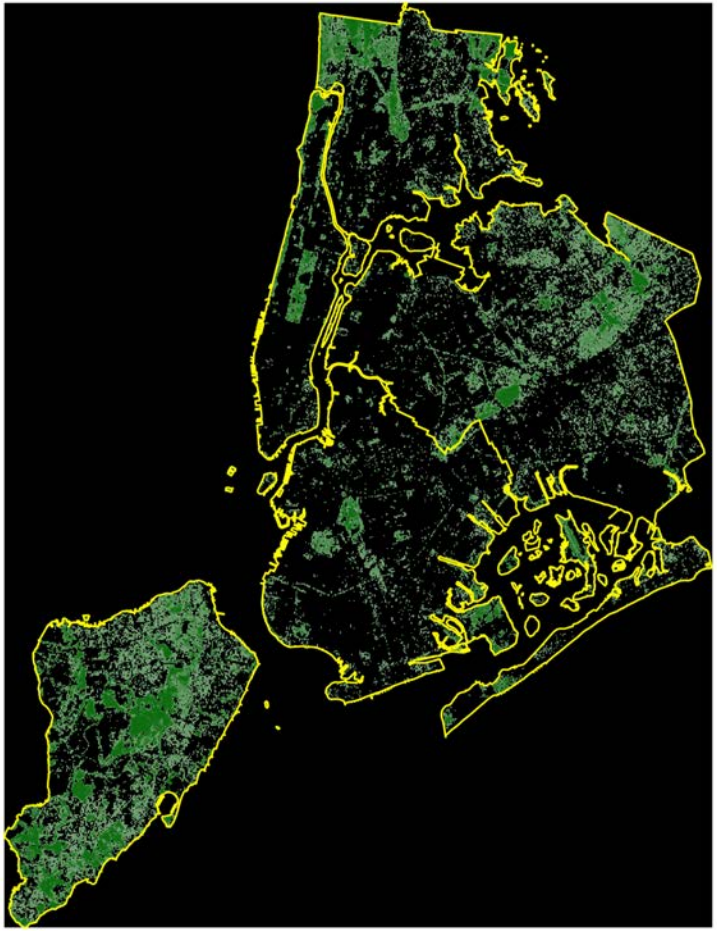
Plot the cropped raster

`mask()` will assign NA to cells outside the polygon layer

`mask()` the canopy layer with the counties

```
masked <- mask(cropped, counties)
```

Plot the masked (and cropped) layer



Extract the values for the buffers using a `mean()` function

```
canopy_vals <- extract(masked, monitor_buffers,
                       fun = mean, na.rm=TRUE)
```

Create the canopy table

```
monitors_canopy <- monitor_buffers %>%
  mutate(canopy_avg = c(canopy_vals)) %>%
  select(site_id, canopy_avg) %>%
  st_drop_geometry()
```

Results of our mini-analysis

We have five result files

- We calculated road density by intersecting the buffers with the roads with `st_intersection()` and `st_length()
- We calculated minimum distance to the nearest road with `st_nearest_feature()` and `st_distance()`
- We calculated population in the buffer by using `st_intersection()` (poly to poly) and `st_area()`
- We used `calc()` and `extract()` to calculated developed land in the buffer from a raster
- We used `extract()` (with some `crop()` and `mask()`) to compute canopy in the raster

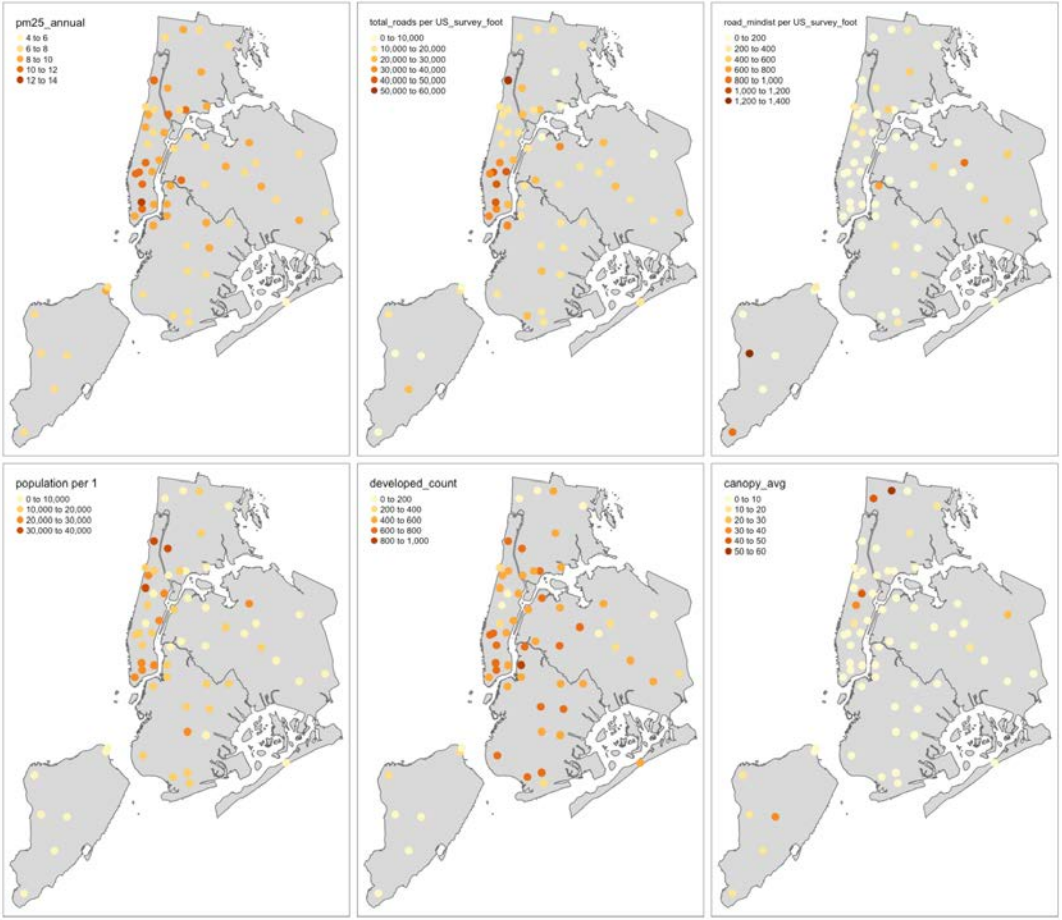
We can assemble the pieces together

```
monitor_results <- monitors %>%
  inner_join(monitor_roads, by = "site_id") %>%
  inner_join(monitor_road_mindist, by = "site_id") %>%
  inner_join(monitor_population, by = "site_id") %>%
  inner_join(monitor_developed, by = "site_id") %>%
  inner_join(monitors_canopy, by = "site_id")
```

Map all our variables in one window

```
tm_shape(counties) + tm_polygons()+
tm_shape(monitor_results) +
  tm_dots(c("pm25_annual", "total_roads",
            "road_mindist", "population",
            "developed_count", "canopy_avg"), size = 0.5)
```

Map all our variables in one window



Which variables are most strongly correlated with air pollution?

Look at correlation using the {base} function `cor()`

Tiny bit of prep

```
# Prepare the data
results <- monitor_results %>%
  select(pm25_annual, total_roads, road_mindist,
         population, developed_count, canopy_avg) %>%
  st_drop_geometry()
```

Look at correlation using the {base} function `cor()`

```
cor(results) %>% round(2)
```

	pm25_annual	total_roads	road_mindist	population	developed_count	canopy_avg
pm25_annual	1.00	0.70	-0.36	0.34	0.49	-0.29
total_roads	0.70	1.00	-0.45	0.43	0.34	-0.24
road_mindist	-0.36	-0.45	1.00	-0.22	-0.19	0.03
population	0.34	0.43	-0.22	1.00	0.52	-0.38
developed_count	0.49	0.34	-0.19	0.52	1.00	-0.74
canopy_avg	-0.29	-0.24	0.03	-0.38	-0.74	1.00

Before creating a few scatter plots look at the data once more

```
glimpse(monitor_results)
```

```
## Observations: 64
## Variables: 9
## $ site_id         <dbl> 228, 952, 2269, 2496, 2596, …
## $ reference       <dbl> 0, 0, 1, 0, 0, 0, 0, 0, 0, 0…
## $ pm25_annual     <dbl> 6.473097, 6.591441, 6.107921…
## $ geom            <POINT [US_survey_foot]> POINT (91…
## $ total_roads     [US_survey_foot] 2772.254 [US_surv…
## $ road_mindist    [US_survey_foot] 838.19670 [US_sur…
## $ population      [1] 2276.8915 [1], 3464.4637 [1], …
## $ developed_count <dbl> 31, 17, 47, 422, 386, 84, 77…
## $ canopy_avg      <dbl> 16.88302752, 18.76931949, 34…
```

Since correlation is strongest with `total_roads` let's plot

```r
library(ggplot2)
# Error, not happy with units
ggplot(monitor_results, aes(total_roads, pm25_annual)) +
  geom_point() + geom_smooth(method = "lm")
```

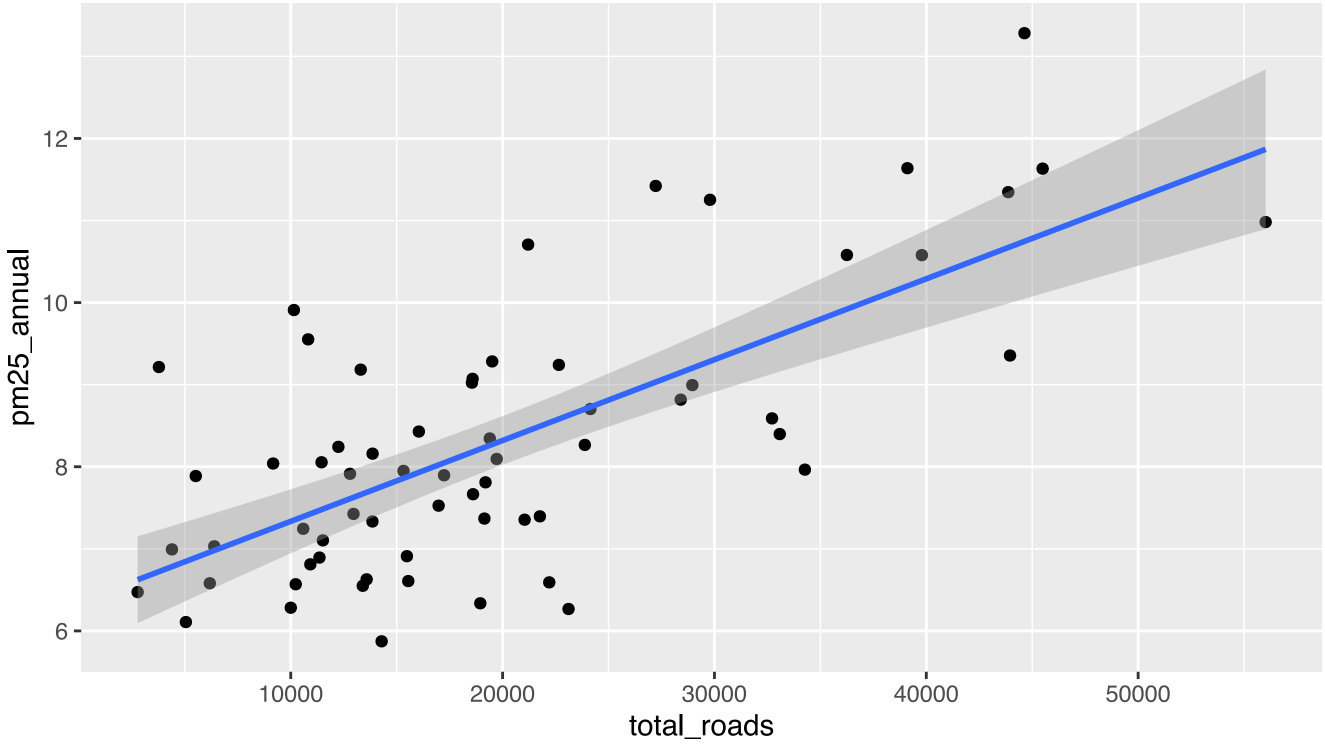
```
## Error in Ops.units(x, range[1]): both operands of the expression
```

Shucks, need to remove units, do you remember how to do this?

```
monitor_results <- monitor_results %>%
  mutate(total_roads = units::drop_units(total_roads))
```

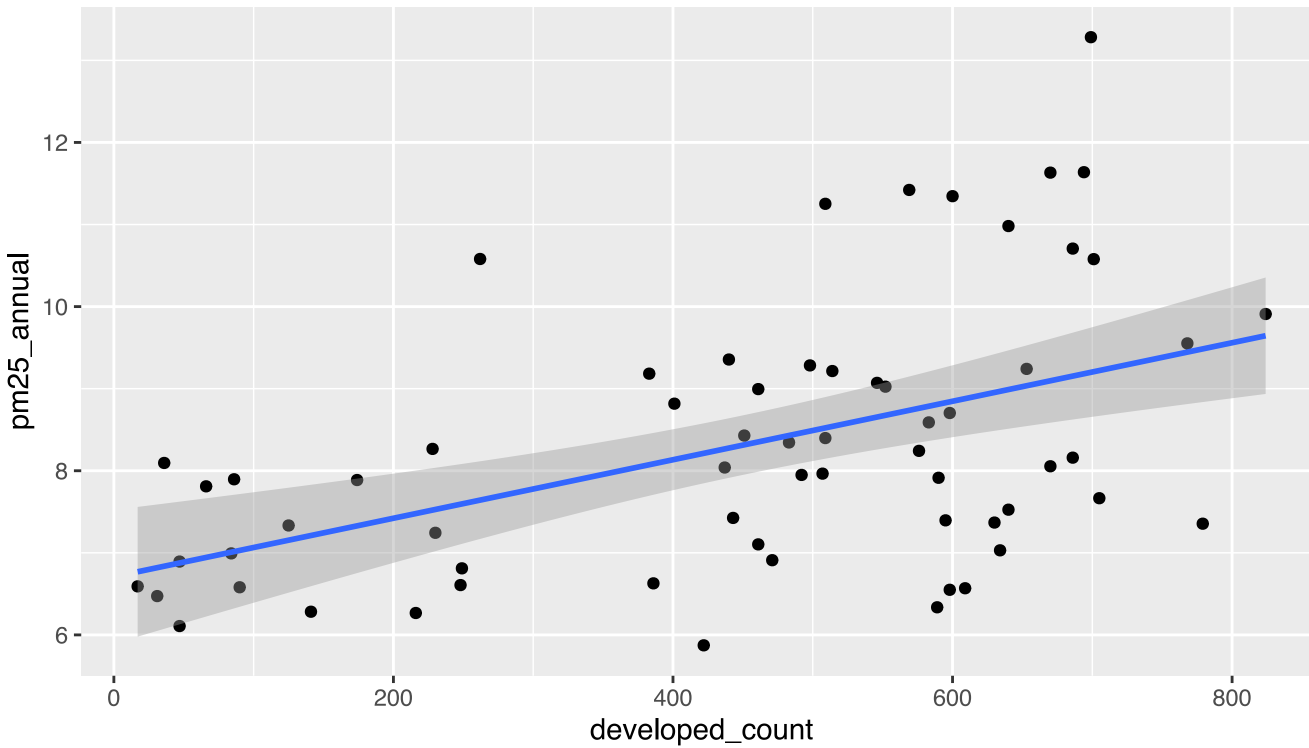
Our scatter plot of roads within 500 meters against air pollution

```r
ggplot(monitor_results, aes(total_roads, pm25_annual)) +
  geom_point() + geom_smooth(method = "lm")
```



Amount of high-intensity developed land within 500 meters against air pollution

```
ggplot(monitor_results, aes(developed_count, pm25_annual)) +
  geom_point() + geom_smooth(method = "lm")
```



Summary of air quality results

- Air quality strongly related to road density and developed land use
- Air quality negatively related to minimum distance to the nearest road and tree canopy
- Air quality modestly related to total population in the buffer

Please provide feedback before finishing the exercise

http://bit.ly/zrsaSpatialWorkshopFeedback open_exercise(7) and finish