geogRaphy: an introduction to spatial data in R

Sarah C. Lotspeich

Vanderbilt University

17 Nov 2020

Introduction

spatial data:

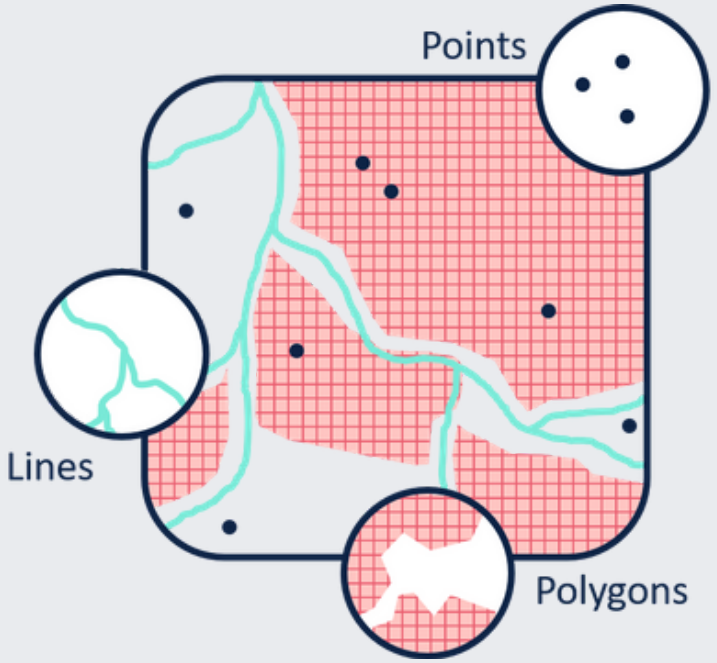
noun • [spay-shull day-ta] • data which is representative of a specific, geographic location on the surface of the Earth.

[1] What Is Spatial Data?

Two main types:

1. Vector data

- Graphical representations of the real world.
- Usually, points, lines, and polygons.
- GIS shapefiles (`.shp`) are typically vector data.



[1] Geospatial Data Models

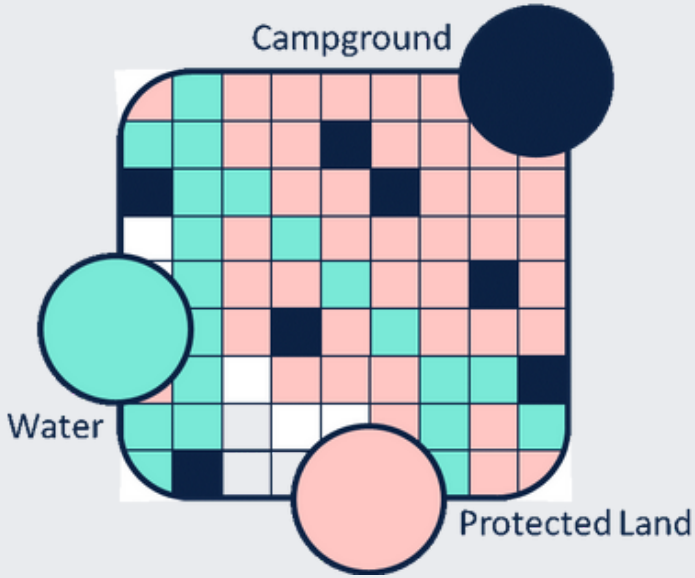
[2] What Is Spatial Data?

[3] Vector and Raster: A Tale of Two Spatial Data Types

Two main types:

2. Raster data

- Continuous surface divided into a grid of cells (pixels).
- Each pixel contains measured value for the area it represents.
- One cell might represent a 10m x 10m area.



[1] Geospatial Data Models

[2] What Is Spatial Data?

[3] Vector and Raster: A Tale of Two Spatial Data Types

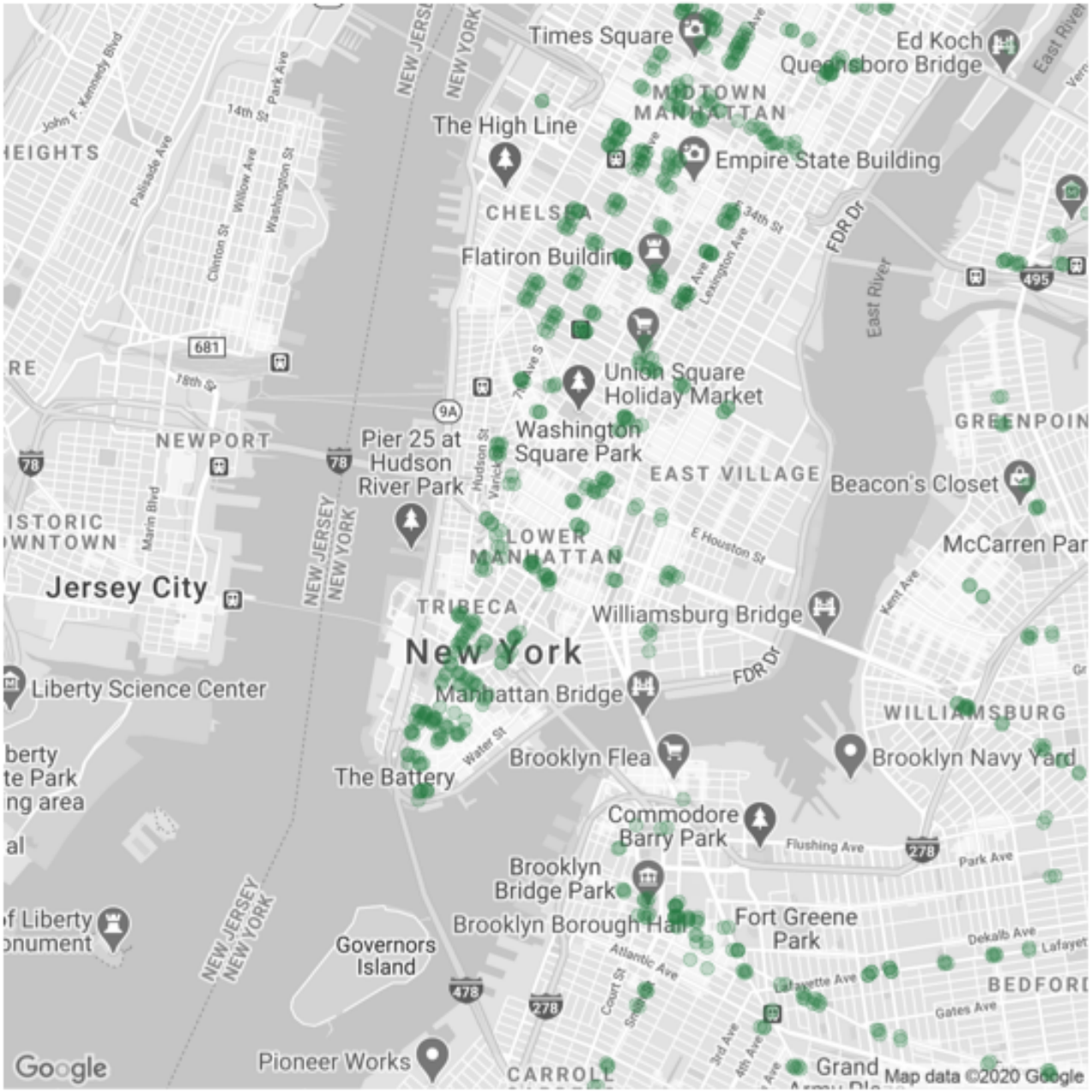
More than the "where"

- Spatial data contain more than just location.
- Additional attributes can be tied to the observation based on its location.
- Non-spatial features are called **attributes**.
- Example: New York City subway entrances.

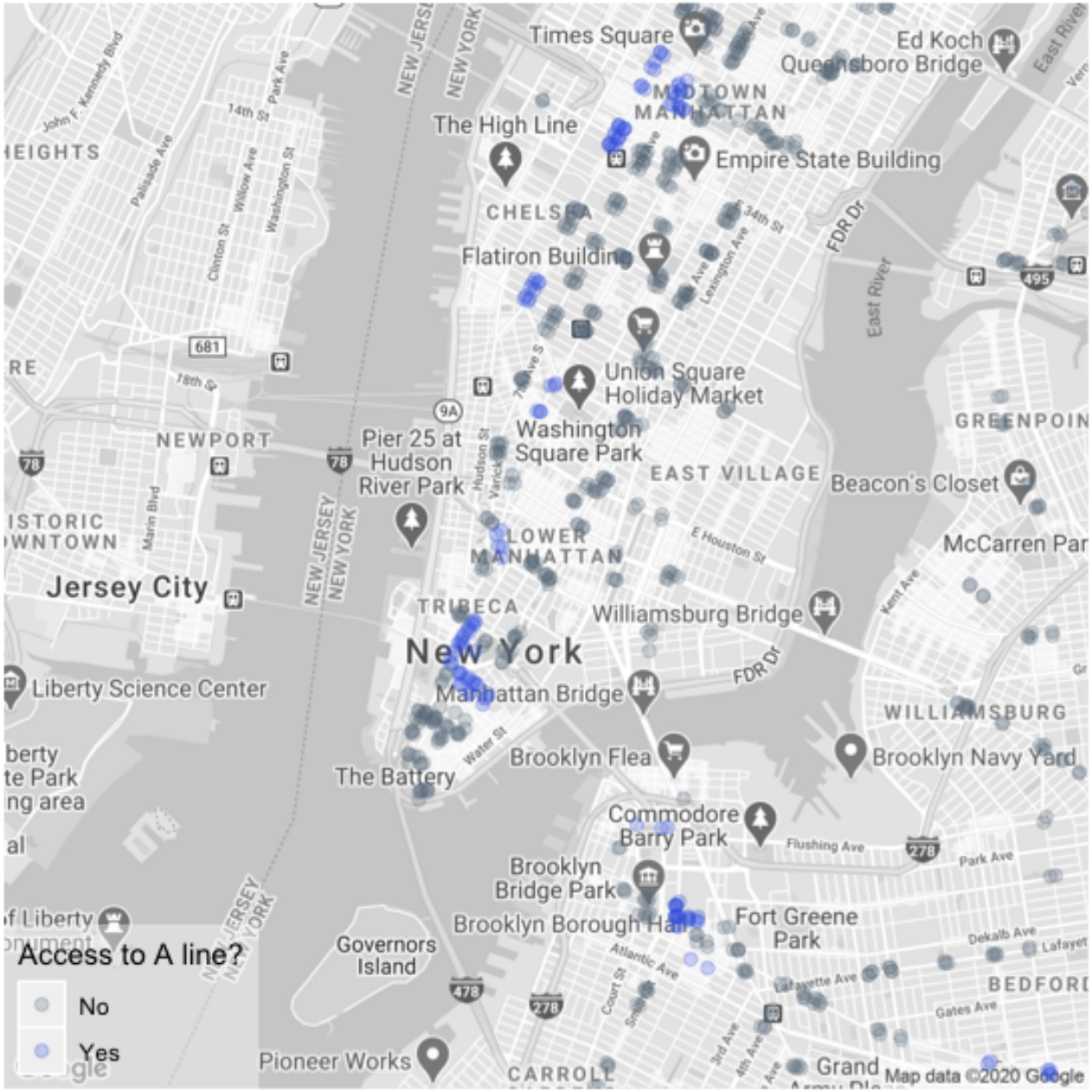
[1] What Is Spatial Data?

[2] Data source: City of New York Subway Entrances

New York City subway entrances



New York City subway entrances



New York City subway entrances

Outline

Today's game plan:

1. Geocoding

2. Distance calculations

3. Map-making

Install/load packages

To follow along with my code as written, you will need to install and load the following `R` packages:

- `magrittr*`
- `dplyr*`
- `ggplot2*`
- `ggmap`
- `geosphere`

* You'll need these from the start. The others will be formally introduced later.

Materials available on GitHub

Access the slides and supplementary materials on GitHub at `https://github.com/sarahlotspeich/geogRaphy`.

- Slides
- Datasets
- Maps
- Additional code

I recommend pausing the video to open the `geogRaphy.html` lecture file for clickable links.

Geocoding

Geocoding is the process of converting addresses (like a street address) into geographic coordinates (like latitude and longitude), which you can use to place markers on a map, or position the map.

[1] Google Maps Geocoding API

Best practices

1. Remove suite/apartment numbers (they will simply create ties).
2. Be mindful of special characters like @, #, ?, etc.
3. Check for misspellings or abbreviations.
4. Prepare to spend some time cleaning addresses.

```
##                                 original
## 1 310 25TH AVE S #103 NASHVILLE TN 37240
## 2 2301 VANDERBILT PLC NASHVILLE TN 37240
## 3   2400 BLAKMORE AVE NASHVILLE TN 37212
## 4    TWENTY 3RD AVE N NASHVILLE TN 37212
##                                 cleaned
## 1     310 25TH AVE S NASHVILLE TN 37240
## 2 2301 VANDERBILT PL NASHVILLE TN 37240
## 3 2400 BLAKEMORE AVE NASHVILLE TN 37212
## 4         23RD AVE N NASHVILLE TN 37212
```

[1] Blossom (2014), Geocoding Best Practices

How it works

The algorithm begins by taking a complete street address and breaking it down into its component parts.

Example: 2525 WEST END AVE NASHVILLE TN 37203

- **Street number:** 2525
- **Street name:** WEST END
- **Street type:** AVE
- **Street suffix direction:**
- **City:** NASHVILLE
- **State:** TN
- **ZIP:** 37203

This address is then compared to a reference table that has already been mapped (e.g., GoogleMaps or TomTom).

[1] Blossom (2014), Geocoding Best Practices

Level of accuracy

Often (but not always) geocoders can pinpoint exact property or building of the address input. When it cannot, a few things can happen:

- a match at a less precise level

  If 170 MAIN STREET, CLEVELAND, OH cannot be found in Cleveland, the address may be matched either with a 170 MAIN STREET in a nearby town or with the city center of CLEVELAND

- address range interpolation along a street

  170 MAIN STREET will be placed 70% of the way along the block of MAIN STREET that ranges from street numbers 100-200

Most geocoders give a level of confidence or matching. You can also confirm matches visually!

[1] Blossom (2014), Geocoding Best Practices

Example: Farmers Markets Directory and Geographic Data

State, address, name, and zip code of farmers markets in the United States, along with attributes such as payment methods accepted and types of goods for sale.

```
##      fmid                street       city    county
## 1 1018261                            Danville Caledonia
## 2 1000709 71 Waterwitch Avenue  Highlands  Monmouth
## 3 1019846      110 W. Main St. West Union     Adams
##        state   zip
## 1    Vermont  5828
## 2 New Jersey  7732
## 3       Ohio 45693
```

Downloaded from the US Department of Agriculture. Read it into R:

```
p <- "https://raw.githubusercontent.com/sarahlotspeich/geogRaphy/main/da
farm <- read.csv(p, stringsAsFactors = F)
```

The ggmap package

A collection of functions to visualize spatial data and models on top of static maps from various online sources (e.g Google Maps and Stamen Maps). It includes tools common to those tasks, including functions for geolocation and routing.

The `geocode()` function takes inputs:

- `location`: a character vector of street addresses or place names
- `output`: amount of output
  - `output = "latlon"` returns latitude/longitude
  - `output = "latlona"` returns latitude/longitude + address
  - `output = "more"` returns same as `"latlon"` + accuracy measures `type` and `loctype`
  - `output = "all"` returns same as `"more"` + many others
- `source`: source of reference table to use (set `source = "google"`)

[1] D. Kahle and H. Wickham. ggmap: Spatial Visualization with ggplot2. The R Journal, 5(1), 144-161.

Setting up your API key

Get a key

First, you'll need to register for a *free* Google Maps API key here: https://cloud.google.com/maps-platform/.

Register your key in R

```
ggmap::register_google(key = "YOUR KEY")
```

You'll need to call the `register_google()` command everytime you open a new session. If you want to set it permanently, instead call:

```
ggmap::register_google(key = "YOUR KEY", write = TRUE)
```

Geocoding the US farmers market directory

```
farm %>%
  mutate(address = paste(street, city, state, zip, sep = ", ")) %>%
  pull(address) %>%
  ggmap::geocode(output = "more", source = "google") -> farm_geo
```

```
## # A tibble: 6 x 9
##       lon   lat type  loctype address north south   east
##     <dbl> <dbl> <chr> <chr>   <chr>   <dbl> <dbl>  <dbl>
## 1  -89.5   34.4 stre… range_… 700 n …  34.4  34.4  -89.5
## 2 -105.    40.6 prem… rooftop 200 w …  40.6  40.6 -105.
## 3  -87.8   41.8 prem… rooftop 10 pin…  41.8  41.8  -87.8
## 4 -105.    38.9 prem… rooftop 7350 p…  38.9  38.9 -105.
## 5  -88.0   30.7 stre… rooftop 300 co…  30.7  30.7  -88.0
## 6  -90.3   42.4 stre… rooftop 2000 t…  42.4  42.4  -90.3
## # … with 1 more variable: west <dbl>
```

Checking accuracy of `ggmap::geocode()`

The `type` and `loctype` outputs tell us about the geocoded match. Let's look at a few examples...

Less precise matches:

- There were a couple of addresses that only gave city, state, and zip. These yielded `approximate` matches.
- Others gave street, city, state, and zip, which were matched with `geometric_center`.

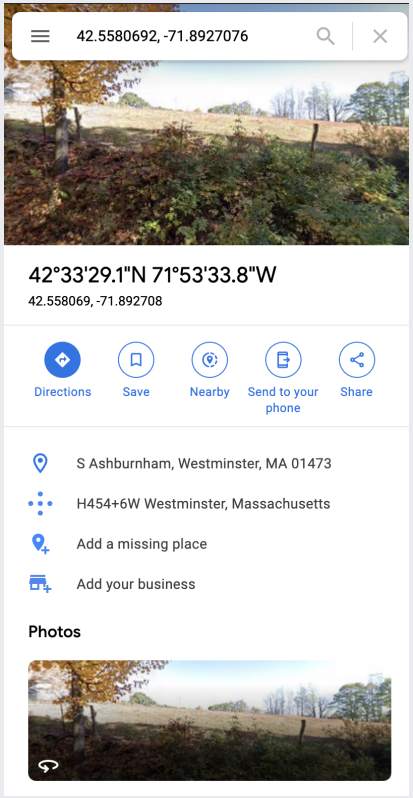
More precise matches:

- Ideally, addresses would be either `rooftop` level matched or `range_interpolated`.

Manual check

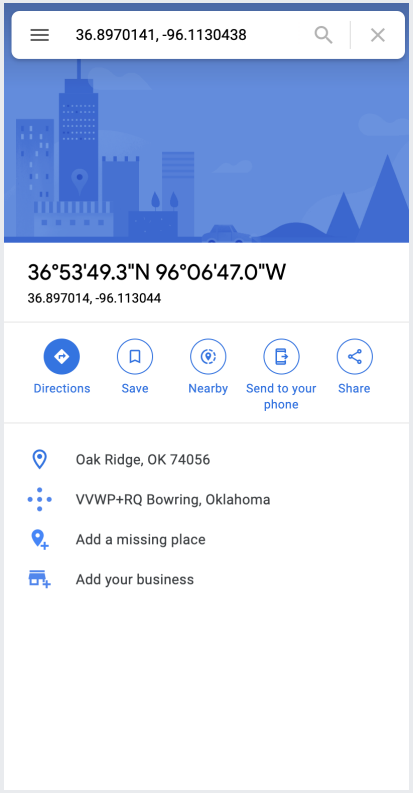
Address: Westminster, MA, 01473

Geocoded: (42.5580692, -71.8927076)



Address: Oak Ridge, OK, 74056

Geocoded: (36.8970141, -96.1130438)



Alternative function `ggmap::mutate_geocode()`

If you want to add latitude/longitude columns to your existing `farm` dataset, you can use `ggmap::mutate_geocode()`. It takes the address column as an input (`location = address`).

```
farm %>%
  mutate(address = paste(street, city, state, zip, sep = ", ")) %>%
  ggmap::mutate_geocode(location = address, source = "google") -> farm
```

Now we have a dataset of 1368 farmers markets in the US with their geocoded locations as `(lat, lon)` coordinates.

Distance calculations

Great-circle distance calculations

"The **great-circle distance**... is the shortest distance between two points on the surface of a sphere..." (Wikipedia)

Given two `(lat, lon)` coordinates, there are two common options here:

1. Spherical Law of Cosines
2. Haversine Formula

Both calculations have been implemented in the `R` package `geosphere`.

[1] Great-circle distance calculations in R

The `geosphere` package

Spherical trigonometry for geographic applications. That is, compute distances and related measures for angular (longitude/latitude) locations.

Functions `distHaversine()` and `distCosine()` take the same inputs:

- `p1` and `p2`: longitude/latitude of point(s)
- `r`: radius of the earth (set `= 3958.8` for output in miles and `= 6378137` for output in meters)

and give individual distance calculations.

[1] R. Hijmans. Introduction to the "geosphere" package

The `geosphere` package

Spherical trigonometry for geographic applications. That is, compute distances and related measures for angular (longitude/latitude) locations.

The `distm()` function takes two sets of points (rather than two points) and returns a matrix of the distances between the sets.

- `x` and `y`: longitude/latitude of point(s). If `y` is null, `distm(x)` returns the pairwise distances between each point in `x`.
- `fun`: which function to use to compute distances (set `fun = distHaversine` or `fun = distCosine`)

Note: Since `distm()` does not have a `r` input, the output will always be in meters by default.

[1] R. Hijmans. Introduction to the "geosphere" package

The `geosphere` package

Spherical trigonometry for geographic applications. That is, compute distances and related measures for angular (longitude/latitude) locations.

The `distm()` function takes two sets of points (rather than two points) and returns a matrix of the distances between the sets.

- `x` and `y`: longitude/latitude of point(s). If `y` is null, `distm(x)` returns the pairwise distances between each point in `x`.
- `fun`: which function to use to compute distances (set `fun = distHaversine` or `fun = distCosine`)

Note: Since `distm()` does not have a `r` input, the output will always be in meters by default.

[1] R. Hijmans. Introduction to the "geosphere" package

Distance between two Knoxville farmers markets

```
farm %>% filter(fmid == 1019189) -> fmid1019189
farm %>% filter(fmid == 1019624) -> fmid1019624
```

Use Haversine formula and Spherical Law of Cosines to find the distance (in miles):

```
geosphere::distHaversine(p1 = fmid1019189[, c("lon", "lat")],
                         p2 = fmid1019624[, c("lon", "lat")],
                         r = 3958.8)
```

```
## [1] 1.300333
```

```
geosphere::distCosine(p1 = fmid1019189[, c("lon", "lat")],
                      p2 = fmid1019624[, c("lon", "lat")],
                      r = 3958.8)
```

```
## [1] 1.300333
```

Distance between *all* Knoxville farmers markets

There are 4 farmers markets registered in Knoxville, TN. Look at the full 4 by 4 distance matrix for them all.

```
farm %>% filter(city == "Knoxville", state == "Tennessee") -> knx_fm
geosphere::distm(x = knx_fm[, c("lon", "lat")],
                 fun = geosphere::distHaversine) * 0.000621371
```

```
##          [,1]      [,2]     [,3]      [,4]
## [1,] 0.000000 1.3017748 4.478106 1.2183703
## [2,] 1.301775 0.0000000 5.543254 0.8626713
## [3,] 4.478106 5.5432535 0.000000 4.8505622
## [4,] 1.218370 0.8626713 4.850562 0.0000000
```

The `* 0.000621371` transforms the matrix from meters to miles.

Defining "distance"

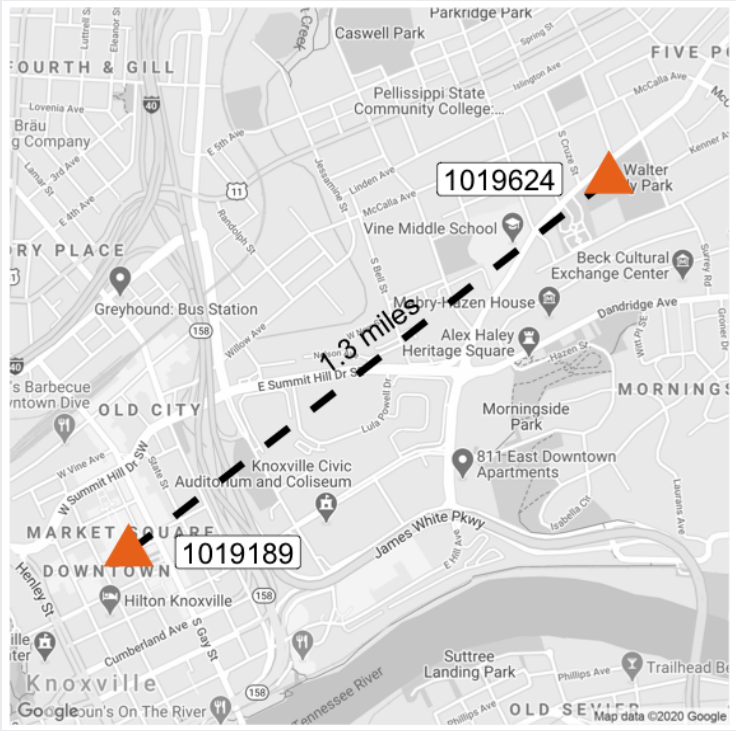
The **great-circle distance** measures assume the shortest possible route between two points, accounting for the approximate curvature of the earth.

This ignores things like:

- Roads
- Traffic
- Other real-world obstacles

Alternatives obtainable using `ggmap`:

1. Route-based travel distance
2. Estimated travel time



Calculating routes in ggmap

The `route()` function uses Google Maps API to calculate a path between two points. Key inputs include:

- `from`/`to`: addresses of the start/end point of the route
  - need to be a character inputs in the form `"latitude, longitude"` or addresses like `"street, city, state, zip"`
- `mode`: mode of transportation (options include `driving`, `bicycling`, `walking`, or `transit`)
- `structure`: structure of the output
  - if you want to map it, you'll want `structure = "route"`

[1] D. Kahle and H. Wickham. ggmap: Spatial Visualization with ggplot2. The R Journal, 5(1), 144-161.

Revisit two Knoxville farmers markets

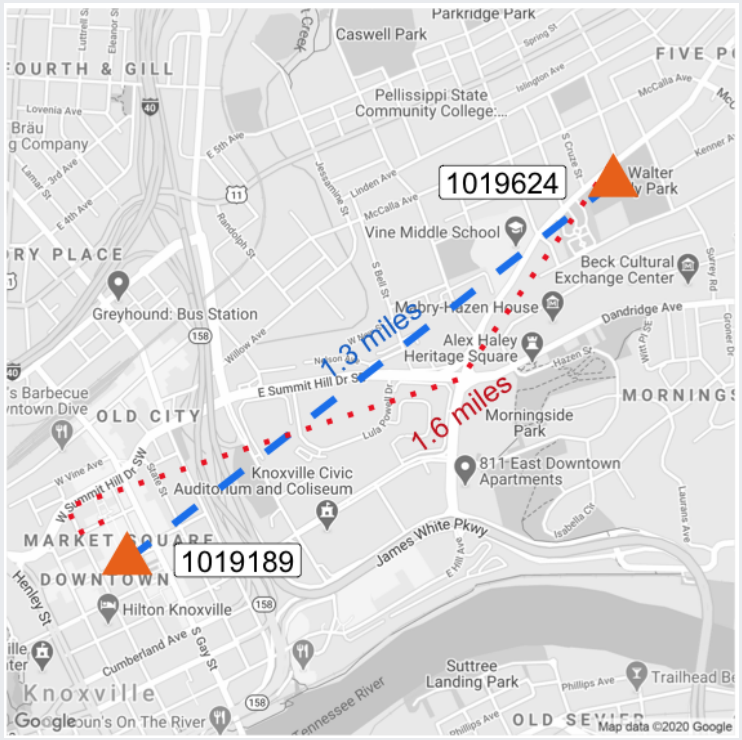
```
ggmap::route(from = fmid1019189$address, to = fmid1019624$address,
             mode = "driving", structure = "legs")
```

```
## # A tibble: 4 x 11
##       m    km  miles seconds minutes   hours start_lon
##   <int> <dbl>  <dbl>   <int>   <dbl>   <dbl>     <dbl>
## 1    96 0.096 0.0597      36    0.6  0.01        -83.9
## 2    98 0.098 0.0609      33    0.55 0.00917     -83.9
## 3  1476 1.48  0.917      223    3.72 0.0619      -83.9
## 4   906 0.906 0.563       67    1.12 0.0186      -83.9
## # … with 4 more variables: start_lat <dbl>, end_lon <dbl>,
## #   end_lat <dbl>, route <chr>
```

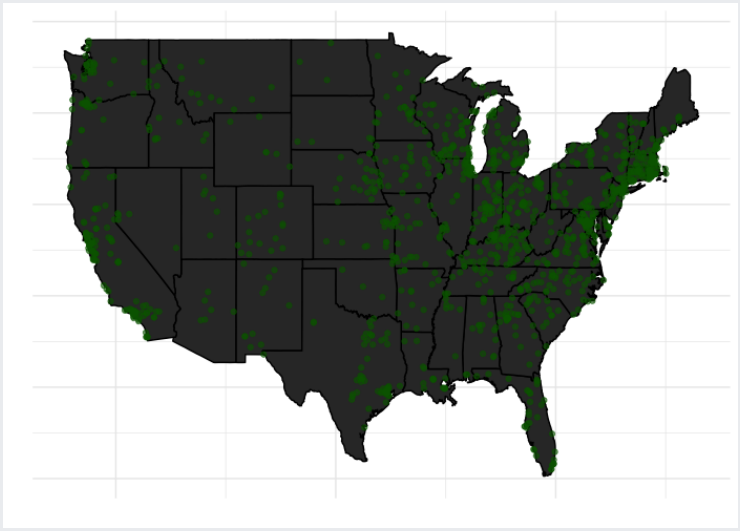
Revisit two Knoxville farmers markets

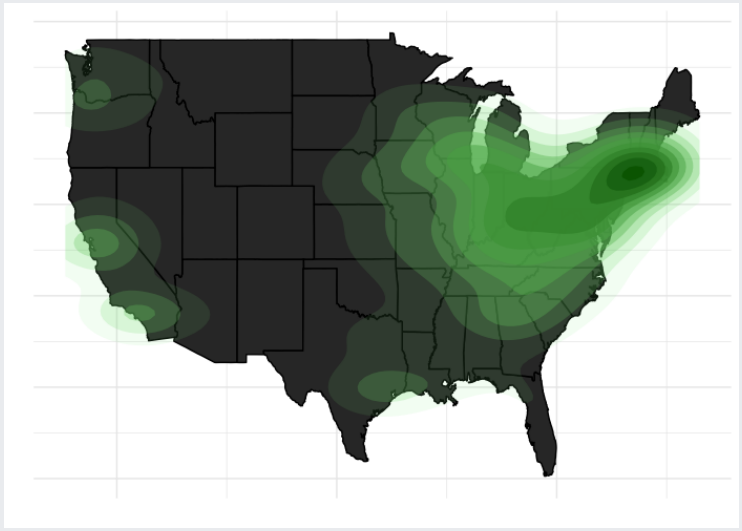
While the great-circle distance indicated that these farmers markets are **1.3 miles apart**, the true travel distance was found to be slightly farther at **1.6 miles** apart. This drive is estimated to take you **6 minutes**.

In fact, the great-circle distance was closer to the walking distance calculated using `ggmap`: **1.5 miles**.

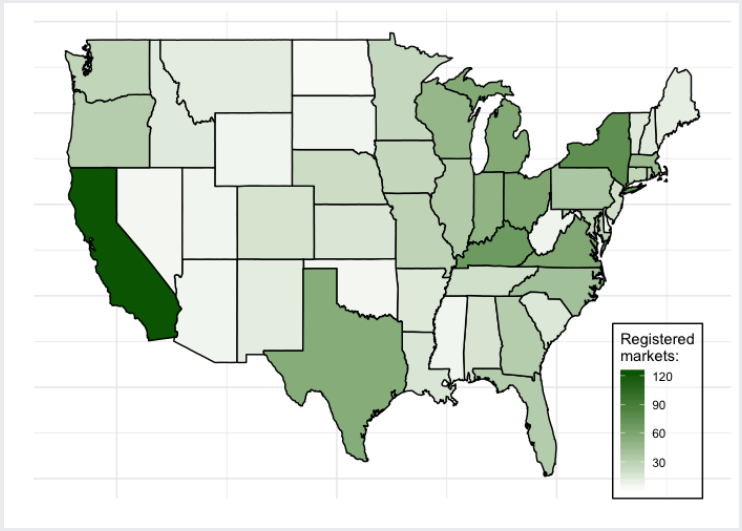


Map-making





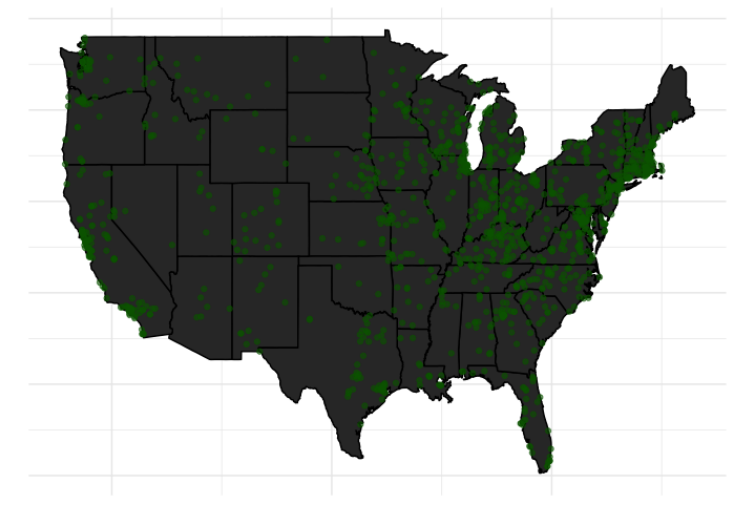
Popular map types



Mapping point data

Data consist of **points/coordinates** (lat, long) of distinct locations.

- Examples: home addresses
- Point can have attributes tied to them.
- Attributes can be incorporated using coloring or shapes.



In the US Farmers Market Registry, we can plot the locations of each individual market based on the geocoded latitude and longitude. This is pictured above.

Let's walk through how to build it...

Basic maps using `ggplot2`

First, we need a blank canvas: our blank map of the US. This can be in the form of a GIS shapefile, but most conveniently we can use the built-in `map_data()` function in `ggplot2`.

```
contig_us <- ggplot2::map_data("state")
```

```
##        long      lat group order  region subregion
## 1 -87.46201 30.38968     1     1 alabama      <NA>
## 2 -87.48493 30.37249     1     2 alabama      <NA>
## 3 -87.52503 30.37249     1     3 alabama      <NA>
```

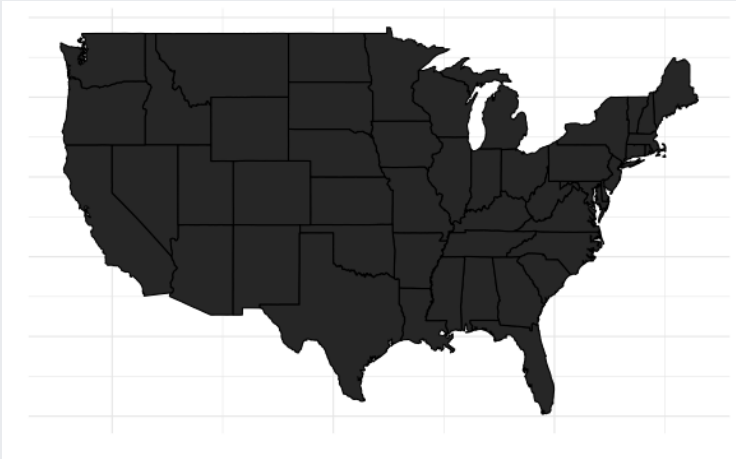
From here, we can plot the map using `geom_polygon()` in the `ggplot2` package.

We say (lat, long), but we *plot* (long, lat)

In other words, `x = long` and `y = lat`

Start with a map of the US

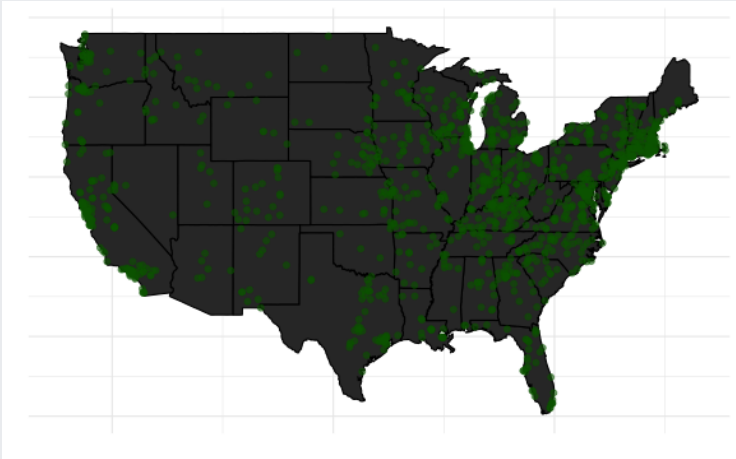
Each pair of coordinates (`lat`, `long`) denotes a point along the polygon's outline. The `order` variable tells it which order in which to connect the points. The `group` variable (in this case the state) specifies individual polygons.



```
us_map <- ggplot() +
          geom_polygon(data = contig_us,
                       aes(x = long, y = lat, group = group),
                       color = "black") +
          theme_minimal() +
          theme(axis.ticks = element_blank(),
                axis.text = element_blank())+
          xlab('') + ylab('')
```

Layering point data

We can overlay the basic map with the latitude and longitude of the markets using `geom_point()` in `ggplot2`. Recall that in the aesthetic, `x = lon` and `y = lat`.

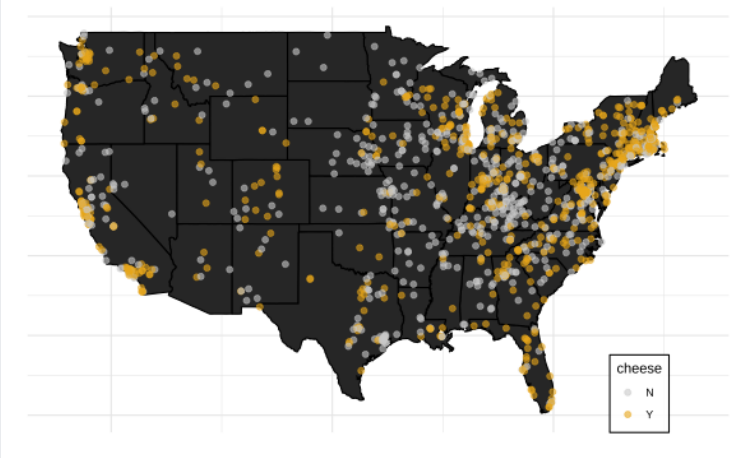


```
farm %>% filter(state != "Alaska", state != "Hawaii") -> flt_farm
us_map + geom_point(data = flt_farm,
                    aes(x = lon, y = lat),
                    color = "darkgreen",
                    size = 2, alpha = 0.6)
```

Adding point attributes

The `farm` data include additional indicators on the offerings at each market. We can include this in the map by coloring the points by whether or not the market offers a specific product.

Let's look at national `cheese` offerings.




```
us_map + geom_point(data = flt_farm,
                    aes(x = lon, y = lat, color = cheese),
                    size = 2, alpha = 0.6) +
  scale_color_manual(values = c("lightgray", "goldenrod2")) +
  theme(legend.justification = c(0, 0), legend.position = c(0.83, 0),
        legend.background = element_rect(fill=alpha('white', 0.9)))
```

Alternative background maps using `ggmap`

Instead of the plain polygon map background, we can also use the `ggmap::get_map()` function to build upon a variety of more detailed views. Some important inputs:

- `location`: a string address, (`long`, `lat`) pair, or (`left`, `bottom`, `right`, `top`) coordinates of the bounding box
- `zoom`: map zoom (integer values) where `zoom = 3` is for continent, `zoom = 21` is building level, and the default is `zoom = 10` for city.
- `source`: where to pull the map from, options include Google Maps (`"google"`), OpenStreetMap (`"osm"`), and Stamen Map (`"stamen"`)
- `color`: options are `"color"` or black-and-white (`"bw"`)

[1] D. Kahle and H. Wickham. ggmap: Spatial Visualization with ggplot2. The R Journal, 5(1), 144-161.

Alternative background maps using `ggmap`

Instead of the plain polygon map background, we can also use the `ggmap::get_map()` function to build upon a variety of more detailed views. Some important inputs (continued):

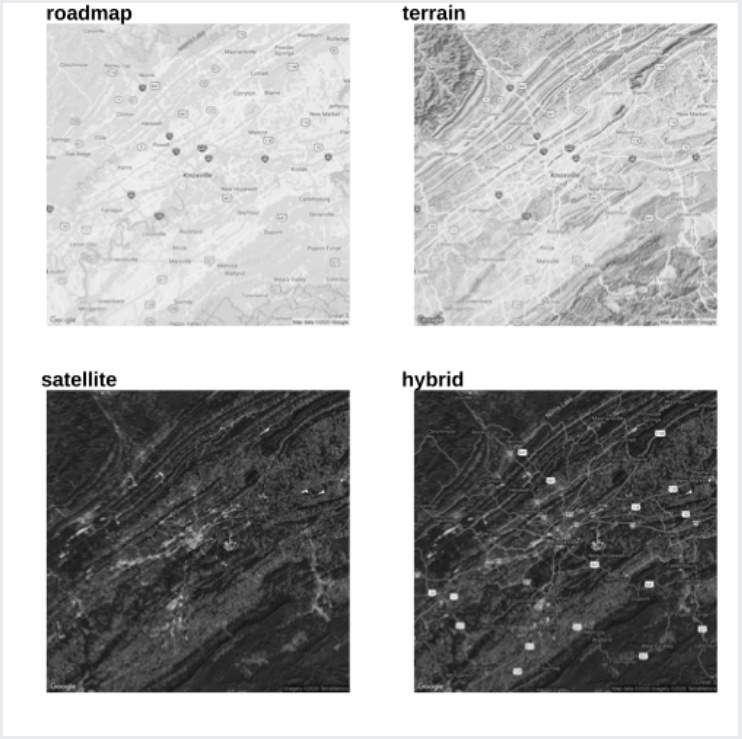
- `maptype`: map theme/aesthetic! Different `source` inputs have different options...
  - when `source = "google"`: `"terrain"`, `"terrain-background"`, `"satellite"`, `"roadmap"`, and `"hybrid"`
  - when `source = "stamen"`: `"terrain"`, `"watercolor"`, and `"toner"`

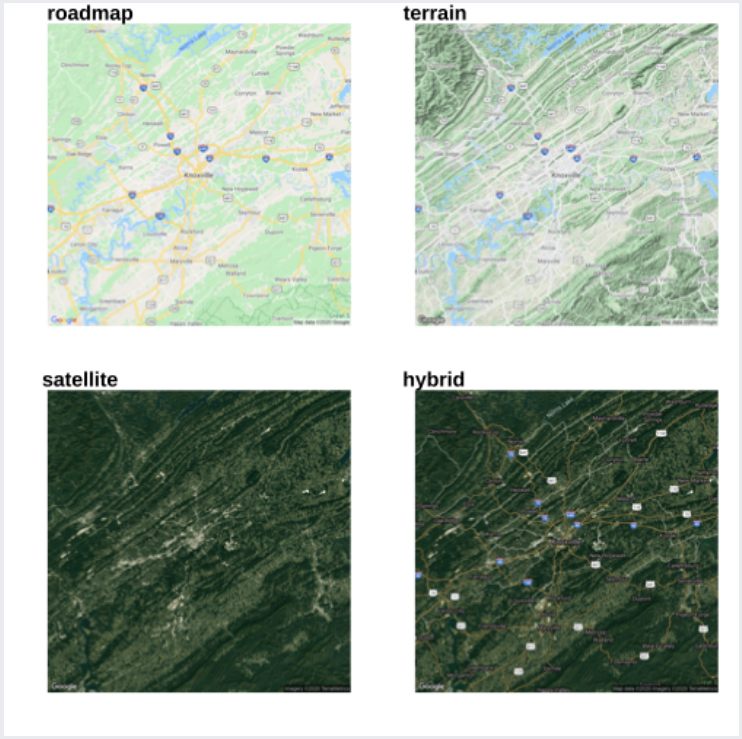
[1] D. Kahle and H. Wickham. ggmap: Spatial Visualization with ggplot2. The R Journal, 5(1), 144-161.

Getting started with `get_map`

```
knox <- get_map(location = 'Knoxville, Tennessee',  zoom = 10,
                maptype = 'roadmap', source = 'google', color = 'bw')
knox_map <- ggmap(knox) +
  theme(axis.ticks = element_blank(), axis.text = element_blank()) +
  xlab('') + ylab('')
```

Map types from Google Maps





Building on Google Maps

```
knox_map + geom_point(data = knx_fm, aes(x = lon, y = lat),
                      color = "darkgreen", size = 8, alpha = 0.6)
```

Focusing your map using `location`

Instead of using `location = 'Knoxville, Tennessee'`, which is based on the city's centroid, we can recenter our map on the 2 nearest Knoxville markets:

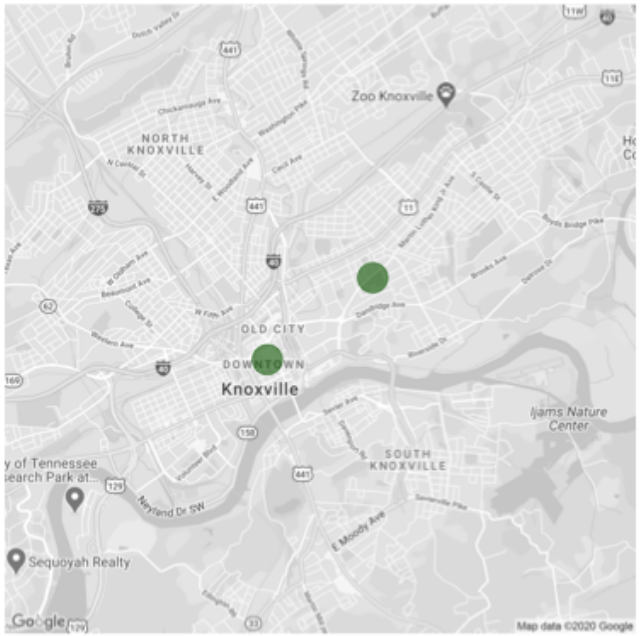
```
##      fmid      lat       lon
## 1 1019624 35.97626 -83.90090
## 2 1019189 35.96469 -83.91925
```

We can center the map in the middle of the markets (i.e., the average latitude and longitude): (-83.910076, 35.970476).

```
knox <- get_map(location = c(-83.91008, 35.97048),
                zoom = 15, maptype = 'roadmap',
                source = 'google', color = 'bw')
knox_map <- ggmap(knox) +
  geom_point(data = knx_fm[c(1:2), ], aes(x = lon, y = lat),
             color = "darkgreen", size = 8, alpha = 0.6) +
  theme(axis.ticks = element_blank(), axis.text = element_blank()) +
  xlab('') + ylab('')
```

And finetuning it with `zoom`

zoom = 13



zoom = 15

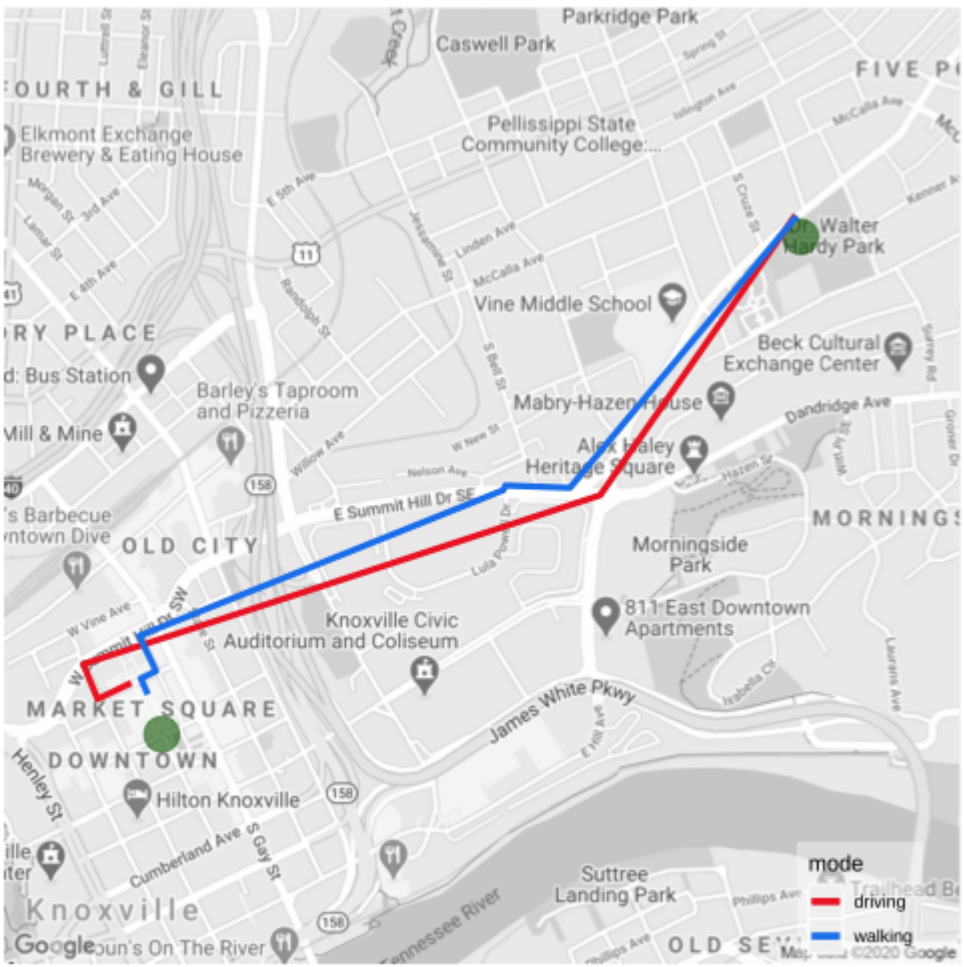
Want to find the balance between zooming too far (and losing context or your data) and being too far away to see details like roads.

Incorporating `route` data

Layer the `route` over a map using `geom_path()` based on the `lat` and `lon` coordinates. Recall that `ggmap::route(..., structure = "route)"` is best suited for this.

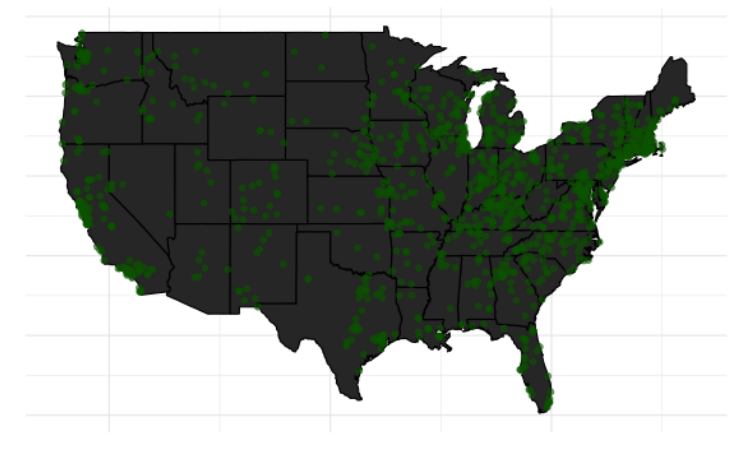
```
drive <- ggmap::route(from = fmid1019189$address,
                      to = fmid1019624$address,
                      mode = "driving",
                      structure = "route")
walk <- ggmap::route(from = fmid1019189$address,
                     to = fmid1019624$address,
                     mode = "walking",
                     structure = "route")
knox_map +
  geom_path(data = drive, aes(x = lon, y = lat, color = "driving")) +
  geom_path(data = walk, aes(x = lon, y = lat, color = "walking")) +
  theme(legend.justification = c(0, 0), legend.position = c(0.83, 0),
        legend.background = element_rect(fill=alpha("white", 0.4))) +
  scale_color_manual(values = c("driving" = "firebrick2",
                                "walking" = "dodgerblue2"),
                     name = "mode")
```

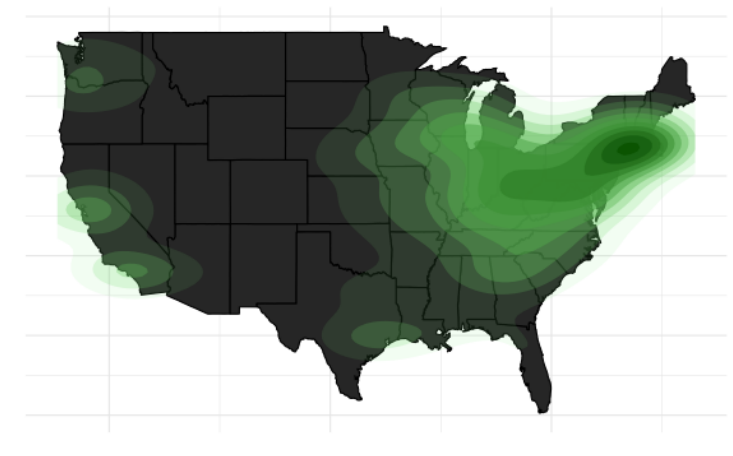
Incorporating `route` data



Density maps

Often, we are less interested in individual locations of each point in our dataset. More so, we inspect potential **clusters** or areas of high concentration.





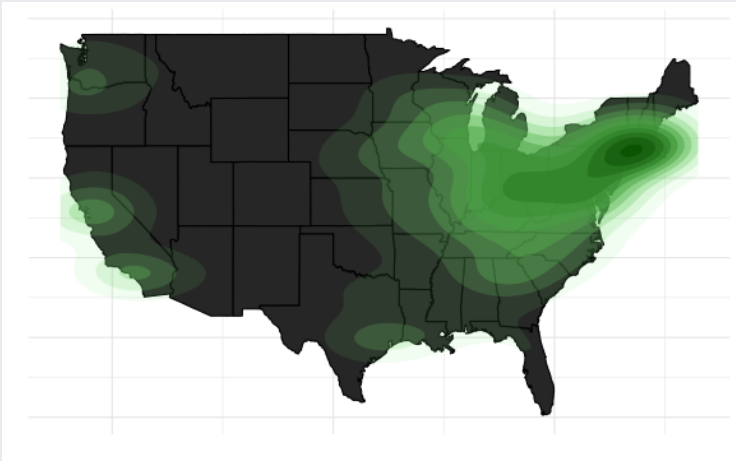
This can be done in `ggplot2` using the `stat_density2d()` function.

[1] A Holder (2018), "Creating a heat map from coordinates using R"

Density maps

```
us_map +
  stat_density2d(data = flt_farm, geom="polygon",
                 aes(x = lon, y = lat,
                     fill = ..level.., alpha = ..level..)) +
  scale_fill_gradient(low = "lightgreen", high = "darkgreen",
                      guide = FALSE) + scale_alpha(guide = FALSE)
```

The `state_density2d()` function is effectively *smoothing* over the (`lat`, `lon`) coordinates of the markets.



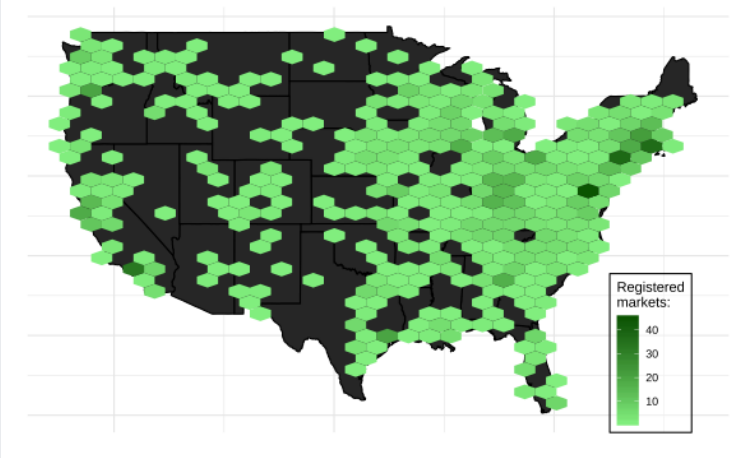
[1] A Holder (2018), "Creating a heat map from coordinates using R"

Concentration maps with hexagons

```
us_map +
  geom_hex(data = flt_farm, aes(x = lon, y = lat)) +
  scale_fill_gradient(low = "lightgreen", high = "darkgreen",
                      name = "Registered\nmarkets:")
```

The `geom_hex()` function... *"divides the plane into regular hexagons, counts the number of cases in each hexagon, and then (by default) maps the number of cases to the hexagon fill."*

This is **raster** data representation, divided into uniform hexagons



[1] Hexagonal heatmap of 2d bin counts

Aggregating point data across space

There are other ways to aggregate point. For example, we might be interested in the **state- or county-level numbers of markets**. This requires two alterations to our `flt_farm` data:

1. Use `dplyr::group_by()` to sum the number of markets by state
2. Use `dplyr::right_join()` to merge the market counts with the US map data

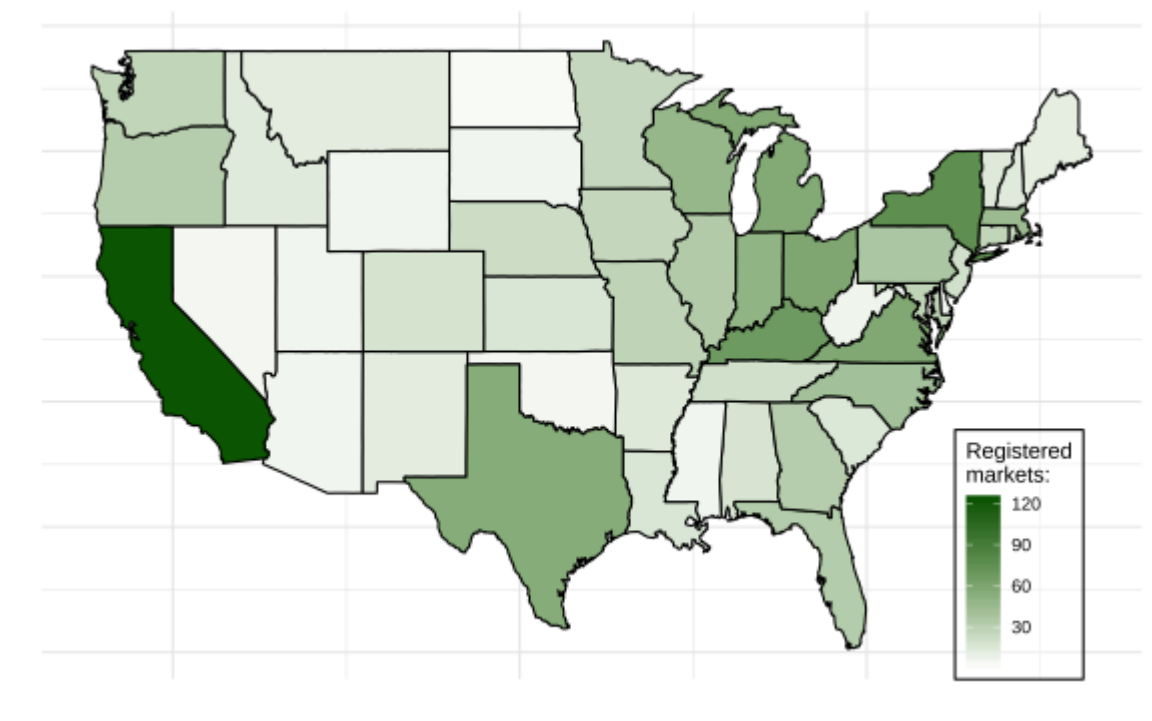
```
flt_farm %>%
  mutate(state = tolower(state)) %>%
  group_by(state) %>%
  summarize(num_markets = n()) %>%
  right_join(contig_us, by = c("state" = "region")) -> flt_farm_summ
```

Maps like this are called **choropleth** maps.

State-level counts of farmers markets

```
flt_farm_summ %>% ggplot() +
  geom_polygon(aes(x=long, y=lat, group=group, fill = num_markets),
               color="black") +
  scale_fill_gradient2(low = "lightgreen", high = "darkgreen",
                       name = "Registered\nmarkets:") +
  theme_minimal() +
  theme(axis.ticks = element_blank(), axis.text = element_blank(),
        legend.justification = c(0, 0),
        legend.position = c(0.83, 0),
        legend.background = element_rect(fill=alpha('white', 0.4))) +
  xlab('') + ylab('')
```

State-level counts of farmers markets



Maps can be misleading

California stands out on our state-level market counts, but not on any of the previous maps. In fact, there are 123 registered California farmers markets (the largest of any state, followed by New York with 76).

This could potentially be explained by:

1. larger land coverage
2. greater population

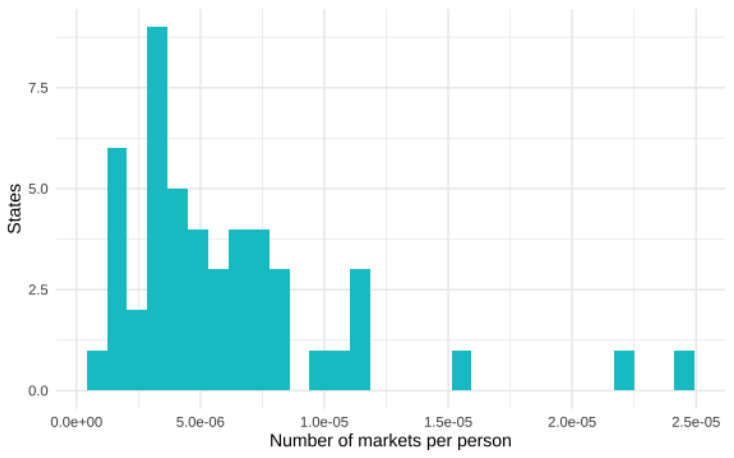
rather than a stronger propensity toward homegrown goods.

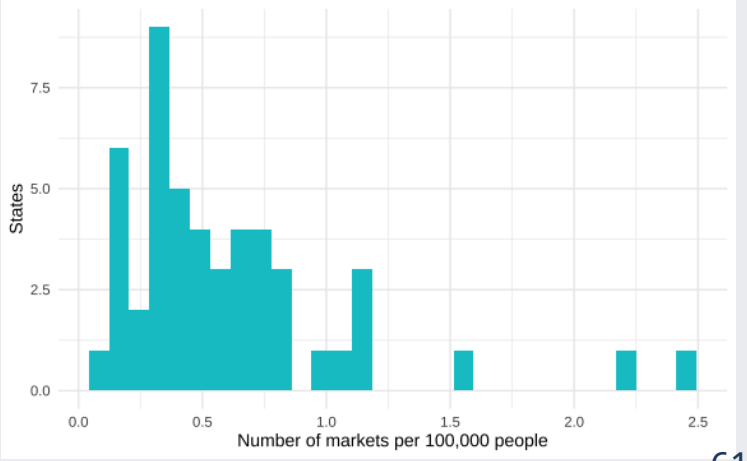
For these reasons, it is often advisable to not simply map "crude" counts and measures, because you can get misleading hotspots that are more related to city centers. To even the playing field, one way to correct for this is to look at per capita counts and measures (or perhaps per mile, etc).

State-level counts of farmers markets *(per capita)*

We merge in state population estimates(US Census Bureau) to map state-level rates of farmers markets *per capita* for comparison. Read `pop` in from the `GitHub`.

```
flt_farm_summ %>% inner_join(pop) %>%
  mutate(per_person = num_markets / pop,
         per_100thou = per_person * 100000) -> flt_farm_summ
```



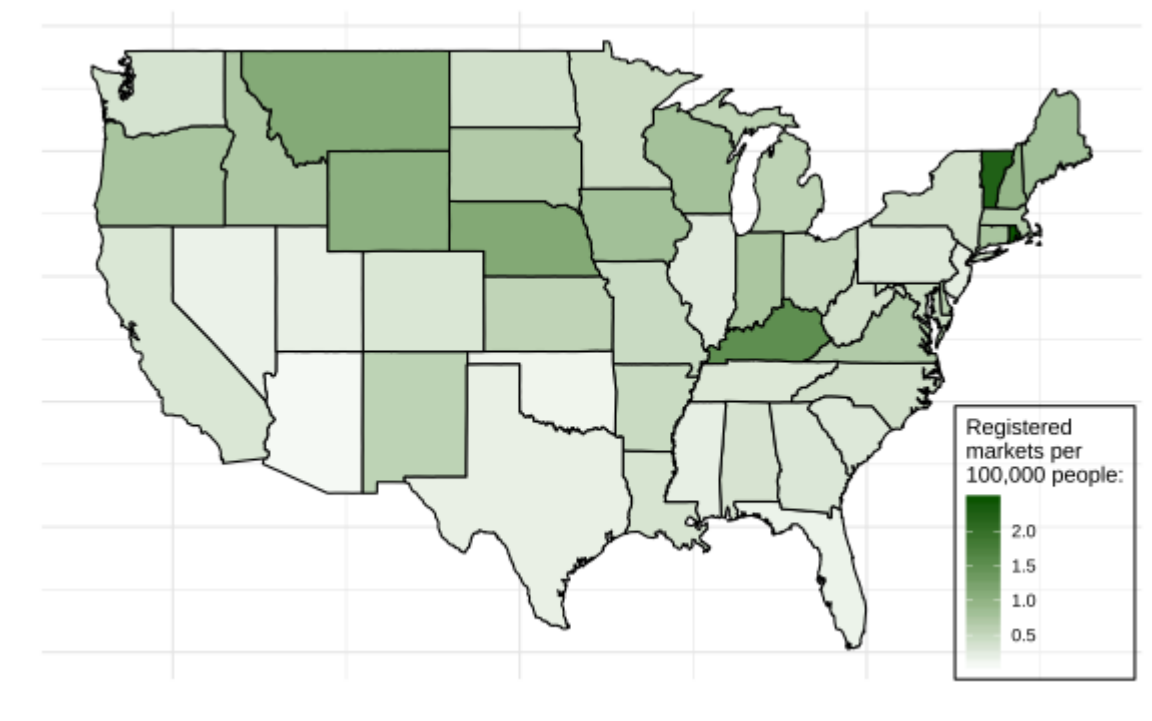


State-level counts of farmers markets *(per capita)*

I choose to color the choropleth by the number of markets per 100,000 people (rather than per person) since it still allows for **fair comparison between states**, but it is easier to see differences between them since the numbers are larger. To do this, set `fill = per_100thou` in your `geom_polygon()` aesthetics.

```
flt_farm_summ %>% ggplot() +
  geom_polygon(aes(x=long, y=lat, group=group, fill = per_100thou),
               color="black") +
  scale_fill_gradient2(low = "lightgreen", high = "darkgreen",
                       name = "Registered\nmarkets per\n100,000 people:"
  theme_minimal() +
  theme(axis.ticks = element_blank(), axis.text = element_blank(),
        legend.justification = c(0, 0),
        legend.position = c(0.83, 0),
        legend.background = element_rect(fill=alpha('white', 0.4))) +
  xlab('') + ylab('')
```

State-level counts of farmers markets *(per capita)*



Thank you!

References

“What Is Spatial Data? The Basics & GIS Examples: FME.” Safe Software, www.safe.com/what-is/spatial-data/.

S. Firmin. “Vector and Raster: A Tale of Two Spatial Data Types.” Velocity Business Solutions Limited, 15 Jan. 2019, www.vebuso.com/2019/01/vector-raster-tale-two-spatial-data-types/.

“Geospatial Data Models.” Humboldt State Geospatial Online, gsp.humboldt.edu/OLM/Courses/GSP_216_Online/lesson3-1/data-models.html.

“Subway Entrances: Map of NYC Subway Entrances.” NYC Open Data, data.cityofnewyork.us/Transportation/Subway-Entrances/drex-xx56.

“Get Started.” Google Maps Platform, Google, developers.google.com/maps/documentation/geocoding/start.

References (cont.)

J. Blossom. “Geocoding Best Practices.” Center for Geographic Analysis, Harvard University, 2 Mar. 2015, gis2.harvard.edu/services/blog/geocoding-best-practices.

D. Kahle and H. Wickham. "ggmap: Spatial Visualization with ggplot2." The R Journal, 5(1), 144-161.

“Farmers Markets Directory and Geographic Data.” Data.gov, Publisher Agricultural Marketing Service, Department of Agriculture, 21 Feb. 2020, catalog.data.gov/dataset/farmers-markets-directory-and-geographic-data.

M. Pineda-Krch. “Great-Circle Distance Calculations in R.” R Bloggers, 12 May 2011, www.r-bloggers.com/2010/11/great-circle-distance-calculations-in-r/.

R. J. Hijmans. "Introduction to the 'geosphere' package (Version 1.5-10)." 25 May 2019, https://cran.r-project.org/web/packages/geosphere/vignettes/geosphere.pdf.

Hodler, Axel. “Creating a Heat Map from Coordinates Using R.” Axel Hodler, 20 Dec. 2018, axelhodler.medium.com/creating-a-heat-map-from-coordinates-using-r-780db4901075.