

SOC 403 Data Workshop

Web Searches and Analysis in R

Brandon Morande
University of Washington

February 4, 2025

I. Introduction

Our class will develop tools to collect data on conditions that might affect social disorder on public transit. Web searches and Census data occasionally offer less expensive and intensive options than direct observation.

In this workshop, we'll learn to gather online data and conduct descriptive analyses. This will include building and subsetting spatial datasets, as well as creating visuals in R (e.g., tables, maps).

You can find the R `Markdown` (.rmd) file and associated R script to create this document on Canvas.

II. Web Searches

Data may already exist for our proposed indicators. For example, the City of Seattle maintains a file with the names and boundaries of [public parks](#). We recommend exploring the following city and county sites:

- [Seattle Open Data](#)
- [Seattle GeoData](#)
- [King County Open Data](#)
- [King County Resource Database](#) (KCRD)

These sites might provide data with geographic information (e.g., coordinates). If so, we can download the files and use R to subset locations within 0.25 miles of our stations. However, some pages may only display interactive maps. In these cases, we need to manually collect information.

We should gather enough data to (1) link the indicator to a station, (2) categorize it for analysis, (3) map its location, and (4) prevent duplicate entries. We'll use this [Google Sheet](#) to document the following details:

- Station Name (e.g., CID)
- Type (e.g., Social Service)
- Feature (e.g., Shelter)
- Name (e.g., Salvation Army)
- Address (e.g., 811 Maynard Ave S, Seattle, WA, 98134)
- Latitude (e.g., 47.59531)
- Longitude (e.g., -122.32541)

However, data may not exist for some of our variables - such as entertainment venues. We'll need to use search engines and mapping platforms to identify nearby sites. Unfortunately, Google Maps does not allow us to set a search radius. If you're unsure whether a feature lies within 0.25 miles, we can still collect the data and confirm the distance in R later. However, you could try third-party tools like [Map Developers](#) or [ArcGIS Online](#) to draw circles or distance lines.

III. R for Managing and Analyzing Data

R is a free, open-source programming language that many scientists use for data analysis and visualization. RStudio offers a user-friendly interface to write and execute R code, as well as view output (e.g., plots). You can install the applications here:

- R-4.4.2 for [Windows](#)
- R-4.4.2 for [macOS](#)
- RStudio from [Posit](#)

Here are some resources for those interested in improving their skills:

- Introductory:
 - *Hands-On Programming with R* ([Grolemund 2014](#))
 - [R cheat sheets](#)
- Intermediate:
 - *R for Data Science* ([Grolemund and Wickham 2023](#))
 - *Data Visualization: A Practical Introduction* ([Healy 2018](#))
 - *Graphical Data Analysis with R* ([Unwin 2015](#))
- Advanced:
 - *Advanced R* ([Wickham 2019](#))

R also offers extensive spatial capabilities, including for US Census data. You can find more texts here:

- *Spatial Data Science: With Applications in R* ([Pebesma & Bivand 2025](#))
- *Geocomputation with R* ([Lovelace et al. 2025](#))
- *Analyzing US Census Data* ([Walker 2023](#))

i. Opening RStudio and Starting Script

After opening RStudio, we can click the “File” tab and select “New File,” then “R Script.” Save the script to a preferred folder. Next, open the “Session” tab and choose “Set Working Directory” to “Source File Location.” This tells R that we’re working from the folder where we saved our script.

We then install any needed packages, which are collections of code, data, and documentation that others create to simplify programming. Paste the following text into the R script, then highlight lines or place the cursor on a line and press “Command + Enter” to run the code.

```
install.packages("readr") # To read in datasets
install.packages("tidyverse") # To clean, subset, and transform data
install.packages("tidycensus") # To query US Census data
install.packages("tigris") # To load Census geographies
install.packages("ggplot2") # To create graphics
install.packages("ggthemes") # To add extra customization for graphics
install.packages("ggspatial") # For different map backgrounds
install.packages("prettymapr") # For different map backgrounds
install.packages("kableExtra") # To format nice tables
install.packages("webshot2") # For saving kable tables
install.packages("sf") # For spatial data and analysis
install.packages("units") # For converting units
```

You only need to *install* packages once (and update them occasionally). However, you need to *load* them into RStudio every time:

```
# Load packages
library(readr)
library(tidyverse)
library(tidycensus)
library(tigris)
library(ggplot2)
library(ggspatial)
library(prettymapr)
library(ggthemes)
library(kableExtra)
library(webshot2)
library(sf)
library(units)

# To use tigris to get Census geographies
options(tigris_use_cache = TRUE)
```

ii. Loading and Cleaning Data

Next, we need to upload our data. Non-spatial and point data often come in .xlsx and .csv formats. For example, we can download data on park restrooms from [Seattle GeoData](#) as a .csv, read it into RStudio with `read_csv()`, and save it as a new dataframe (df) called “restrooms.”

```
# Load data from the 'data' folder within working your directory
restrooms <- read_csv("data/restrooms.csv")
```

We can use the `head()` function to view the beginning of the df. You can see the complete df using `View()` or by clicking it in the “Environment” pane.

```
# View first lines of df
head(restrooms)
## # A tibble: 6 x 37
##   OBJECTID `Park Name`      `Facility Name` `Description (AMWO)` `PMA ID (AMWO)`
##   <dbl> <chr>                <chr>           <chr>                <chr>
## 1    6315 Washington Park~ Washington Par~ ARBORETUM COMPOSTIN~ 393
## 2    6316 Bar-S Playground Bar S Playgrou~ BAR S PLAYGROUND SH~ 3703
## 3    6317 Jefferson Park ~ Jefferson      <NA>                 <NA>
## 4    6318 <NA>              SEWARD PARK FI~ SEWARD PARK FISH HA~ 428
## 5    6319 Volunteer Park  VOLUNTEER PARK~ VOLUNTEER PARK TRAN~ 399
## 6    6320 West Seattle Go~ WEST SEATTLE G~ WEST SEATTLE GOLF R~ 469
## # i 32 more variables: `Location ID (AMWO)` <dbl>, `AMWO ID` <chr>,
## # `Life Cycle Status` <lgl>, `Life Cycle Status Code (AMWO)` <chr>,
## # `Open to Public (AMWO)` <chr>, `Year Constructed (AMWO)` <dbl>,
## # `Season (AMWO)` <chr>, `Current Status (AMWO)` <chr>,
## # `Reason Closed (AMWO)` <chr>, `Season Closure Date (AMWO)` <chr>,
## # `Usage (AMWO)` <chr>, `Sani-Can Onsite (AMWO)` <chr>,
## # `Hours Open (AMWO)` <chr>, `Locked By (AMWO)` <chr>, POINT_X <dbl>, ...
```

This df has 151 observations (rows) and 37 variables (columns). Use `select()` from the `dplyr` package to only retain needed columns. If a variable has a space in its name, we need to add backticks or quotation marks. The concatenate function `c()` allows us to list more than item.

```
# Keep needed columns
restrooms <- restrooms |>
  select(c(OBJECTID, `Park Name`, `Facility Name`, `Open to Public (AMWO)`, `Season (AMWO)`,
           `Hours Open (AMWO)`, `Latitude (AMWO)`, `Longitude (AMWO)`))
```

Not all park restrooms appear open to the public. To only keep the publicly-accessible locations, we can `filter()` “Yes” rows under the “Open to Public” column.

```
# Filter to only include public restrooms
restrooms <- restrooms |>
  filter(`Open to Public (AMWO)` == "YES") # Use two equal signs for specifying a value
```

iii. Transforming Data into a Spatial Object

Linking restrooms to stations requires us to convert the dataset into a “simple features” (`sf`)⁴ object that recognizes spatial information. However, we first need to check if any observations lack coordinates.

```
# See if there is missing (NA) data by column
colSums(is.na(restrooms))
##           OBJECTID           Park Name           Facility Name
##                0                23                1
## Open to Public (AMWO)           Season (AMWO)           Hours Open (AMWO)
##                0                0                1
##           Latitude (AMWO)           Longitude (AMWO)
##                0                0

# Remove observations with missing coordinate data
restrooms <- restrooms |>
  filter(!is.na(`Latitude (AMWO)`), # Remove rows where Latitude is NA
         !is.na(`Longitude (AMWO)`))
```

After removing missing data, we can create a spatial df with `st_as_sf()`. We provide the “coords” argument with the variable names for longitude and latitude. This function creates a new column called “geometry.”

```
# Convert new spatial object called "restrooms_sf"
restrooms_sf <- st_as_sf(restrooms,
  coords = c("Longitude (AMWO)", "Latitude (AMWO)"),
  crs = 4269, # Common CRS for lat/lon in North America
  remove = FALSE) # Setting remove to FALSE keeps the lon/lat columns
```

We should keep the Coordinate Reference System (CRS) consistent across our datasets. A CRS defines the *coordinate system* (e.g., degrees, meters) and *projection* used to represent locations on Earth’s curved surface in two dimensions. We can use the NAD83 Washington North metric projection for Seattle, with an ESPG code of 6596.

```

# # To check CRS st_crs(restrooms_sf)

# If incorrect, transform to appropriate projection
restrooms_sf <- st_transform(restrooms_sf, crs = 6596)

```

iv. Linking Point Data to Stations

Now we're ready to match restrooms to nearby stations. Seattle GeoData provides a dataset with all the Light Rail station locations [here](#). We can download the dataset as a GeoPackage (.gpkg) or Shapefile (.shp) to preserve the spatial features.

GeoPackages may have multiple layers of data. We can check the list of layers with `st_layers()` and then read in the ones we want. Always remember to check and transform the CRS if needed. The geometry column for this df is called "SHAPE."

```

# Check list of layers in gpkg
st_layers("data/light_rail_stations.gpkg")
## Driver: GPKG
## Available layers:
##           layer_name geometry_type features fields
## 1 Light_Rail_Stations      Point      58      13
##           crs_name
## 1 NAD83(HARN) / Washington North (ftUS)

# Read in the file
stations <- read_sf("data/light_rail_stations.gpkg", layer = "Light_Rail_Stations")

# # Check CRS st_crs(light_rail_stations) # EPSG:2926, NAD83(HARN) State Plane
# Washington North (ftUS) is also common for Seattle

# Transform CRS
stations <- st_transform(stations, crs = 6596)

```

Again, we should clean the df to only keep necessary columns. Let's also focus on Capitol Hill, Westlake, and CID stations.

```

# Keep project stations and needed columns
stations <- stations |>
  filter(STATION %in% c("Capitol Hill", # Use %in% for multiple values
                      "Westlake",
                      "International District / Chinatown"))
)|>
select(c(STATION,
        SHAPE))

```

Using `st_buffer()`, we can create a 0.25 mile radius around each station (in meters for our CRS). Notice how the "SHAPE" column now contains polygons instead of points.

```

# Specify buffer distance
distance <- 0.25 * 1609.34 # 0.25 miles in meters

```

```
# Create radius polygons
stations_buffer <- st_buffer(stations, dist = distance)
```

We can then match restrooms to light rail station buffers using `st_join()`. We specify the “join” argument as `st_intersects` because we want to match any park whose point falls within or on the edge of a station radius. Here’s a [cheatsheet](#) for the different `st_()` functions.

```
# Match restrooms to stations buffer (for later tables)
stations_restrooms <- stations_buffer |>
  st_join(restrooms_sf, join = st_intersects)

# Match stations to restrooms (for later mapping) This keeps restroom
# geometries
restrooms_sf <- restrooms_sf |>
  st_join(stations_buffer, join = st_intersects)
```

We can create a table with counts of the restrooms using `kable`. `Kable` will print nicely with R Markdown, then you can crop it from the `.pdf` or `.html` file.

```
# Create a df of restroom counts
restrooms_count <- stations_restrooms |>
  # Group by station
  group_by(STATION) |>
  # Sum up the restroom rows that aren't NA
  summarize(Count = sum(!is.na(`Facility Name`))) |>
  # Remove the geometry column
  st_drop_geometry()

# Construct table with kable
restrooms_table <- knitr::kable(restrooms_count,
  row.names = FALSE, # Remove numbers in front of rows
  col.name = c("Station", "Count"), # Name the columns
  caption = "Number of Public Restrooms Within .25 miles")

# Print table
restrooms_table
```

Table 1: Number of Public Restrooms Within .25 miles

Station	Count
Capitol Hill	1
International District / Chinatown	0
Westlake	0

v. Linking Polygon Data to Stations

We can follow a similar process for polygon data, such as [public parks](#).

```

# Check list of layers in gpkg
st_layers("data/parks.gpkg")
## Driver: GPKG
## Available layers:
##           layer_name geometry_type features fields
## 1 Parks_Boundary__outline_ Multi Polygon      513      5
##           crs_name
## 1 NAD83(HARN) / Washington North (ftUS)

# Read in the file
parks <- read_sf("data/parks.gpkg", layer = "Parks_Boundary__outline_")

# # Check CRS st_crs(parks) # EPSG:2926

# Transform CRS
parks <- st_transform(parks, crs = 6596)

```

After loading and transforming our spatial data, we can select our needed columns and then match parks to the “stations_buffer” df.

```

# Select needed park columns
parks <- parks |>
  select(c(NAME, SHAPE))

# Match parks to stations (for tables)
stations_parks <- stations_buffer |>
  st_join(parks, join = st_intersects)

# Match stations to parks (for mapping)
parks <- parks |>
  st_join(stations_buffer, join = st_intersects)

```

Now let’s create another table of counts.

```

# Create df of park counts
parks_count <- stations_parks |>
  group_by(STATION) |>
  summarize(Count = sum(!is.na(NAME))) |>
  st_drop_geometry()

# Construct table with kable
parks_table <- knitr::kable(parks_count, row.names = FALSE, col.name = c("Station",
  "Count"), caption = "Number of Public Parks Within .25 miles")

# Print table
parks_table

```

Table 2: Number of Public Parks Within .25 miles

Station	Count
Capitol Hill	3
International District / Chinatown	5
Westlake	3

IV. R for US Census Data

We can also analyze neighborhood demographic characteristics in R. For our class, we'll apply data from the 2020 decennial Census at the *block* level - the smallest geographic unit available. The US Census Bureau also provides yearly population *estimates* via the American Community Survey, but only for larger areas. The public can manually download datasets from data.census.gov or the National Historic Geographic Information System (NHGIS). However, the `tidycensus` package allows us to request data directly in R. If the following code doesn't work, you may need to obtain an Application Programming Interface (API) key - just fill in your organization (University of Washington) and school email [here](#).

Then set and save your API key for future R sessions.

```
## Set and save Census API key census_api_key('your_api_key_here', install =
# TRUE, overwrite = TRUE)
```

Next, we can pull 2020 total population data ("P1_001N") for King County blocks using `get_decennial()`. You can find other variable codes [here](#). Replacing the `variables` argument with `table` allows you to download every variable for that category.

```
# Request decennial population data
block_pop <- get_decennial(geography = "block",
  state = "WA",
  county = "King",
  year = 2020,
  geometry = TRUE, # To get block geometry
  #table = "P1", # To download all household/demographic data
  variables = "P1_001N"
)

# Transform crs
block_pop <- st_transform(block_pop, crs = 6596)
```

We want to know both population size and density, so let's calculate the area of blocks. The `mutate()` function creates or edits column values, while `set_units()` allows us to change from meters² to miles².

```
# Calculate area
block_pop <- block_pop |>
  mutate(area_sqmile = set_units(st_area(geometry), "mile^2"))
```

Now we match blocks that fall within 0.25 miles of stations. For simplicity, let's count all polygons that intersect with a buffer zone.

```
# Match block populations to stations
stations_block_pop <- stations_buffer |>
  st_join(block_pop, join = st_intersects)
```

We can now sum the block populations and areas for each station, then calculate population densities.

```
# Calculate population size and density
stations_total_pop <- stations_block_pop |>
  group_by(STATION) |>
  # Sum the block population and area values by station
  summarize(tot_pop = sum(value),
            tot_area = sum(area_sqmile)) |>
  # Create new density column
  mutate(pop_dens = tot_pop/tot_area) |>
  st_drop_geometry()

# Create table
pop_table <- stations_total_pop |>
  # Drop units from values using as.numeric
  mutate(tot_area = round(as.numeric(tot_area), 2), # Round to two places
         pop_dens = round(as.numeric(pop_dens))) |> # Remove decimals
  knitr::kable(
    row.names = FALSE,
    col.name = c("Station", "Size", "Area (sq mile)", "Density"),
    caption = "Surrounding Population Size and Density")

# Print table
pop_table
```

Table 3: Surrounding Population Size and Density

Station	Size	Area (sq mile)	Density
Capitol Hill	9714	0.26	36706
International District / Chinatown	5927	0.32	18674
Westlake	7610	0.26	29457

V. Creating Maps

R supports numerous packages for creating maps, including base `plot`, `ggplot2`, `tmap`, and `leaflet`. We'll use `ggplot2` given its accessibility, flexibility, and popularity for creating graphics.

Let's plot a map of the Capitol Hill station with its buffer zone. Here, the `annotation_map_tile()` function supplies Open Street Maps (OSM) as the map background.

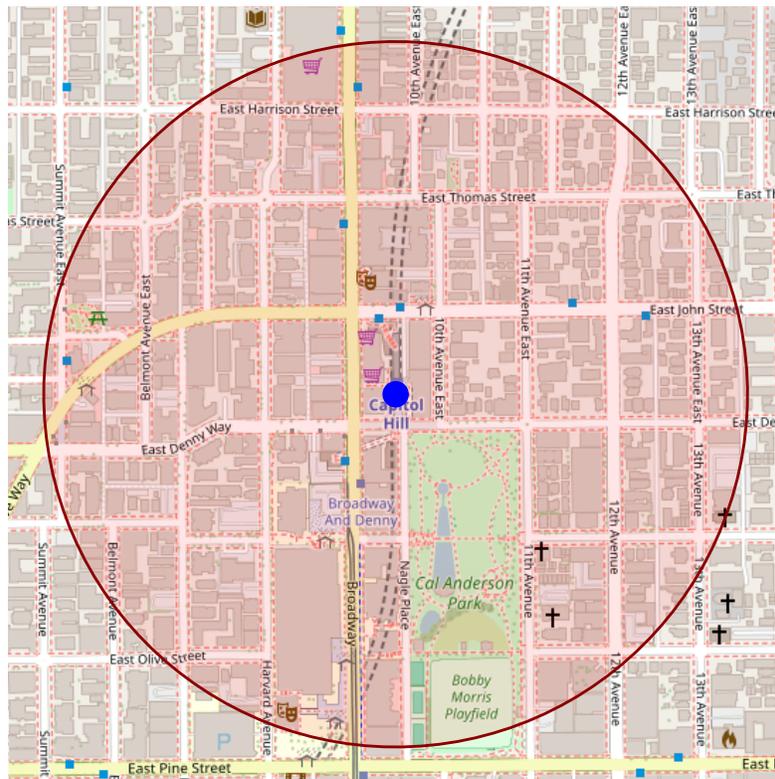
```
# Map of Capitol Hill
(capitol_hill_map <- ggplot() +
  # Add DSM background
  annotation_map_tile(type = "osm",
                    zoom = 16) + # Can change zoom of the map
```

```

# Add station buffer zone
geom_sf(data = subset(stations_buffer, STATION == "Capitol Hill"),
  aes(geometry = SHAPE),
  color = "darkred", # Color the boundary
  linewidth = 0.5, # Set width of line
  fill = "red", # Color in circle
  alpha = 0.1) + # Make circle transparent to see map background
# Add station point
geom_sf(data = subset(stations, STATION == "Capitol Hill"),
  aes(geometry = SHAPE),
  color = "blue",
  size = 4) + # Change size of point
# Add a title
labs(title = "Capitol Hill Station") +
# Remove axes
theme(
  axis.text.x = element_blank(),
  axis.ticks.x = element_blank(),
  axis.text.y = element_blank(),
  axis.ticks.y = element_blank()
)
)

```

Capitol Hill Station



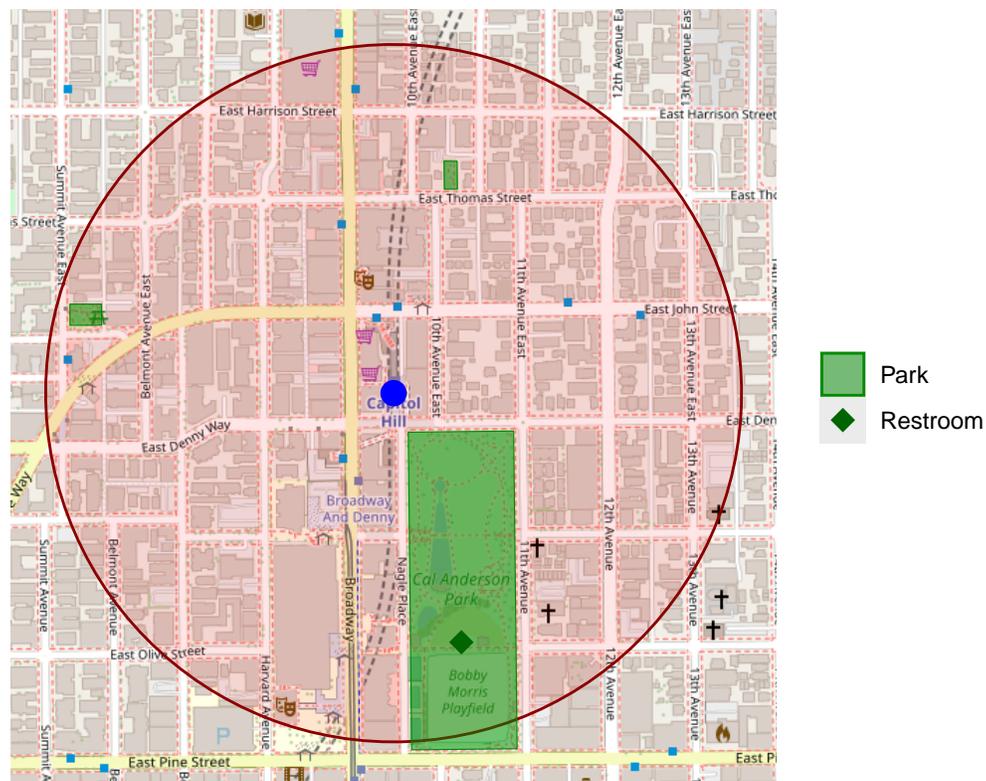
Then we add nearby parks and restrooms.

```

# Add parks and restrooms to map
(capitol_hill_map <-
  # Start with existing map
  capitol_hill_map +
  # Add items
  geom_sf(data = subset(parks, STATION == "Capitol Hill"),
    aes(geometry = SHAPE,
      color = "Park"), # Add parks to color legend
    fill = "green4", # Fill polygon
    alpha = 0.5) +
  geom_sf(data = subset(restrooms_sf, STATION == "Capitol Hill"),
    aes(geometry = geometry,
      color = "Restroom"), # Add restrooms to color legend
    size = 4, shape = 18) +
  # Add a legend and specify colors
  scale_color_manual(name = NULL, # Remove legend title
    values = c("Park" = "green4",
      "Restroom" = "darkgreen")) +
  # Customize the legend
  guides(color = guide_legend(override.aes = list(size = 4, # Increase size of points
    fill = "green4"))) # Fill polygon
)

```

Capitol Hill Station



We can save our maps and other plots using `ggsave()`.

```
# Save map, specifying file path, dimensions, and resolution
ggsave(plot = capitol_hill_map, "figures/capitol_hill_map.pdf", width = 6, height = 6,
       dpi = 300)
```

However, what if we want a simple background that does not contain lots of elements? We could map Census block boundaries instead. We can download geographies directly to R with the `blocks()` function from the `tigris` package. You could also manually read in data.

```
# Download census block geography
block_geo <- blocks(state = "WA", county = "King", year = 2020, class = "sf")
```

Now let's recreate the Capitol Hill map with a new background.

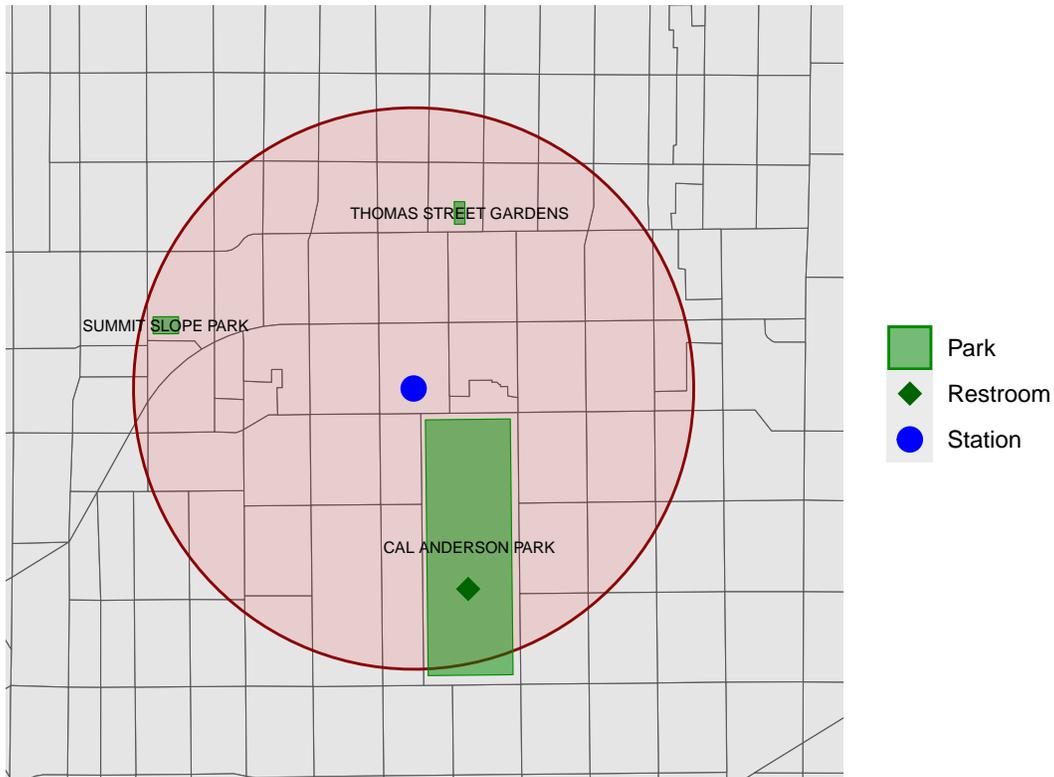
```
# Map of Capitol Hill
(capitol_hill_map_simple <- ggplot() +
  # Add Census boundaries
  geom_sf(data = block_geo, aes(geometry = geometry)) +
  # Add station buffer zone
  geom_sf(data = subset(stations_buffer, STATION == "Capitol Hill"),
          aes(geometry = SHAPE, color = "darkred", linewidth = 0.5,
              fill = "red", alpha = 0.1) +
  # Add station point
  geom_sf(data = subset(stations, STATION == "Capitol Hill"),
          aes(geometry = SHAPE, color = "Station"), size = 4) +
  # Add parks
  geom_sf(data = subset(parks, STATION == "Capitol Hill"),
          aes(geometry = SHAPE, color = "Park"),
          fill = "green4", alpha = 0.5) +
  # Add park label
  geom_text(data = subset(parks, STATION == "Capitol Hill"),
            aes(x = st_coordinates(st_centroid(SHAPE))[, 1], # Extract x-coords
                y = st_coordinates(st_centroid(SHAPE))[, 2], # Extract y-coords
                label = NAME),
            size = 2) +
  # Add restrooms
  geom_sf(data = subset(restrooms_sf, STATION == "Capitol Hill"),
          aes(geometry = geometry, color = "Restroom"),
          size = 4, shape = 18) +
  # Add legend
  scale_color_manual(name = NULL,
                     values = c("Station" = "blue",
                                "Park" = "green4",
                                "Restroom" = "darkgreen")) +
  # Customize legend
  guides(color = guide_legend(override.aes = list(size = 4, fill = "green4"))) +
  # Add a title
  labs(title = "Capitol Hill Station") +
  # Remove axes
  theme(
    axis.title = element_blank(),
    axis.text.x = element_blank(),
    axis.ticks.x = element_blank(),
```

```

axis.text.y = element_blank(),
axis.ticks.y = element_blank()
) +
# Zoom into Capitol Hill
coord_sf(xlim = c(-122.328, -122.312), ylim = c(47.614, 47.624), expand = FALSE)
)

```

Capitol Hill Station



```

# Save plot
ggsave(plot = capitol_hill_map_simple, "figures/capitol_hill_map.pdf",
        width = 6, height = 6, dpi = 300)

```

We could add other elements to this map, as well as change font type and size. Conveying all the information we want often requires creativity, so have fun! If you have any questions, please email me (bmorande@uw.edu) or reference the resources linked in this document.