Spatial Analysis in R

M. Jordan and TA Gelmi-Candusso

2024-05-15

Spatial data

Spatial data is any type of data that directly or indirectly references a specific geographical area or location. These can be, for example, geographic features on the landscape or environmental properties of an area such as temperature or air quality.

Spatial data can be continuous or discrete just like regular data, and in both cases it can be represented as a vector or a raster. The main difference being vector data uses points and lines to represent spatial data, while raster data represents data uses pixelled or gridded, where each pixel/cell represents a specific geographic location and the information therein. Raster data will be heavily influenced by the size of the pizels/cells, i.e. resolution.

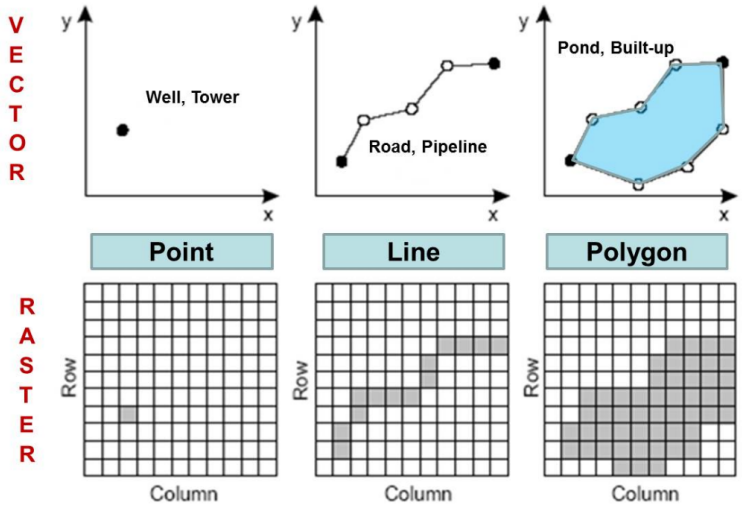
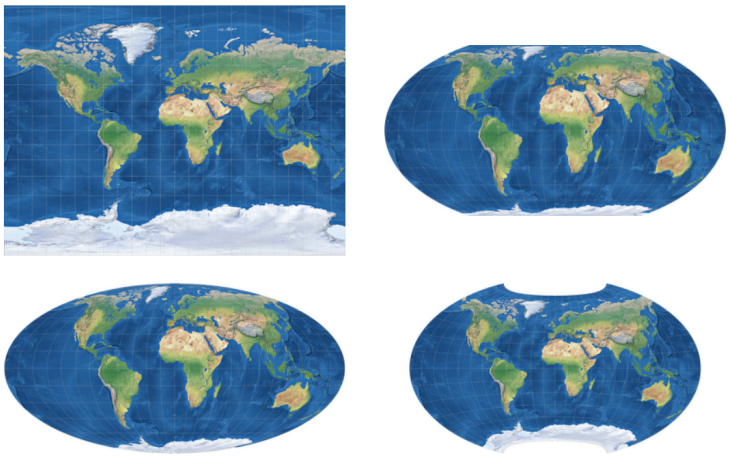


Figure 1: Figure showing difference between vectors and rasters

Both vector and raster data are planar representations of the world, a 3-dimensional sphere, and as such are not perfect copies. Depending on how the planar representation is created it will distort more or less certain areas of the world, therefore many representations exist. These are called projections, as the representations project the 3 dimensional spheric image into a planar, 2-dimensional image.

Maps with different projections are not comparable and cannot be overlaid. Therefore, we need to make sure we work always on the same projection when using more maps. In addition, projections can have different coordinate systems and therefore when extracting distance information from maps, some projections (i.e. metric based projections e.g. Mercator projection or the Albers equal-area projection) will give a more accurate representation of the distance than others. To work around this, we can transform our maps between projections, in R we use EPSG codes to do this.



Left to right, top to bottom: Miller, Wagner IV, Mollweide, Canters W14.

Figure 2: Figure showing difference between vectors and rasters

Vector data

In this section we will read and manipulate vector data in R. * Vector data represents real world features within the GIS environment. A feature is anything you can see on the landscape. * Vector data is commonly stored as a shapefile and can contain either point data or polygon data. * Data contains information attached to each feature, we call these attributes.

Features can be points (red) representing specific x,y locations, such as a trees or camera sites; polygons (white) representing areas, such as forests or residential areas; and lines (yellow/green and blue) representing continuous linear features, such as roads or rivers

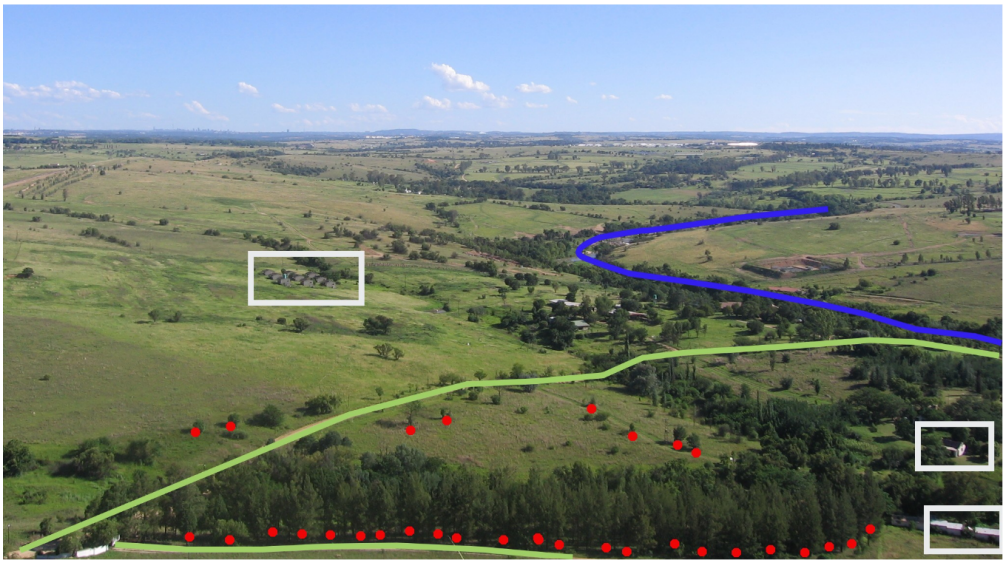


Figure 3: Figure showing polygons, points and lines in the landscape

Vector data reads as a data frame would, each row is a feature and each column is an attribute, and contains usually a geometry column where the xy coordinates for the shapes are stored. Plotting these data will plot the points or shapes in the map using the xy coordinates stored for each feature.

```
## Simple feature collection with 6 features and 3 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: -122.3607 ymin: 47.64339 xmax: -122.3607 ymax: 47.64339
## Geodetic CRS:  WGS 84
##     speciesname    locationid                date                    geometry
## 1 Procyon lotor SEWA_N01_DRP2 2019-07-03 05:02:21 POINT (-122.3607 47.64339)
## 2 Procyon lotor SEWA_N01_DRP2 2019-07-03 06:04:45 POINT (-122.3607 47.64339)
## 3 Procyon lotor SEWA_N01_DRP2 2019-07-03 06:12:10 POINT (-122.3607 47.64339)
## 4 Procyon lotor SEWA_N01_DRP2 2019-07-03 06:13:16 POINT (-122.3607 47.64339)
## 5 Procyon lotor SEWA_N01_DRP2 2019-07-05 03:55:53 POINT (-122.3607 47.64339)
## 6 Procyon lotor SEWA_N01_DRP2 2019-07-05 04:05:21 POINT (-122.3607 47.64339)
```

Packages used to read an manipulate data include the sf package, that reads the shapefile as a spatial data frame, and the terra package that reads the shapefiles as a Spatvector, previously there was also the raster package, but we will try to avoid it as it has been deprecated.

```
library(sf)
library(terra)
```

Vector data: points

Point data can be obtained directly from a shapefile or a csv file where each row is a feature. In this case we will work with camera trap site data and the information collected at each site, i.e. point.

The camera trap sites here are located in Seattle, and have captured coyote and raccoon presence and absence from the 2019 spring season to the 2021 winter season.

The data is stored as a data frame in a csv.

```
captures.table <- read.csv("data/captures.csv")
print(head(captures.table))
```

```
##     speciesname    locationid                date latitude longitude
## 1 Procyon lotor SEWA_N01_DRP2 2019-07-03 05:02:21 47.64339 -122.3607
## 2 Procyon lotor SEWA_N01_DRP2 2019-07-03 06:04:45 47.64339 -122.3607
## 3 Procyon lotor SEWA_N01_DRP2 2019-07-03 06:12:10 47.64339 -122.3607
## 4 Procyon lotor SEWA_N01_DRP2 2019-07-03 06:13:16 47.64339 -122.3607
## 5 Procyon lotor SEWA_N01_DRP2 2019-07-05 03:55:53 47.64339 -122.3607
## 6 Procyon lotor SEWA_N01_DRP2 2019-07-05 04:05:21 47.64339 -122.3607
```

The coordinates are stored in the latitude and longitude, to be able to observe these points in the map, and extract environmental information based on their location, we will have to convert it to a spatial data frame object. We will use the st_as_sf() function from the sf package and we will specify the projection (crs). How do we know which projection our data is in?

<>(This section will also introduce the idea of a CRS. Specifically introduce 4326 and 26910. Tell people how to find their UTM and state plane (others?))

```
captures.spatial <- st_as_sf(captures.table,
                     coords = c("longitude","latitude"),
                     crs = 4326)
print(head(captures.spatial))
```

```
## Simple feature collection with 6 features and 3 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: -122.3607 ymin: 47.64339 xmax: -122.3607 ymax: 47.64339
## Geodetic CRS:  WGS 84
##     speciesname    locationid                date                    geometry
## 1 Procyon lotor SEWA_N01_DRP2 2019-07-03 05:02:21 POINT (-122.3607 47.64339)
## 2 Procyon lotor SEWA_N01_DRP2 2019-07-03 06:04:45 POINT (-122.3607 47.64339)
## 3 Procyon lotor SEWA_N01_DRP2 2019-07-03 06:12:10 POINT (-122.3607 47.64339)
## 4 Procyon lotor SEWA_N01_DRP2 2019-07-03 06:13:16 POINT (-122.3607 47.64339)
## 5 Procyon lotor SEWA_N01_DRP2 2019-07-05 03:55:53 POINT (-122.3607 47.64339)
## 6 Procyon lotor SEWA_N01_DRP2 2019-07-05 04:05:21 POINT (-122.3607 47.64339)
```

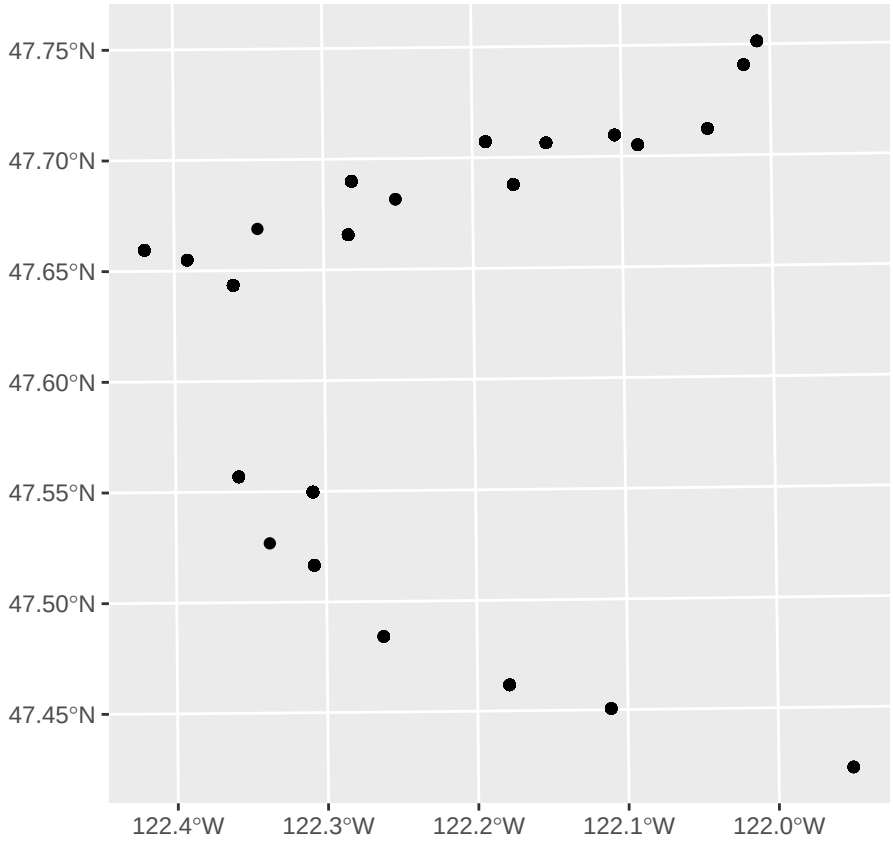
We want our data to be in the NAD83 projection, because we need our data in the UTM coordinate system to be compatible with google map <>(I dont know if I understood correctly the comment below but wrote a draft anyways.) <>(Transform to UTM. Will only plot on google map if it's in lat/lon, so we need to think about where we introduce this.)

```
captures.utm <- st_transform(captures.spatial, crs = 26910)
print(head(captures.utm))
```

```
## Simple feature collection with 6 features and 3 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: 548015.5 ymin: 5276863 xmax: 548015.5 ymax: 5276863
## Projected CRS: NAD83 / UTM zone 10N
##     speciesname    locationid                date                  geometry
## 1 Procyon lotor SEWA_N01_DRP2 2019-07-03 05:02:21 POINT (548015.5 5276863)
## 2 Procyon lotor SEWA_N01_DRP2 2019-07-03 06:04:45 POINT (548015.5 5276863)
## 3 Procyon lotor SEWA_N01_DRP2 2019-07-03 06:12:10 POINT (548015.5 5276863)
## 4 Procyon lotor SEWA_N01_DRP2 2019-07-03 06:13:16 POINT (548015.5 5276863)
## 5 Procyon lotor SEWA_N01_DRP2 2019-07-05 03:55:53 POINT (548015.5 5276863)
## 6 Procyon lotor SEWA_N01_DRP2 2019-07-05 04:05:21 POINT (548015.5 5276863)
```

Let's observe the spatial distribution of the points by plotting them using the ggplot2 package. The geom_sf() function will allow us to plot the spatial data frame object.

```
library(ggplot2)
ggplot(captures.utm) + geom_sf()
```

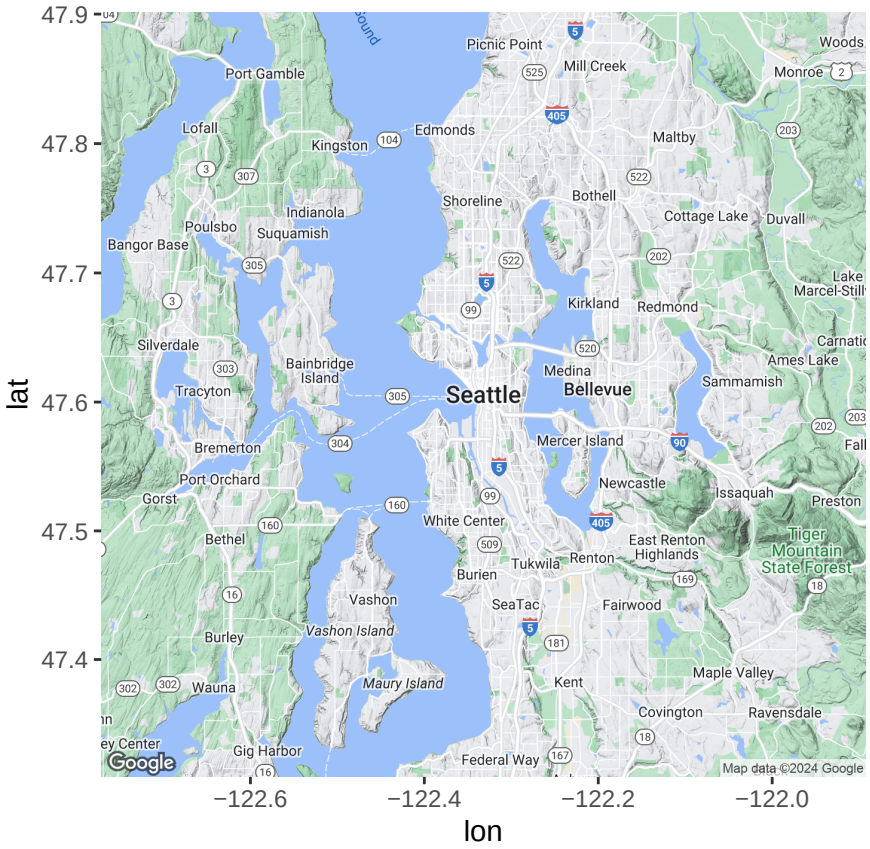


There is no basemap in this plot, we want to add a reference so we can easily distinguish between locations. We will use google maps for this, first we load the ggmap package and register an api from google. <>(Use an API key for the 'uwin-mapping' project that I created for this. <>(Describe in the Rmd how to get your own setup API key to use)

```
library(ggmap)
my_api <- 'AIzaSyBt73bzxdvlS6ioit4OTCaIE6SrZJ9aWnA'
register_google(key = my_api)
```

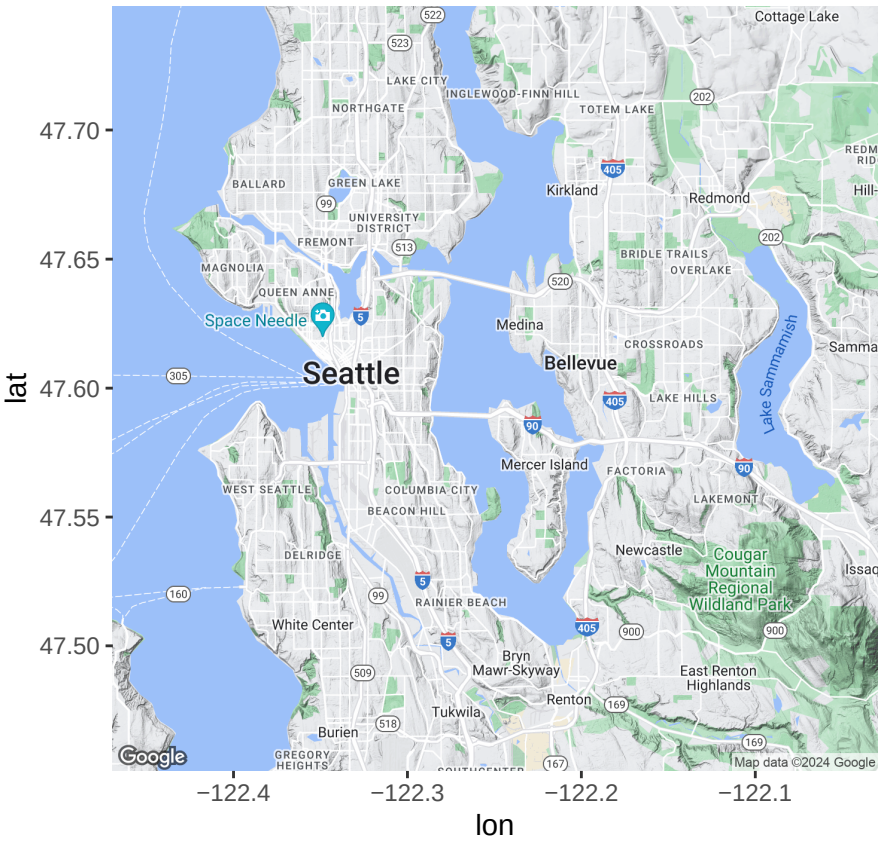
We then get the map relevant to our region using the get_map() function. This can be done both using a bounding box with coordinate information if we want a specific study area, or just the city's name.

```
seattle <- get_map("seattle", source= "google", api_key = my_api)
ggmap(seattle)
```



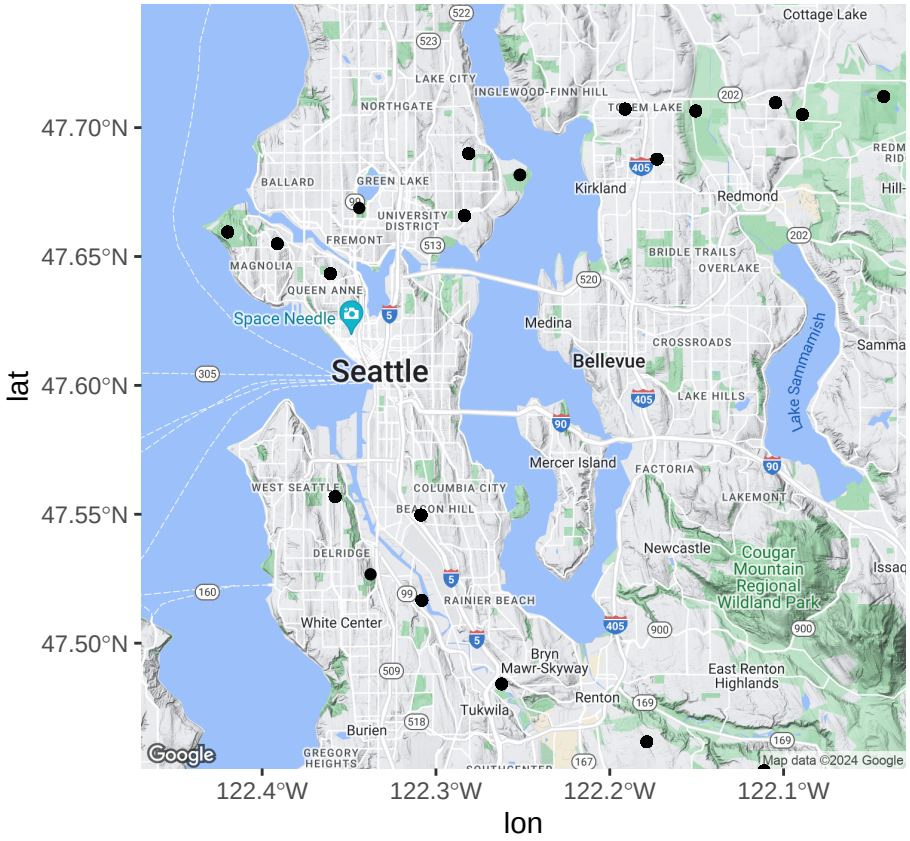
If we use a bounding box, the code will look like this:

```
seattle <- get_map(location = c(left = -122.5, bottom = 47.4,
                                right = -122.0, top = 47.8),
                   source ="google", api_key = my_api)
ggmap(seattle)
```



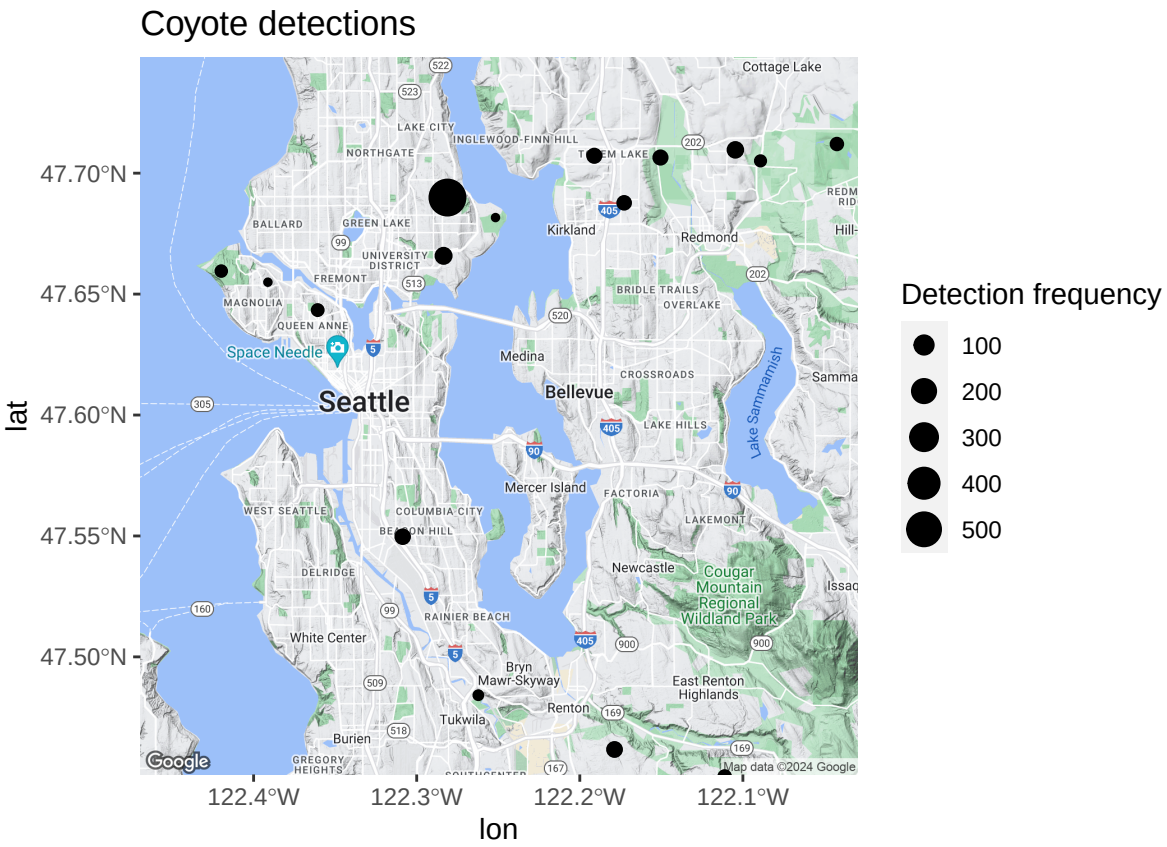
Now we can plot our camera site locations on the Seattle map <>(note the original crs works with google, not the utm.)

```
ggmap(seattle) +
  geom_sf(data=captures.spatial, inherit.aes = FALSE)
```



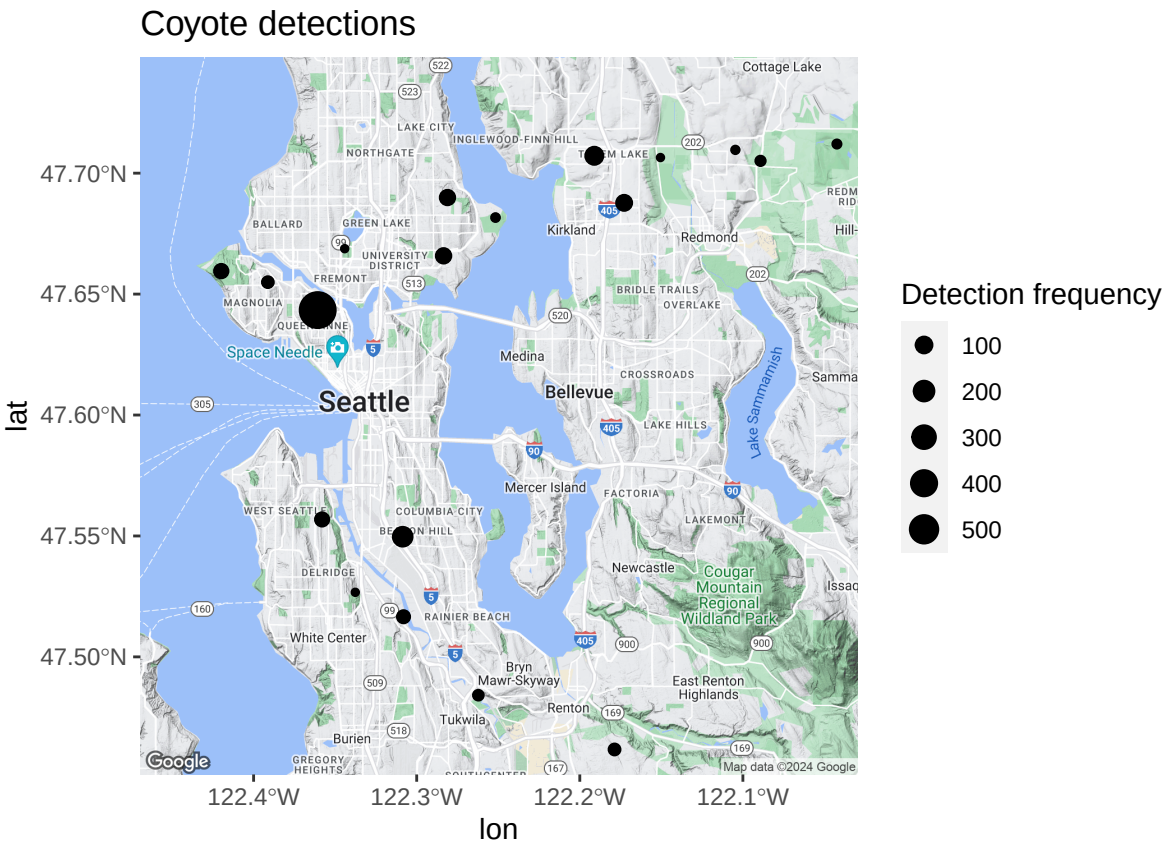
Now lets plot on a map the coyotes captured at each the camera trap sites. We will filter the data based on species name, using the dplyr package, and count detections at each site. We will then plot using the function seen above, but setting point size based on the number of detections at each site.

```
library(dplyr)
coyotes <- filter(captures.spatial, speciesname == "Canis latrans") %>%
  group_by(locationid) %>%
  summarize(detections = n())
ggmap(seattle) +
  geom_sf(data = coyotes, inherit.aes = FALSE, aes(size = detections)) +
  ggtitle("Coyote detections") +
  labs(size = "Detection frequency") +
  scale_size_continuous(breaks=seq(100, 500, by=100))
```



Now try to do the same for raccoons. <>(we can hide code below using echo=FALSE, but depends on how we knit this document, maybe not a good idea for the pdf)

```
raccoons <- filter(captures.spatial, speciesname == "Procyon lotor") %>%
  group_by(locationid) %>%
  summarize(detections = n())
ggmap(seattle) +
  geom_sf(data = raccoons, inherit.aes = FALSE, aes(size = detections)) +
  ggtitle("Coyote detections") +
  labs(size = "Detection frequency") +
  scale_size_continuous(breaks=seq(100, 500, by=100))
```



Vector data: lines

We will look into vector data in the form of lines using the TIGER database for Washington, composed of primary and secondary roads. The spatial object will be read in the same way as we did the points, but in this case we will load directly the shapefile containing the features, downloaded from here

The dataset contains 6 attributes (fields) for each feature.

```
roads <- st_read("maps/roads/tl_2019_53_prisecroads.shp")
```

```
## Reading layer 'tl_2019_53_prisecroads' from data source
##   'C:\Users\tizge\Documents\Spatial Analysis in R Workshop\spatial_analysis_workshop_UWIN\maps\roads'
##   using driver 'ESRI Shapefile'
## Simple feature collection with 2912 features and 4 fields
## Geometry type: LINESTRING
## Dimension:     XY
## Bounding box:  xmin: -124.6384 ymin: 45.55945 xmax: -117.0359 ymax: 49.00241
## Geodetic CRS:  NAD83
```

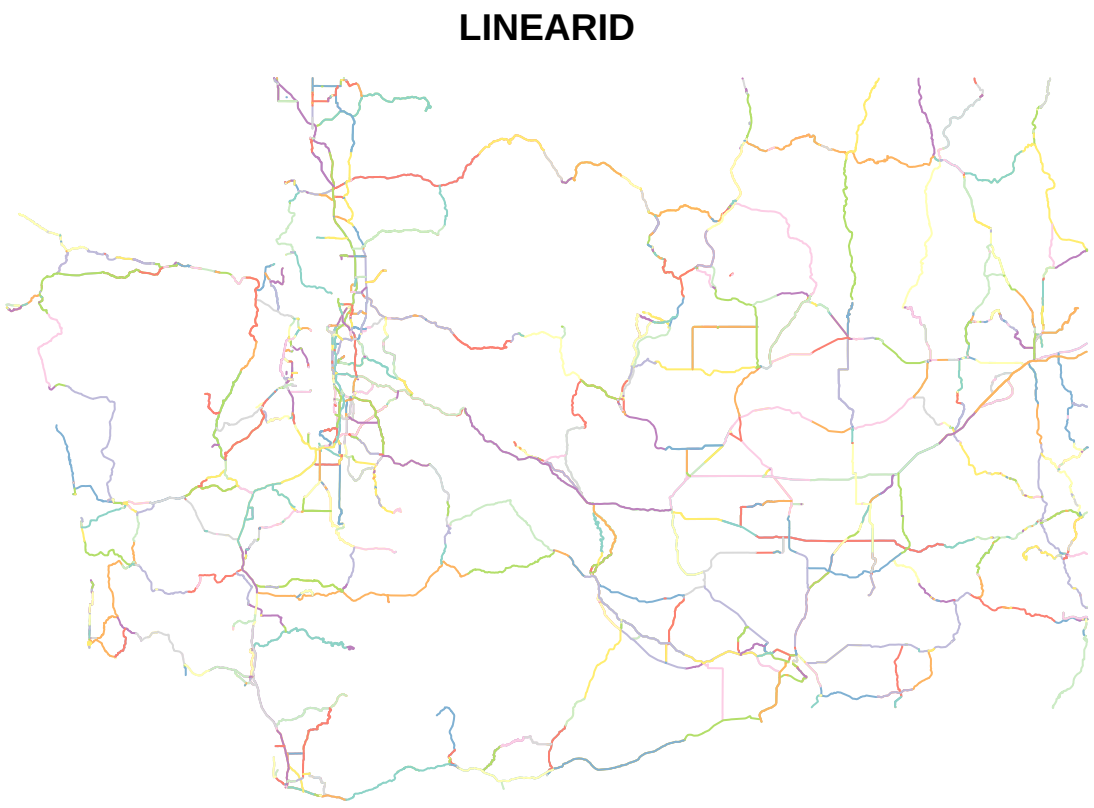
```
print(head(roads))
```

```
## Simple feature collection with 6 features and 4 fields
## Geometry type: LINESTRING
## Dimension:     XY
```

```
## Bounding box:  xmin: -124.0058 ymin: 46.32205 xmax: -119.9736 ymax: 47.845
## Geodetic CRS:  NAD83
##        LINEARID          FULLNAME RTTYP MTFCC                       geometry
## 1  110569183105 State Rte 109 Byp     S S1200 LINESTRING (-123.9242 46.98...
## 2 1102219128828     US Hwy 97 Alt     U S1200 LINESTRING (-120.0899 47.83...
## 3 1102219128829     US Hwy 97 Alt     U S1200 LINESTRING (-120.3242 47.47...
## 4 1105007792960    US Hwy 101 Alt     U S1200 LINESTRING (-124.0055 46.33...
## 5 1103730584015     US Hwy 97 Alt     U S1200 LINESTRING (-119.9736 47.84...
## 6 1105008001942    US Hwy 101 Alt     U S1200 LINESTRING (-124.0058 46.32...
```

Let's plot the dataset to see how it looks, we will only plot one of the attributes, otherwise it will plot one map for each attribute

```
plot(roads[,1])
```



Again, this dataset can be converted to a data frame, this is useful when dealing with large vector data that may be slow to manage.

```
roads.df <- as.data.frame(roads)
print(head(roads.df))
```

```
##        LINEARID          FULLNAME RTTYP MTFCC                       geometry
## 1  110569183105 State Rte 109 Byp     S S1200 LINESTRING (-123.9242 46.98...
## 2 1102219128828     US Hwy 97 Alt     U S1200 LINESTRING (-120.0899 47.83...
## 3 1102219128829     US Hwy 97 Alt     U S1200 LINESTRING (-120.3242 47.47...
```

```
## 4 1105007792960    US Hwy 101 Alt     U S1200 LINESTRING (-124.0055 46.33...
## 5 1103730584015     US Hwy 97 Alt     U S1200 LINESTRING (-119.9736 47.84...
## 6 1105008001942    US Hwy 101 Alt     U S1200 LINESTRING (-124.0058 46.32...
```

We can estimate the length of these roads, which will come in handy when estimating road density in a certain area. First we will transform to a distance-friendly projection, and then I will use the st_length() function from the sf package to estimate the length of each road.

```
roads <- st_transform(roads, crs="EPSG:5070")
roads$length <- sf::st_length(roads)
print(head(roads[,6]))
```

```
## Simple feature collection with 6 features and 1 field
## Geometry type: LINESTRING
## Dimension:     XY
## Bounding box:  xmin: -2130827 ymin: 2908853 xmax: -1789090 ymax: 2989999
## Projected CRS: NAD83 / Conus Albers
##            length                       geometry
## 1  2948.97485 [m] LINESTRING (-2103609 297704...
## 2    35.07614 [m] LINESTRING (-1797725 298956...
## 3 48720.08804 [m] LINESTRING (-1825003 295515...
## 4    74.62914 [m] LINESTRING (-2130537 290974...
## 5  9526.23736 [m] LINESTRING (-1789090 298793...
## 6   934.32284 [m] LINESTRING (-2130827 290885...
```

The unit will be automatically in meters, we can convert to numeric if we dont want the unit directly in the column using as.numeric()

```
roads$length <- as.numeric(roads$length)
print(head(roads[,6]))
```

```
## Simple feature collection with 6 features and 1 field
## Geometry type: LINESTRING
## Dimension:     XY
## Bounding box:  xmin: -2130827 ymin: 2908853 xmax: -1789090 ymax: 2989999
## Projected CRS: NAD83 / Conus Albers
##        length                       geometry
## 1  2948.97485 LINESTRING (-2103609 297704...
## 2    35.07614 LINESTRING (-1797725 298956...
## 3 48720.08804 LINESTRING (-1825003 295515...
## 4    74.62914 LINESTRING (-2130537 290974...
## 5  9526.23736 LINESTRING (-1789090 298793...
## 6   934.32284 LINESTRING (-2130827 290885...
```

I can estimate total length for each road type, or following any other attribute, for example all roads within a certain county, or any polygon, such as camera trap buffer area.

```
road_lengths <- aggregate(length ~ RTTYP, data=roads, FUN="sum")
print(road_lengths)
```

```
##   RTTYP     length
```

```
## 1     C   12589.73
## 2     I 2471012.92
## 3     M 5794528.48
## 4     O   11446.29
## 5     S 9159988.42
## 6     U 3811870.92
```

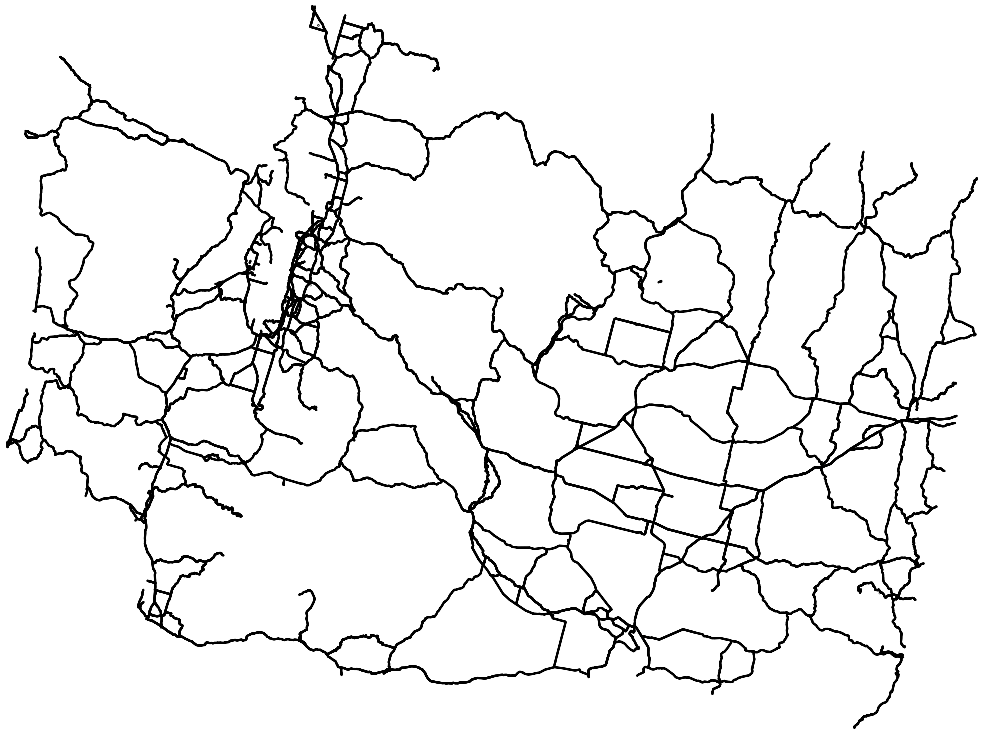
With this information I can estimate the road density of each road type within Washington state.

```
road_lengths$road_density <- ((road_lengths$length)/1e+6)/184827
print(head(road_lengths))
```

```
##   RTTYP     length road_density
## 1     C   12589.73 6.811629e-08
## 2     I 2471012.92 1.336933e-05
## 3     M 5794528.48 3.135109e-05
## 4     O   11446.29 6.192975e-08
## 5     S 9159988.42 4.955980e-05
## 6     U 3811870.92 2.062399e-05
```

Sometimes it is useful to convert lines to polygons, for example when we want a better representation of the area a linear feature occupies. This might be good for connectivity analysis as road width might define crossing probability, or for considering impervious surface generated by roads. For this we use the st_buffer() function and decide a buffer size we will use for the linear feature expansion.

```
roads_p <- sf::st_buffer(roads, dist=12) #buffer to 12 meters
plot(roads_p[,1])
```




```
print(head(roads_p)) #now our roads are a polygon map layer
```

```
## Simple feature collection with 6 features and 5 fields
## Geometry type: POLYGON
## Dimension:     XY
## Bounding box:  xmin: -2130839 ymin: 2908841 xmax: -1789078 ymax: 2990011
## Projected CRS: NAD83 / Conus Albers
##        LINEARID          FULLNAME RTTYP MTFCC                          geometry
## 1  110569183105 State Rte 109 Byp     S S1200 POLYGON ((-2103573 2977164,...
## 2 1102219128828     US Hwy 97 Alt     U S1200 POLYGON ((-1797762 2989559,...
## 3 1102219128829     US Hwy 97 Alt     U S1200 POLYGON ((-1824999 2955186,...
## 4 1105007792960    US Hwy 101 Alt     U S1200 POLYGON ((-2130535 2909763,...
## 5 1103730584015     US Hwy 97 Alt     U S1200 POLYGON ((-1792234 2988143,...
## 6 1105008001942    US Hwy 101 Alt     U S1200 POLYGON ((-2130824 2908883,...
##        length
## 1  2948.97485
## 2    35.07614
## 3 48720.08804
## 4    74.62914
## 5  9526.23736
## 6   934.32284
```

Vector data: polygons

Polygon data, sometimes also multipolygon data, are data that delimits an area, the shape of this area might represent specific physical features, such as buildings, or it might delimit an area with similar characteristics, for example residential areas or parks, or forest.

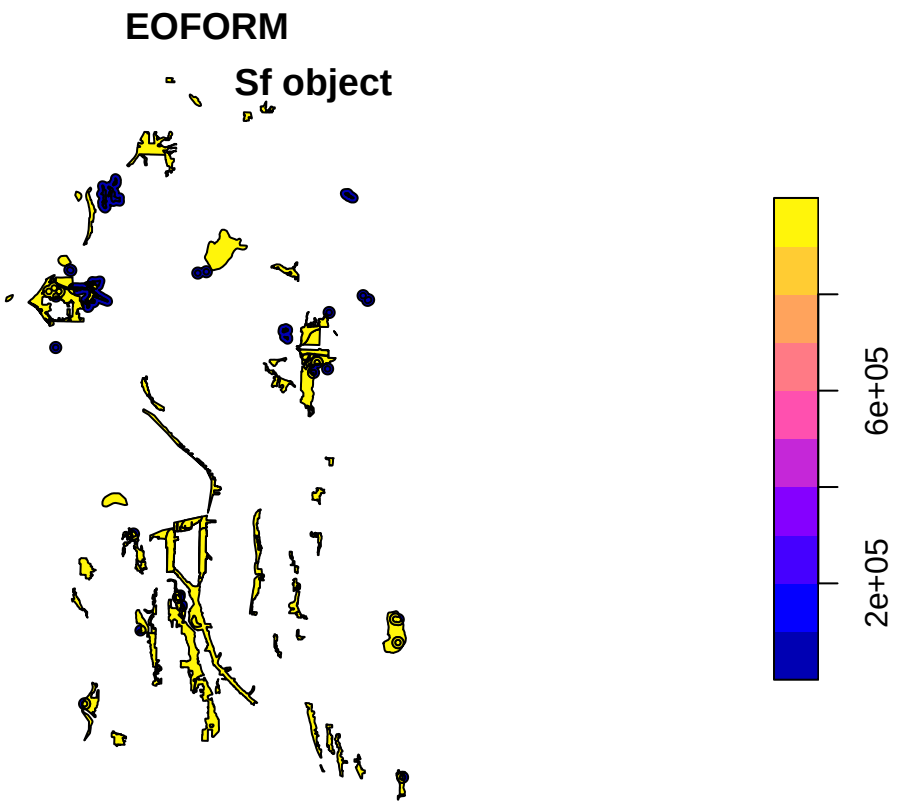
We will first load a shapefile with polygons delimiting urban wildlife habitat areas, a shapefile containing all trees in seattle, and then we will extract our own polygons from the OpenStreetMap database.

We can read a polygon dataset, like points and lines, with either the sf package or the terra package. The main difference is only the speed at which certain processes happen, but most functions are found in equivalent versions in both packages. When we plot a Spatvector from terra, we dont need to specify one attribute, it draws only one map regardless.

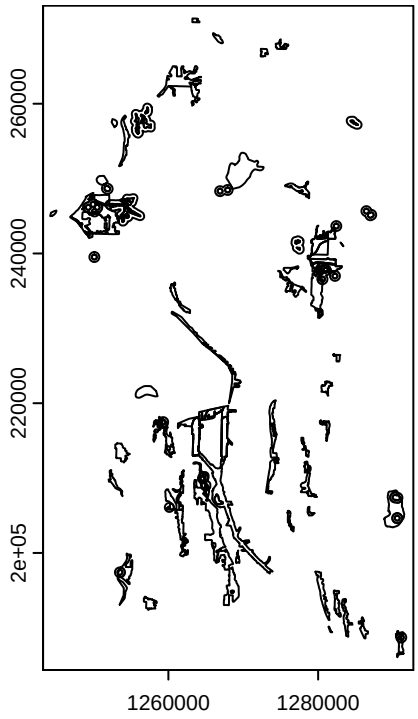
```
habitat_sf <- sf::st_read("maps/Wildlife_habitat/ECA_Wildlife_Habitat.shp")
```

```
## Reading layer 'ECA_Wildlife_Habitat' from data source
##   'C:\Users\tizge\Documents\Spatial Analysis in R Workshop\spatial_analysis_workshop_UWIN\maps\Wildl
##   using driver 'ESRI Shapefile'
## Simple feature collection with 100 features and 8 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 1244211 ymin: 186072.4 xmax: 1291765 ymax: 271387.1
## Projected CRS: NAD83(HARN) / Washington North (ftUS)
```

```
plot(habitat_sf[,1])
title("Sf object")
```



```
library(terra)
habitat <- terra::vect("maps/Wildlife_habitat/ECA_Wildlife_Habitat.shp")
plot(habitat)
```



Let's check the projection before we go forward. It is NAD83 which works well for spatial measurements <>(we should double check the projections used across the document, choose one that we will use for measurements and stick to it)

```
crs(habitat) #"EPSG:6152"
```

```
## [1] "PROJCRS[\"NAD83(HARN) / Washington North (ftUS)\",\n    BASEGEOGCRS[\"NAD83(HARN)\",\n        DA
```

We can estimate the area of each wildlife habitat, and measure the total surface area of wildlife habitat and corridors in the city.

```
library(ggplot2)
library(terra)
habitat$area <- terra::expanse(habitat) #in km2
total_ar <- sum(habitat$area/1000000)

#not knitting complaining about object being vector instead of dataframe
# ggplot(habitat, aes(fill=area/1000000))+geom_spatvector()+
#   ggtitle(paste("Seattle: Total wildlife habitat", round(total_ar), "km2")) #24km2 of wildlife habita
```

The tree dataset comes from the Seattle open data and delimits tree crowns which found with LiDAr data. I have cropped this layer to a section of seattle given the document size. We will load and look at the attributes contained within, we can also do this for sf object, by using the str() function.

```
trees <- st_read("maps/trees_seattle.shp")
```

```
## Reading layer 'trees_seattle' from data source
##   'C:\Users\tizge\Documents\Spatial Analysis in R Workshop\spatial_analysis_workshop_UWIN\maps\trees_
##   using driver 'ESRI Shapefile'
## Simple feature collection with 1000 features and 7 fields
## Geometry type: POLYGON
## Dimension:     XY
## Bounding box:  xmin: -122.3744 ymin: 47.73252 xmax: -122.363 ymax: 47.73419
## Geodetic CRS:  WGS 84
```

```
print(trees, n=3)
```

```
## Simple feature collection with 1000 features and 7 fields
## Geometry type: POLYGON
## Dimension:     XY
## Bounding box:  xmin: -122.3744 ymin: 47.73252 xmax: -122.363 ymax: 47.73419
## Geodetic CRS:  WGS 84
## First 3 features:
##   OBJECTID Hgt_Q98 Hgt_Q99   Radius       Type Shape__Are Shape__Len
## 1        1   94.06   94.89 32.39058 Coniferous   3295.562   203.5093
## 2        2   98.27   99.00 37.12950 Coniferous   4330.423   233.2838
## 3        3  116.91  120.27 31.80548 Coniferous   3177.577   199.8331
##                         geometry
## 1 POLYGON ((-122.3717 47.7326...
## 2 POLYGON ((-122.3736 47.7327...
## 3 POLYGON ((-122.3731 47.7329...
```

The dataset includes height with certain confidence levels, as it was obtained from machine learning algorithms, a tree crown radius for which we dont have a unit, and a tree type. Let's do some stats to understand the tree composition of the area we cropped.

We can count the number of trees in the area per tree type.

```
tree_numbers <- aggregate(OBJECTID  ~ Type, data=trees, FUN="length")
print(tree_numbers)
```

```
##         Type OBJECTID
## 1 Coniferous      974
## 2  Deciduous       26
```

With the same function we can get statitistics with the attributes in the dataset. For example estimate the mean and max radius and height for each tree type.

```
tree_numbers$mean_radius <- aggregate(Radius  ~ Type, data=trees, FUN="mean")[,2]
tree_numbers$max_radius <- aggregate(Radius  ~ Type, data=trees, FUN="max")[,2]
tree_numbers$mean_height <- aggregate(Hgt_Q99  ~ Type, data=trees, FUN="mean")[,2]
tree_numbers$max_height <- aggregate(Hgt_Q99  ~ Type, data=trees, FUN="max")[,2]

print(head(tree_numbers))
```

```
##         Type OBJECTID mean_radius max_radius mean_height max_height
## 1 Coniferous      974    18.23390   45.03986   108.32541     217.32
## 2  Deciduous       26    13.93828   22.48987    78.03423     109.73
```

We can do simple math using the attributes, for example use the crown radius to estimate the total tree cover in the area, pretending it was measured in inches and converting total cover to m2. We estimate the total area of each crown and then add them up.

```
trees$cover <- (pi*((trees$Radius*0.0254)^2))  #converted from inch2 to in metres2
sum(trees$cover)
```

```
## [1] 741.4985
```

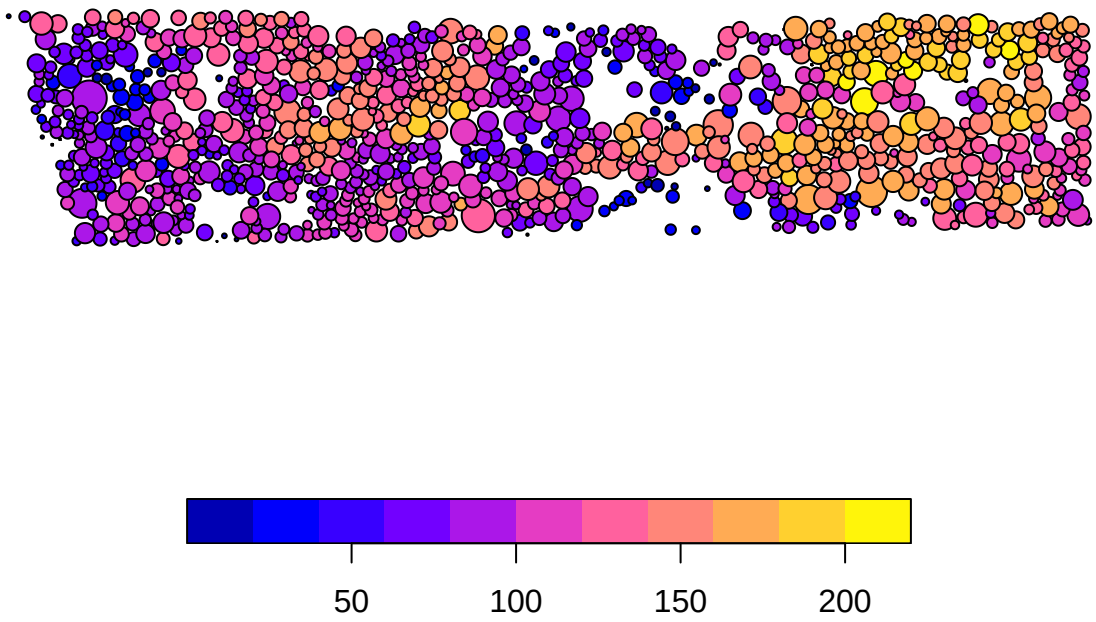
We can also add attributes to a polygon dataset as we would to a dataframe. In this case we will use the tidyterra package, to use a similar syntax we'd use with dplyr. We can also use base R for this.

For this case we will first extract all the tallest trees into an object and then use this object to attribute a rank to all the trees in the dataset. This codes works for both sf objects and spatvectors.

```
library(tidyterra)

#let's extract a set of features based on their attribute values. We can do this two ways:
tallest_trees <- trees[trees$Hgt_Q98>=150, ]
#another way:
tallest_trees<- trees[which(trees$Hgt_Q98>=150),]

## add rank of 1 to trees in the tree dataset if they are in the tallest_trees dataset, using their tre
trees  <- trees %>% mutate(rank=ifelse(OBJECTID %in% tallest_trees$OBJECTID, 1, 0))
#we can also do this with base R
trees$rank<- ifelse(trees$OBJECTID %in% tallest_trees$OBJECTID, 1, 0)
```

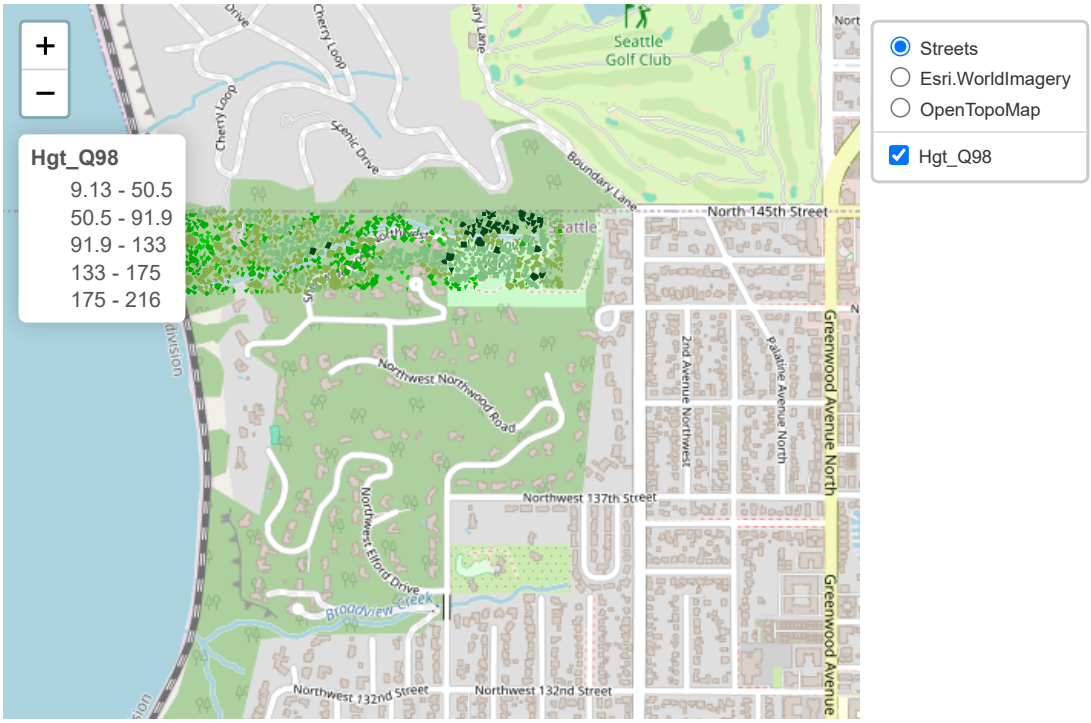
We can plot the trees following the different attributes using plot

```
plot(trees[,"Hgt_Q99"])
```



If we can also plot on a basemap as we did using ggmap in the points section, but we will use plet(), we just need to convert the sf object to a vector. plet() creates an interactive map you can explore, similar to what you would be able to do in QGIS or ArcGIS.

```
#library(leaflet)
trees_T <- vect(trees)

plet(trees_T, "Hgt_Q98", col=c("#00cd00", "#00b300", "#86B049","#7fbf7f", "#008000", "#02471a"), cex=1,
```



Finally, we can convert the polygons to points, in this case working with polygons is slower, and tree locations as points would work faster than manipulating the tree crown polygon layer.

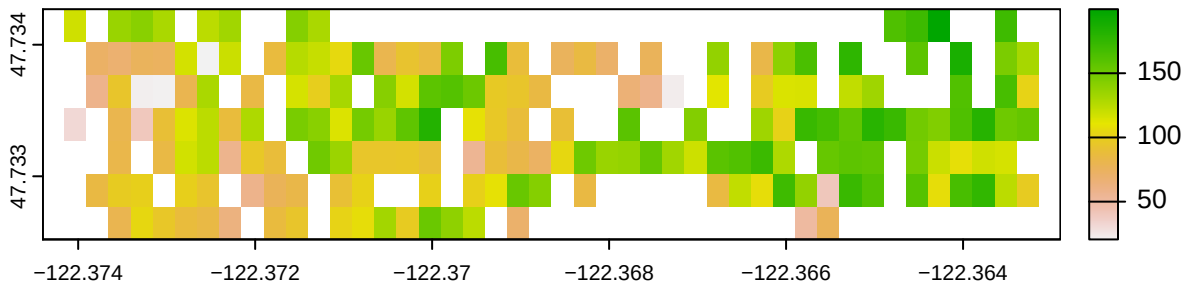
With the points layer of trees I can estimate the size of the treed area we are analyzing by generating a polygon that encloses them, estimating the size of that polygon (the treed area) and then estimate the tree density within that area.

```
trees_p <- st_centroid(trees) # we can also avoid this and just count the number of trees in total and
perimeter <- trees_p %>%
  summarise() %>%
  concaveman::concaveman(concavity = 1)
#check crs before estimating area
perimeter <- st_transform(perimeter, crs="EPSG:5070")
treed_area <- st_area(perimeter)

nrow(trees_p)/(as.numeric(treed_area)/1e+6) #in trees/km2
```

```
## [1] 9046.067
```

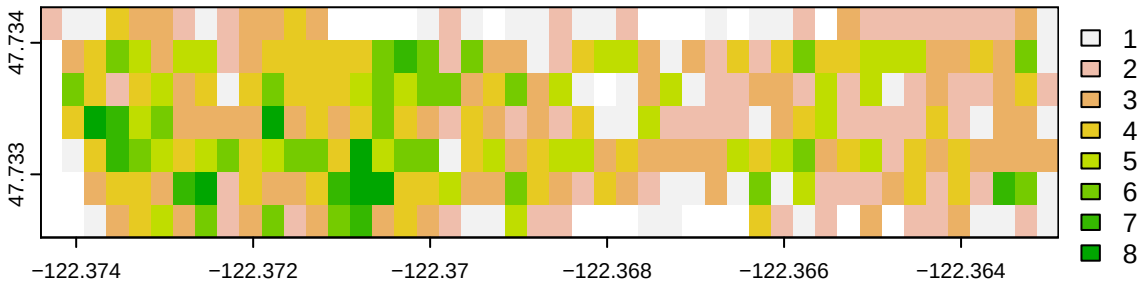
We can convert a polygon to a raster, based on an attribute. First we need a template raster that defines the resolution and the size (extent) of the raster. Then we use the rasterize() function from terra.

```
r <- rast(resolution=0.00025, extent=c(-122.3744, -122.363, 47.73252, 47.73419))
trees_height_R <- rasterize(vect(trees), r, "Hgt_Q98")
plot(trees_height_R)
```



We can use the points layer we created to generate a raster with tree counts within each cell, using the previous raster template.

```
#tree_p <- st_centroids(trees)
trees_PR <- rasterize(trees_p, r, fun=sum)
plot(trees_PR)
```



To save vector data into a shapefile we use the writeVector function from terra

```
writeVector(vect(trees), "trees_output.shp")
```

To save our output raster to a .tif file, we use the writeRaster function from terra

```
writeRaster(trees_PR, "tree_density.tif"
```

Vector data: Extracting data from OSM First we need the table with the set of validated osm keys to make sure we dont miss any urban osm features, we will use the "keys" object in the next chunk.

```
osm_kv <- read.csv("data/osm_key_values.csv")
osm_kv <- osm_kv %>% filter(!is.na(key))
keys <- unique(osm_kv$key)
```

Then we can extract the osm data for Seattle. Osm stores data both in several formats including points, lines and polygons, the ones that we are interested for this case are lines and polygons.

```
Seattle_pol <- osmextract::oe_get("Seattle",
                            layer = "multipolygons",
                            extra_tags=keys)
```

```
##   |                                                                      |
```

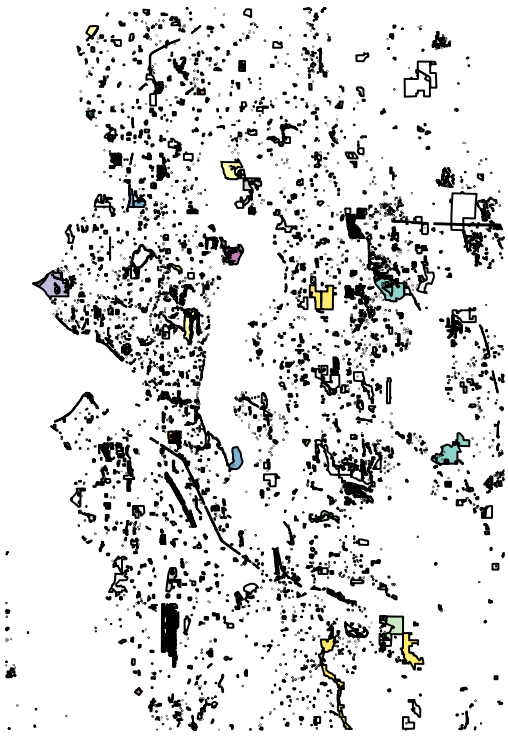
```
Seattle_lines <- osmextract::oe_get("Seattle",
                             layer = "lines",
                             extra_tags=keys)
```

```
## 0...10...20...30...40...50...60...70...80...90...100 - done.
## Reading layer 'lines' from data source
##   'C:\Users\tizge\AppData\Local\Temp\RtmpMTpB1v\bbbike_Seattle.gpkg'
##   using driver 'GPKG'
## Simple feature collection with 355001 features and 34 fields
## Geometry type: LINESTRING
## Dimension:     XY
## Bounding box:  xmin: -122.46 ymin: 47.39 xmax: -122.01 ymax: 47.83
## Geodetic CRS:  WGS 84
```

Now we can extract from the Seattle data extracted we can filter based on the land cover class or land features we are interested in. We will focus on forest areas, grass areas, and roads. Note that roads will be filtered from the lines layer.

```
forest <- Seattle_pol %>% dplyr::filter(landuse %in% c("forest")|
                                          natural  %in% c("wood")|
                                          boundary %in% c("forest", "forest_compartment"))

grass <- Seattle_pol %>% dplyr::filter(landuse %in% c("park", "grass", "cemetery", "greenfield", "recrea
                                          (!is.na(golf) & !(golf %in% c("rough","bunker"))) |
                                          amenity %in% c("park") |
                                          leisure %in% c("park", "stadium", "playground", "pitch",
                                          sport %in% c("soccer")|
                                          power %in% c("substation")|
                                          surface %in% c("grass"))

grass<-sf::st_make_valid(grass) #eliminates self intersecting multipolygons

#linear features with low traffic (Residential roads)

res_roads   <- Seattle_lines %>% dplyr::filter(highway  %in% c("residential", "rest_area", "busway"))

plot(forest[,1])
```


```
plot(grass[,1])
```




```
plot(res_roads[,1])
```

We can look at the frequency of each feature attribute within a landscape class. For example what are the diffrent types of grass areas in Seattle and how much area they occupy.

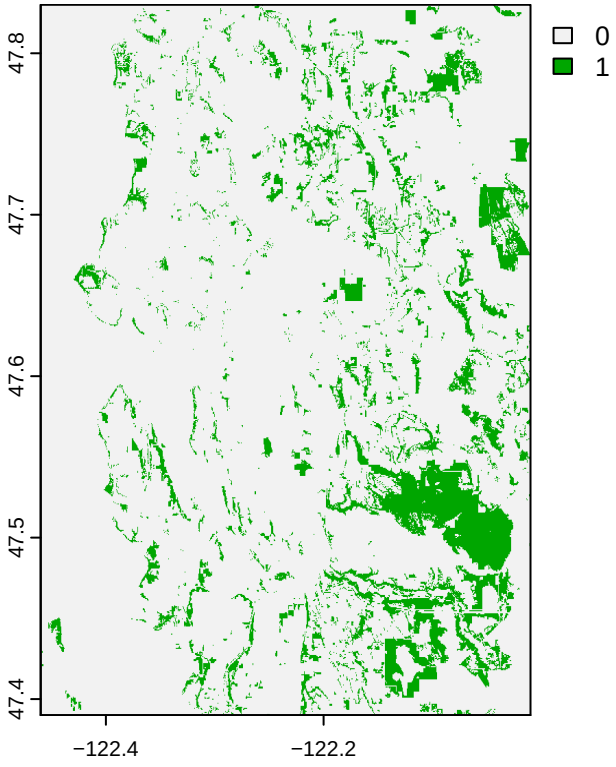
```
sf_use_s2(FALSE)
grass <- sf::st_transform(grass, crs="EPSG:5070")
grass <- grass %>% mutate(grass_area_km2 = as.numeric(st_area(grass))/1000^2)
freq_table <- aggregate(grass_area_km2 ~ leisure, data=grass, FUN="sum") #check proportion of urban gre
```

We can save the polygons we extracted from OSM as shapefiles.

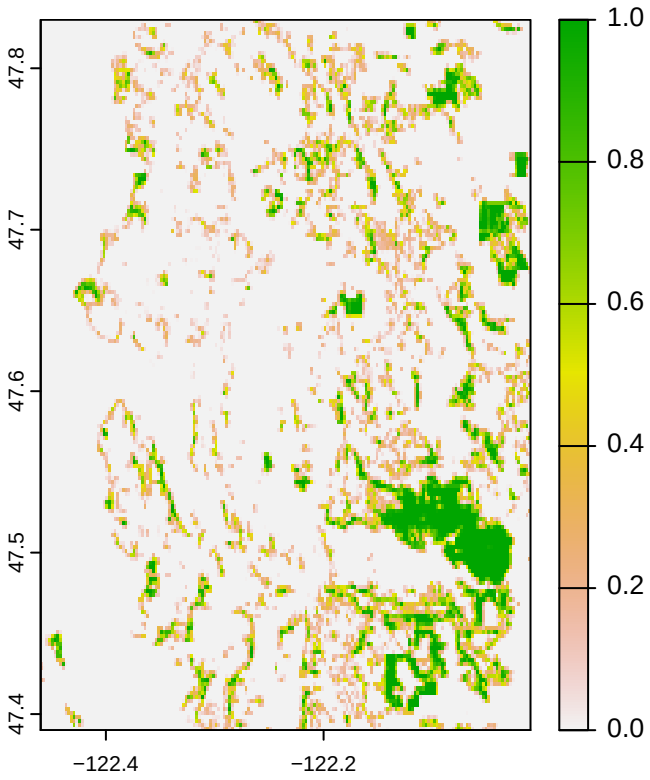
```
sf::st_write(forest, "maps/Seattle_forest.shp")
sf::st_write(grass, "maps/Seattle_grass.shp")
sf::st_write(roads_low_traffic, "maps/Seattle_roads.shp")
```

OR we can convert them as rasters, either using binary values or proportion of pixel covered by the class

```
r <- rast(resolution=0.00025, extent=c(-122.4599,-122.01,  47.39002,47.82995))
forest_r <- rasterize(forest, r, value=1)
forest_r <- ifel(is.na(forest_r),0,forest_r) #value has to be 0, not NA for the next to work
for_prop_n <- aggregate(forest_r, fact=10, fun="mean")
plot(forest_r)
```




```
plot(for_prop_n)
```



Raster data

We will focus on three raster datasets, two numerical rasters: Normalized Vegetation density index (NDVI), downloaded directly from earthexplorer, and the global human settlement building density layer GHL and one categorical raster: a land cover map from 2008 from CONUS categorizing the landscape in a developed area, and several categories for natural areas

I have previously cropped these rasters in order to keep only Washington state and speed up the computational process.

To load the rasters we use the rast() function from the terra package.

```
library(terra)
NDVI <-rast("maps/NDVI_Seattle.tif")
BUILT <- rast("maps/BUILT_Seattle.tif")
LULC <- rast("maps/LULC_Seattle.tif")
```

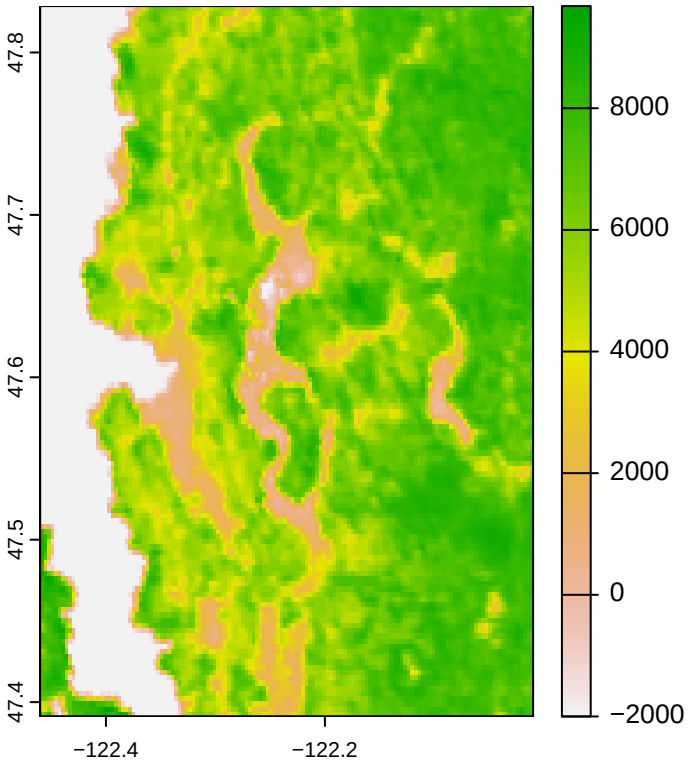
We will directly reproject the rasters to a common projection.

```
NDVI <- project(NDVI, "EPSG:4326")
BUILT <- project(BUILT, "EPSG:4326")
LULC <- project(LULC, "EPSG:4326")
```

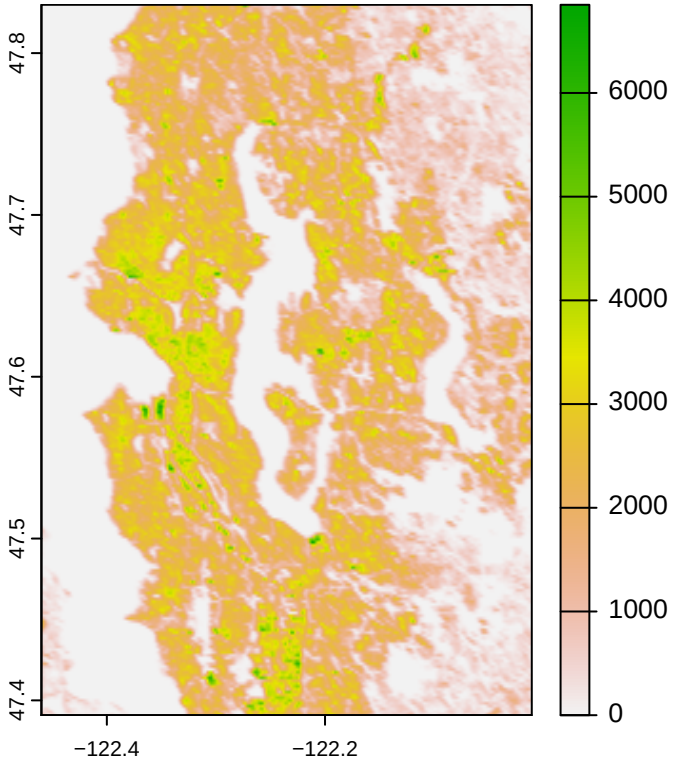
```
## |---------|---------|---------|---------|=========================================
```

We will further crop the Washington state layer to Seattle, using one of the layers we use in the previous section, as a cookie-cutter we will use the forest layer we extracted from OSM read as spatvector.

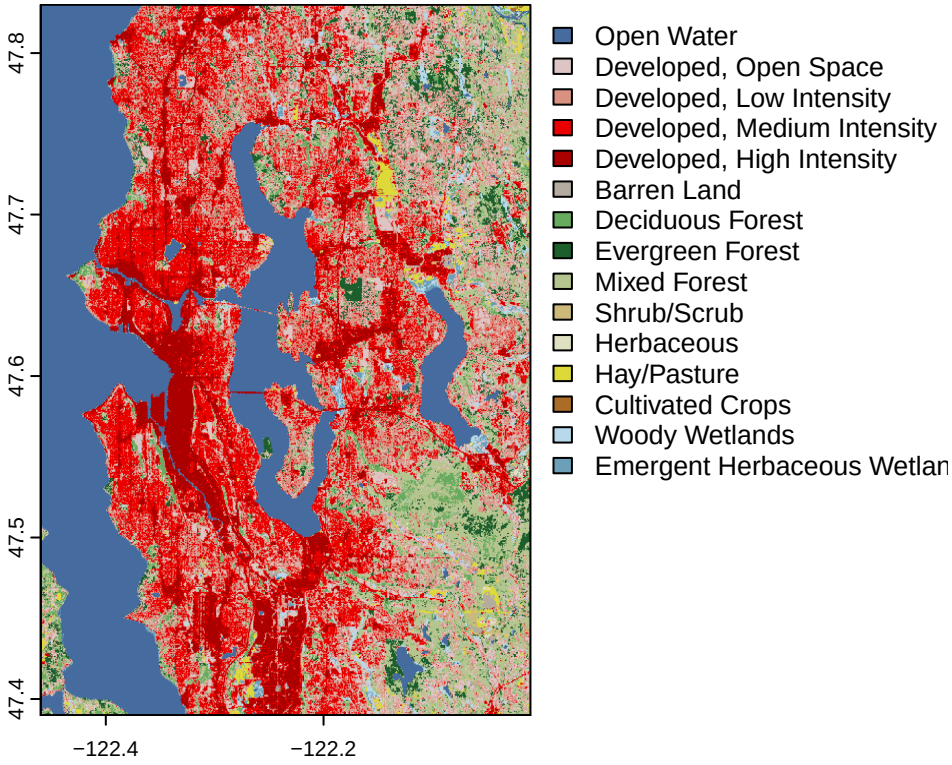
```
NDVI_c <- crop(NDVI, vect(forest))
BUILT_c <- crop(BUILT, vect(forest))
LULC_c <- crop(LULC, vect(forest))
plot(NDVI_c)
```




```
plot(BUILT_c)
```

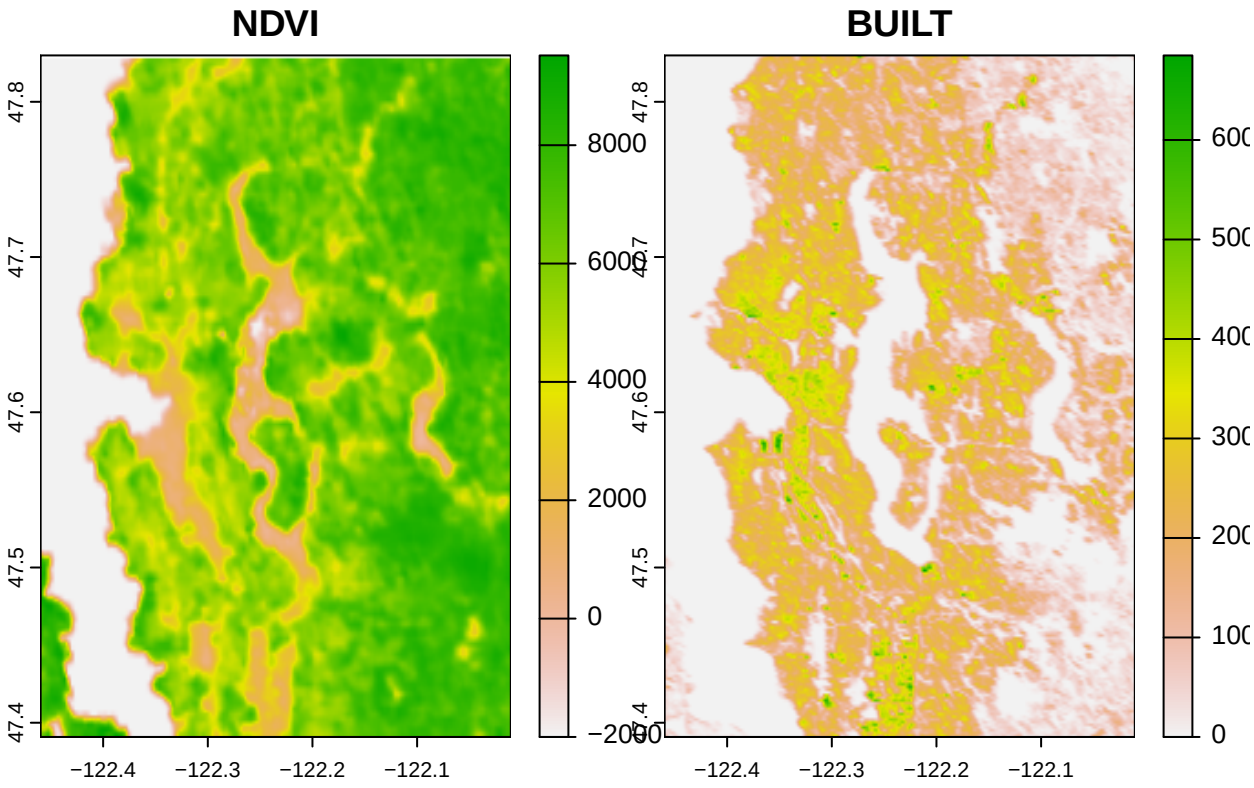



```
plot(LULC_c)
```



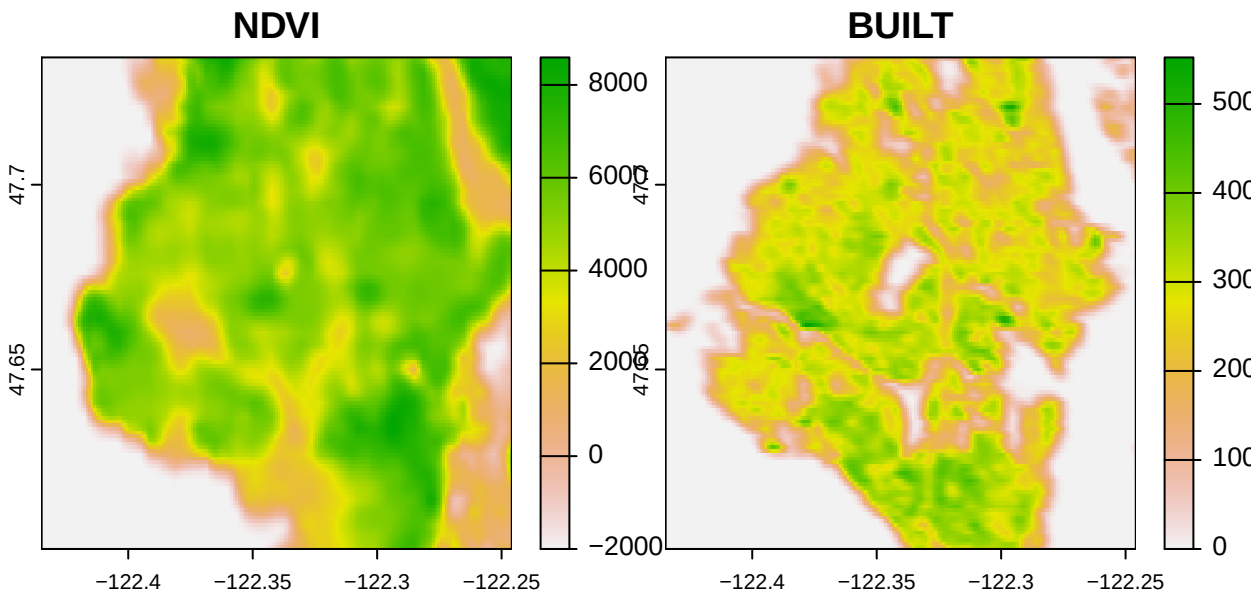
When we are dealing with many rasters, we can stack them together and apply functions directly to all of them, but first they have to perfectly match in terms of extent. So we will first project using one another as a cookie-cutter and then stack. We will stack the numerical rasters only for now and rename them.

```
NDVI_c <- project(NDVI_c, BUILT_c)

#now we can stack them
stack <- c(NDVI_c, BUILT_c)
names(stack) <- c("NDVI","BUILT")
plot(stack)
```



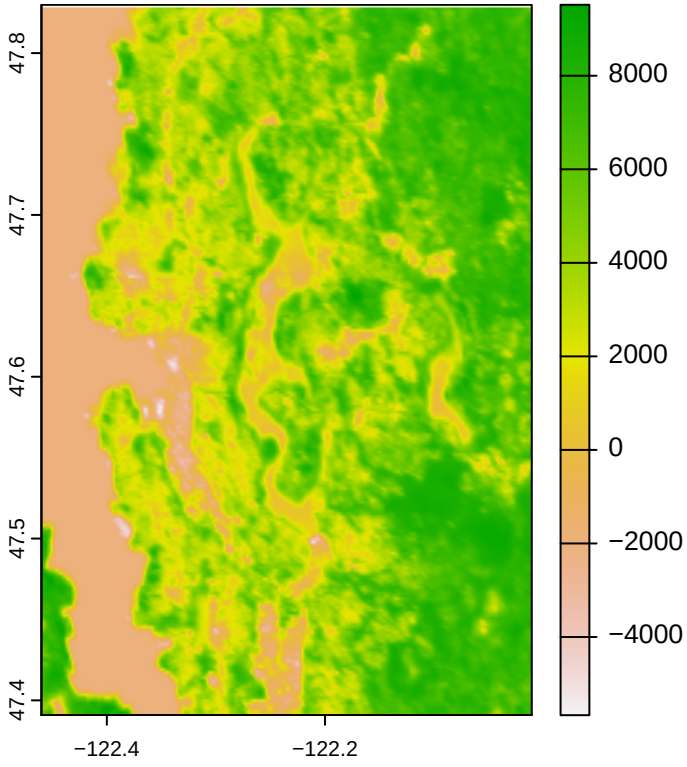
Raster layers can be resampled, to increase or decrease resolution. For this we should generate a raster template with the resolution we want. In this case we are decreasing the resolution.

```
#set template raster with resolution and extent wanted
r <- rast(resolution=0.001, extent=c(-122.4349, -122.2456, 47.60144, 47.73419 ))
#resample to the resolution in r, method can be changed
stack_r <- resample(stack, r, method="bilinear")
plot(stack_r)
```



With rasters we can also do math, the function will be applied to each cell against each overlaying cell. For example if we substract building density from Vegetation density.

```
stack_diff <- NDVI_c - BUILT_c
plot(stack_diff)
```



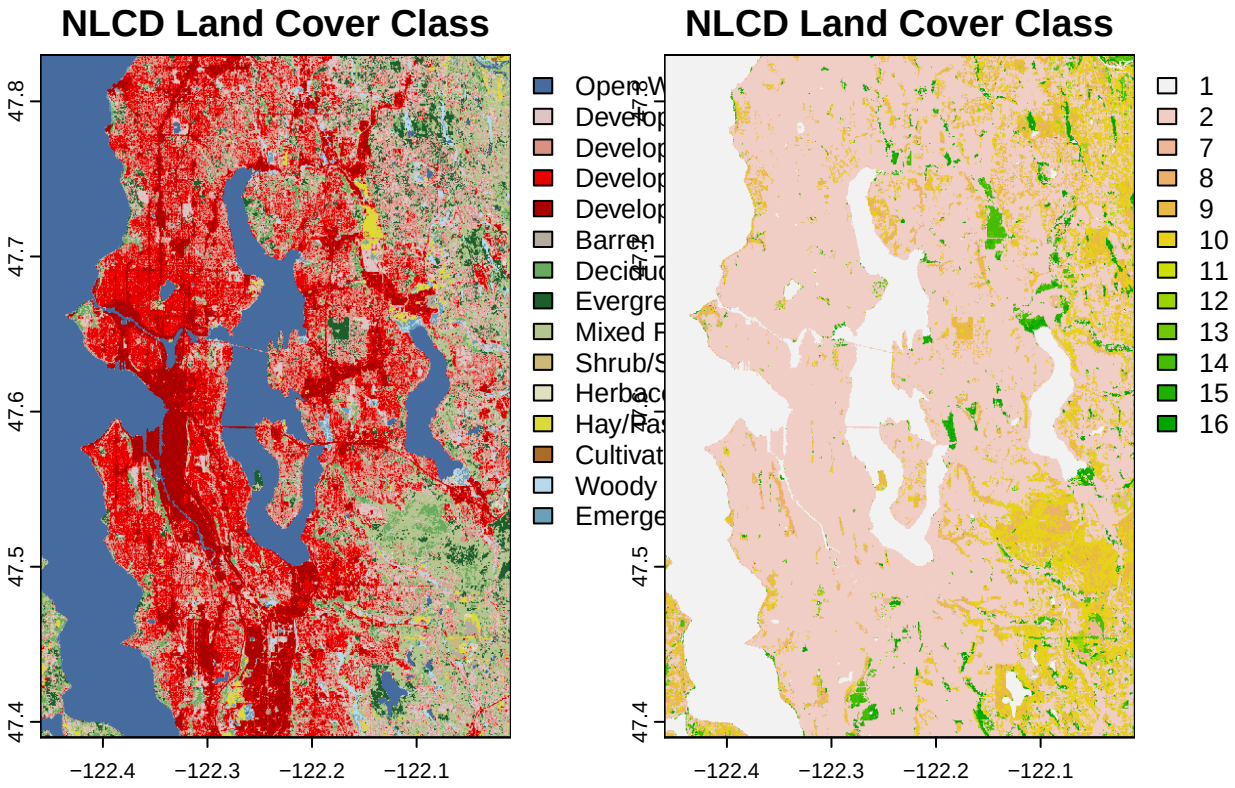
To change values across the cells following a certain logic, we reclassify the values, let's say we want to categorize a numerical layer from a continuous set of data to value ranges, or if we want to change the names or values of each category in a categorical dataset.

In this case we will reclassify the land cover map (LULC) to match the classes in another LULC map.

For this, first I create a table, or a matrix, where we have in one column the original values and in the second column the new values. If we are reclassifying from continuous numerical data to categories, then we will have three columns, the first two for the first and last number of the value range, and the third for the name or value that will indicate that value range category.

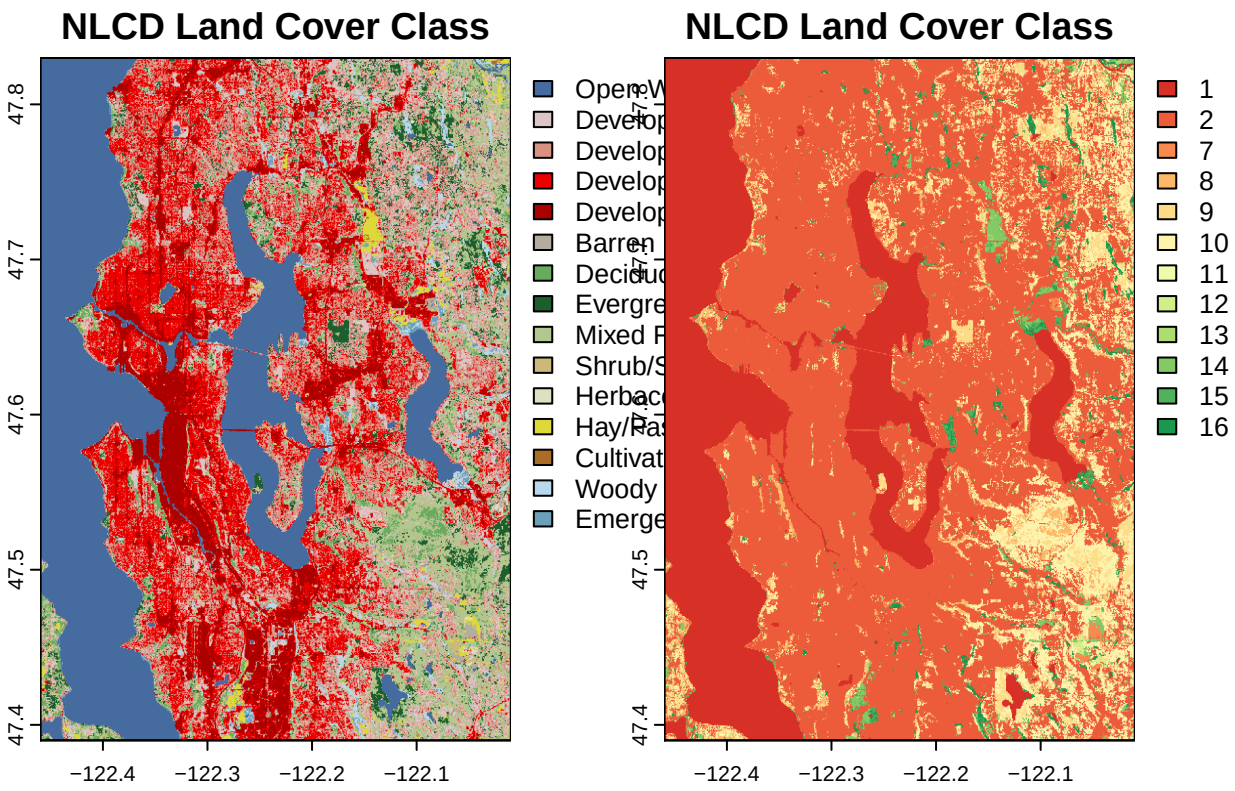
We then use the matrix of that table and the function classify() from the terra package, to reclassify our raster.

```
ctoc <- read.csv("data/reclass_conus2008toconus1938.csv")
ctoc1 <-as.matrix(ctoc[,1:2])
LULC_c_rec <- classify(LULC_c,ctoc1)
LULC_c_rec <- as.factor(LULC_c_rec)
plot(c(LULC_c, LULC_c_rec))
```

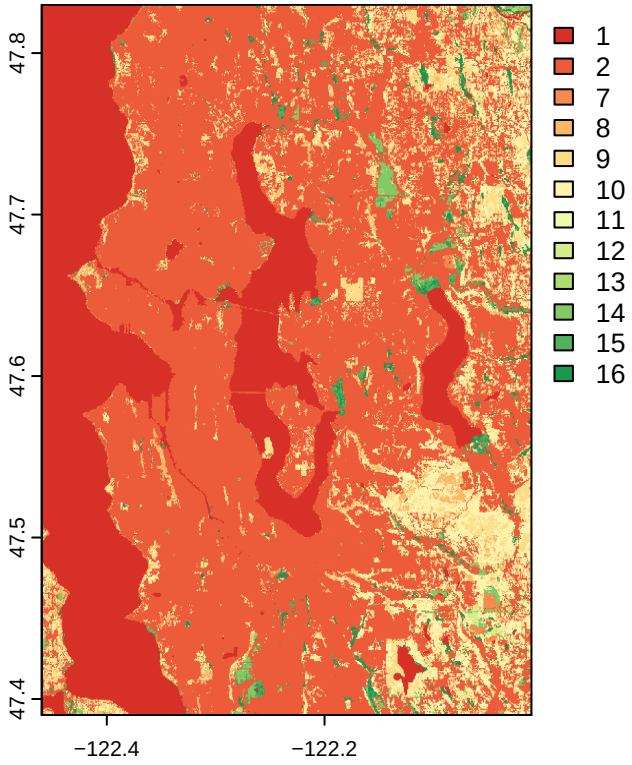


We should change the colors in the new map to make it comparable to the classes in the original map.

```
library(RColorBrewer)
cols <- brewer.pal(9, "RdYlGn")
pal <- colorRampPalette(cols)
plot(c(LULC_c, LULC_c_rec), col=pal(12))
```




```
plot(LULC_c_rec, col=pal(12))
```



For better control on the colors representing each land cover class, we can plot the new rasters in ggplot.

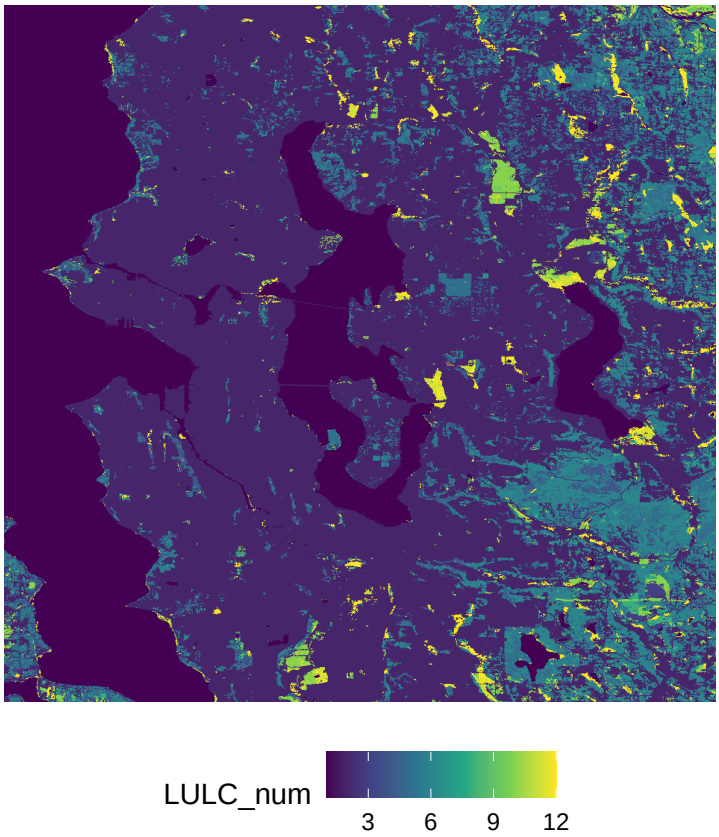
First, to plot a raster with ggplot, we convert raster to data frame and fix it a bit

```
LULC_df <- as.data.frame(LULC_c_rec, xy = TRUE)
colnames(LULC_df)<- c("x","y","LULC")
LULC_df$LULC_num <- as.numeric(LULC_df$LULC)
head(LULC_df)
```

```
##           x        y LULC LULC_num
## 1 -122.4596 47.82971    1        1
## 2 -122.4592 47.82971    1        1
## 3 -122.4588 47.82971    1        1
## 4 -122.4584 47.82971    1        1
## 5 -122.4581 47.82971    1        1
## 6 -122.4577 47.82971    1        1
```

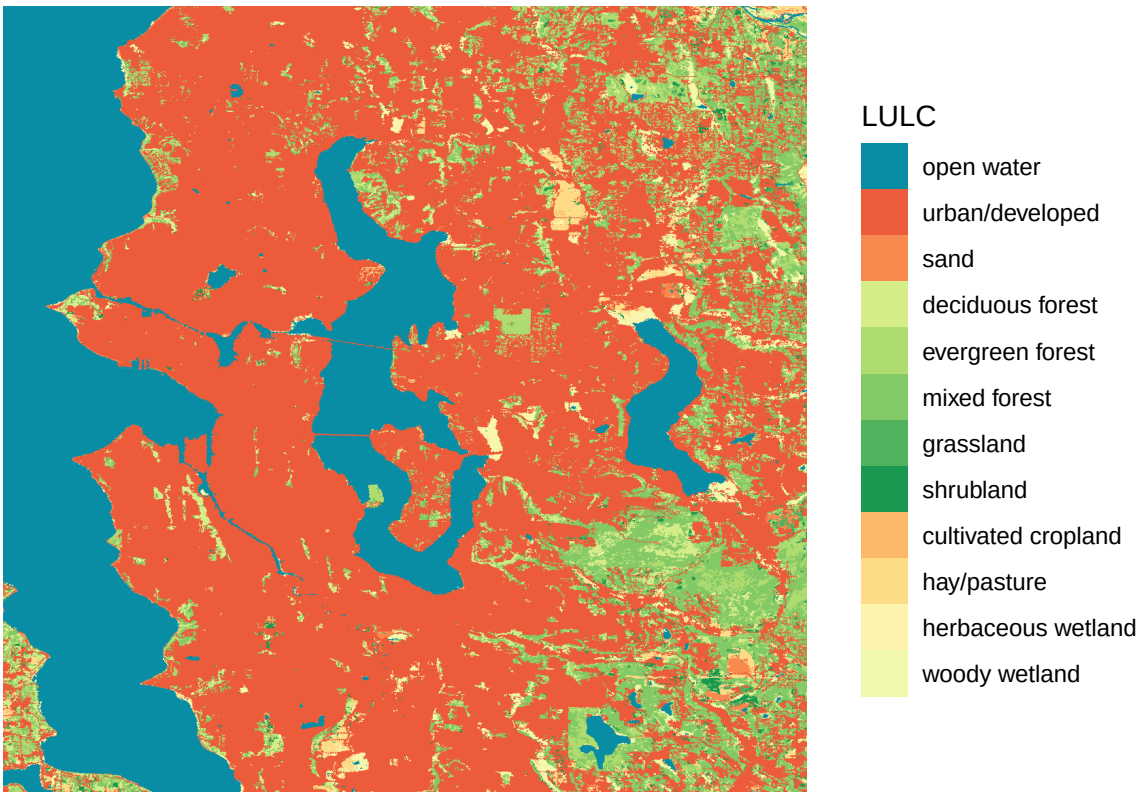
Now we can plot using ggplot, using the category value names as numerical

```
ggplot(data = LULC_df) +
  geom_raster(aes(x = x, y = y, fill = LULC_num)) +
  scale_fill_viridis_c() +
  theme_void() +
  theme(legend.position = "bottom")+
  coord_equal()
```



However, if we plot the values as categorical, we can define better the class labels and their colors.

```
ggplot(data = LULC_df) +
  geom_raster(aes(x = x, y = y, fill = LULC)) +
  scale_fill_manual(values=c("#088da5", "#EC5C3B", "#F88A50",
                             "#D4ED88","#AFDC70", "#83C966", "#51B25D", "#1A9850",
                             "#FDB768","#FDDB87", "#FEF3AC",
                             "#F1F9AC", "#D4ED88"),
                    labels=c("open water",
                             "urban/developed",
                             "sand",
                             "deciduous forest",
                             "evergreen forest",
                             "mixed forest",
                             "grassland",
                             "shrubland",
                             "cultivated cropland",
                             "hay/pasture",
                             "herbaceous wetland",
                             "woody wetland")) +
  theme_void() +
  theme(legend.position = "right")+
  coord_equal()
```



Case study: Extracting map data for camera trap analysis