Historical Persistence

Applied Economics Research Course

Bas Machielsen
Utrecht University
2023-11-13

Introduction

Introduction

- In the previous class, I have shown you an example of *vector*-format spatial data
- This means that spatial data is represented in a `data.frame`-like format with objects like polygons or lines
    - Polygons: Municipality boundaries
    - Lines: Borders of the Roman Empire
- Another way to represent spatial data is *raster* data
- Raster data can be seen much like an image, consisting of a matrix of "pixels"
    - But these pixels have specific geographic coordinates indicating their place on the world globe

Outline

- In this lecture, I demonstrate:
    - Part 1: Manipulate and join vector data
    - Part 2: Manipulate raster data
    - Part 3: How to use *vector data* when you have rastered data as basic unit of analysis
    - Part 4: How to use *raster data* when you have vector data as basic unit of analysis

Why Different Formats?

- In a research project, you must pick your **unit of analysis**
- A unit of analysis in a research project with a spatial dimension can be either a geographical unit such as a *municipality*, a *region* or a *country*
    - This naturally fits well with *vector* data, containing polygons
- However, your unit of analysis may also be a *1x1 latitude x longitude* area
    - Raster data is much more suitable for this

Preliminaries

- Both raster and vector data have various things in common
    - Both are geocoded and use a particular *CRS* (Coordinate Reference System)
    - Can have multiple variables (vector data) and layers (raster data)
- You can generally convert data from vector to raster
- Converting from raster to vector is more difficult
    - Usually what we do is use either raster data as a basis, or aggregate raster data to an already existing vector basis
    - Example: aggregate nightlights data (raster data) to the country level
    - See this paper

Part 1: Computing on Vector Data

Getting The CRS

- You have already seen you can do various things on vector data
- For example, you can ask what *CRS* it is in:

```r
netherlands <- geodata::gadm("Netherlands", level=1, path="./") |> st_as_sf()
st_crs(netherlands)[1]
```

```
## $input
## [1] "WGS 84"
```

- (You can also change that with `st_transform`)
- R-Spatial has a very good introduction to manipulating spatial data. In particular, it details how to:
    - Aggregate feature sets
    - Summarize feature sets
    - Join two feature sets based on feature geometry

Spatial Data Operations

- `sf` allows you to overlay spatial objects to find their intersections, unions, or differences.

- Common operations include:

  - **Intersection**: Finding common areas between two spatial objects. (`st_intersection`)
  - **Union**: Combining the geometries of two or more objects. (`st_union`)
  - **Difference**: Identifying the areas where one object differs from another. (`st_difference`)

- You can also use *buffering*: buffering involves creating a buffer (a zone or area) around a spatial object.

- Useful for proximity analysis and determining distances to specific features.

```r
utrecht <- netherlands |> filter(NAME_1 == "Utrecht")
buffer_around_utrecht <- st_buffer(utrecht,
                                   dist = 1000)
```

Clipping and Cropping

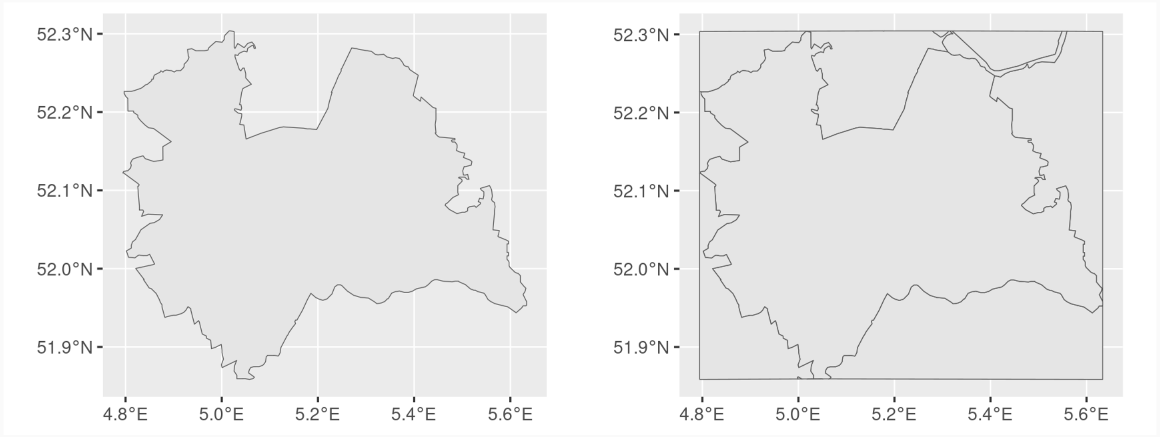
- Clipping is the process of extracting a subset of spatial data within a defined boundary.

```r
clipped_data ← st_intersection(netherlands, utrecht)
```

- Cropping is similar but keeps only the part of the data within a specific polygon.

```r
cropped_data ← st_crop(netherlands, utrecht)
```

```r
library(gridExtra)
p1 ← ggplot(clipped_data) + geom_sf(); p2 ← ggplot(cropped_data) + geom_sf()
grid.arrange(p1, p2, ncol=2)
```



Spatial Joins

- `sf` supports spatial joins to combine attribute data based on the spatial relationship.
- Types of spatial joins include:
- **Inner Join**: Keep only matching records from both datasets.

```r
inner_join_result ← st_join(data1, data2)
```

- **Left Join**: Include all records from the left dataset and matching records from the right.

```r
left_join_result ← st_join(data1, data2, left = TRUE)
```

- **Right Join**: Include all records from the right dataset and matching records from the left.

```r
right_join_result ← st_join(data1, data2, right = TRUE)
```

- **Full Join**: Include all records from both datasets.

```r
full_join_result ← st_join(data1, data2, join = st_nearest_feature)
```

Spatially Merging Two Vector Data Sets

- I will now demonstrate a very common approach to merging two vector data sets:
- Suppose you have one **province**-level dataset and one **municipality**-level dataset
- You want to merge them together, so that you retain the lowest level (municipality)
    - But you want your data to also contain provinces, so you can go back and forth
- Download the two maps using the `geodata` package
    - Convert them to `sf` format using `st_as_sf`

```r
municipalities <- geodata::gadm('Netherlands', level=2, path='./') |>
  st_as_sf() |>
  dplyr::select(NAME_2, geometry)

provinces <- geodata::gadm('Netherlands', level=1, path='./') |>
  st_as_sf() |>
  dplyr::select(NAME_1, geometry)
```

Inspecting The Data

- The datasets look as follows (I only show municipality):

```
municipalities ▷ head(5)
```

```
## Simple feature collection with 5 features and 1 field
## Geometry type: POLYGON
## Dimension:     XY
## Bounding box:  xmin: 6.223702 ymin: 52.61322 xmax: 7.041859 ymax: 53.09421
## Geodetic CRS:  WGS 84
##          NAME_2                       geometry
## 1   Aa en Hunze POLYGON ((6.569905 52.94651 ...
## 2         Assen POLYGON ((6.640786 53.02571 ...
## 3 Borger-Odoorn POLYGON ((6.745668 52.87925 ...
## 4     Coevorden POLYGON ((6.871562 52.65302 ...
## 5     De Wolden POLYGON ((6.273223 52.66813 ...
```

Spatial Joins

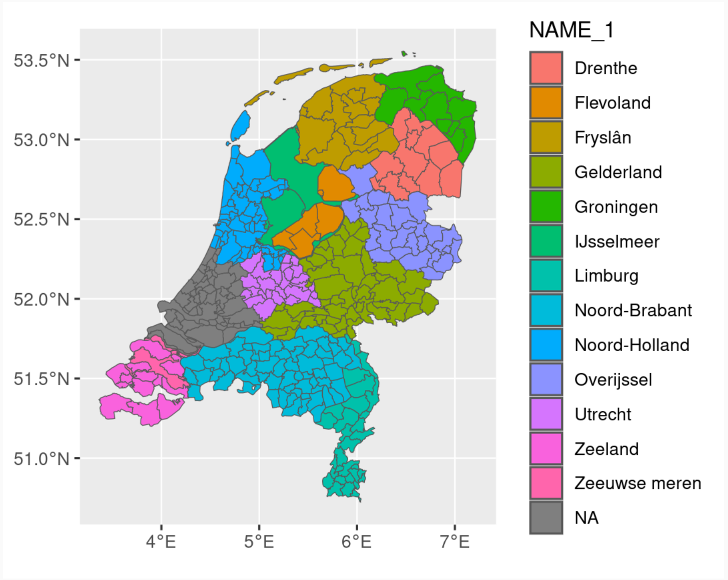
- Just like ordinary `join`'s, such as `left_join`, `full_join`, etc., you can also spatially join several datasets
- This is done using the `st_join` command:
  - You can specify various types of `join`
  - This time, we use `st_covered_by`: a municipality (the first object) should be 'covered by' a province (the second object)

```
merged_data <- st_join(municipalities, provinces, join=st_covered_by)
```

Spatial Join Plot

- We can see that we have recovered our provinces (almost) perfectly

```
merged_data ▷
  ggplot(aes(fill=NAME_1)) + geom_sf()
```



Spatial Join Plot

- ..The inaccuracy is easy to solve:

```r
merged_data ← merged_data ▷
  mutate(NAME_1 = if_else(
    is.na(NAME_1), "Zuid-Holland", NAME_1)
    )
```

- The source of the inaccuracy was the missing name for Zuid-Holland in the `provinces` spatial data.frame

Join Spatial Data

- You have already seen you can use `mutate` on a spatial data.frame, just as you would on a normal data.frame
- In fact, you can use this spatial data.frame as you would use any other data.frame
- For example, you can merge it with other data sets in exactly the way you're used to
    - Use the `cbsodataR` package:

```r
library(cbsodataR)
population <- cbsodataR::cbs_get_data('85385NED') |>
  dplyr::select(Naam_2, Inwonertal_54) |>
  mutate(Naam_2 = str_trim(Naam_2))
```

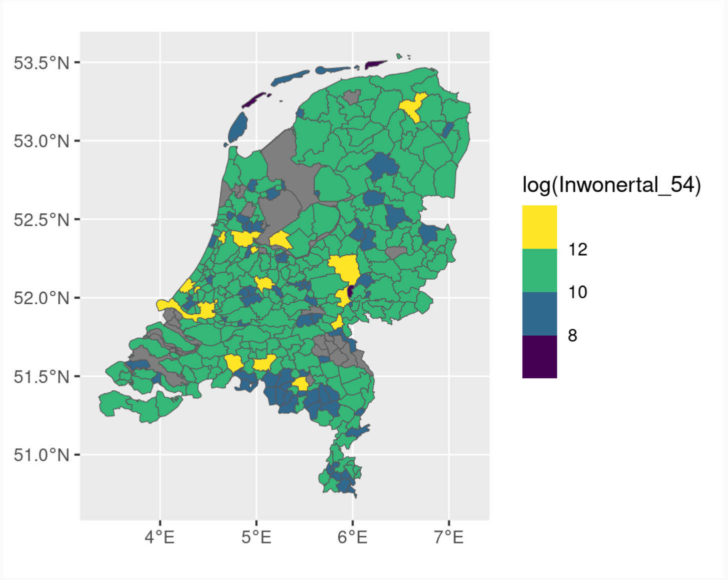
```
##
  |
  |
  |
  |==========================================================================
```

```r
data_with_pop <- left_join(merged_data,
                           population,
                           by=c("NAME_2" = "Naam_2"))
```

Plot Resulting Map

- Now, we can plot the resulting map
  - Some provinces have been omitted because of faulty matches
  - Generally, you need *identifiers* to match two data.frames without errors

```
data_with_pop ▷
  ggplot(aes(fill=log(Inwonertal_54))) +
  geom_sf() +
  scale_fill_viridis_b()
```



Part 2: Computing on Raster Data

Computing On Raster Data

- Raster data is a type of spatial data that represents information as a grid of cells.
- Each cell contains a value, often representing a measurement or attribute.

- Commonly used in fields such as economics, environmental science, and data science.

- Characteristics of raster data:
    - Regular grid structure: Data is organized in rows and columns.
    - Continuous or discrete values: Can represent continuous phenomena like temperature or discrete features like land use.
    - Spatial resolution: Grid cell size determines the level of detail in the data.

Examples of Raster Data

- Satellite imagery: Grid cells contain the RGB-values (plus other bands) of satellite images
- Climate data: Grid cells contain temperature, precipitation, and other climate variables.
- Land use data: Categorizes land parcels into different classes (e.g., urban, agriculture).
- Although more constrained than vector data, there are various things you can do with raster data:
  - Raster algebra
  - High-level functions
  - Summarizing functions

Raster Algebra

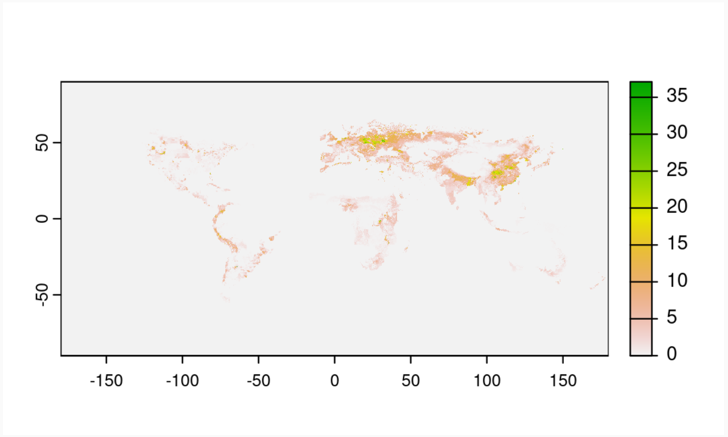
- The basic library you need for importing and changing raster data is called `raster`
- You can just change the entries of a raster object in the same way as you would change anything else:
- You can use operations like `+, -, *, /`, logical operators such as `>, >=, <, ==, !` and functions like `abs, round, ceiling, floor, trunc, sqrt, log, exp, cos, sin, atan, tan, max, min, range, prod, sum, any, all`. In these functions you can mix raster objects with numbers, as long as the first argument is a raster object.

Raster Algebra Example

- Example:
- I download potato land suitability over the entire world
    - Then I edit all the raster values by taking the square root

```r
library(raster)
potato <- geodata::crop_monfreda(crop="potato", path="./")
```

```r
sqrt_potato <- sqrt(potato)
raster::plot(sqrt_potato)
```



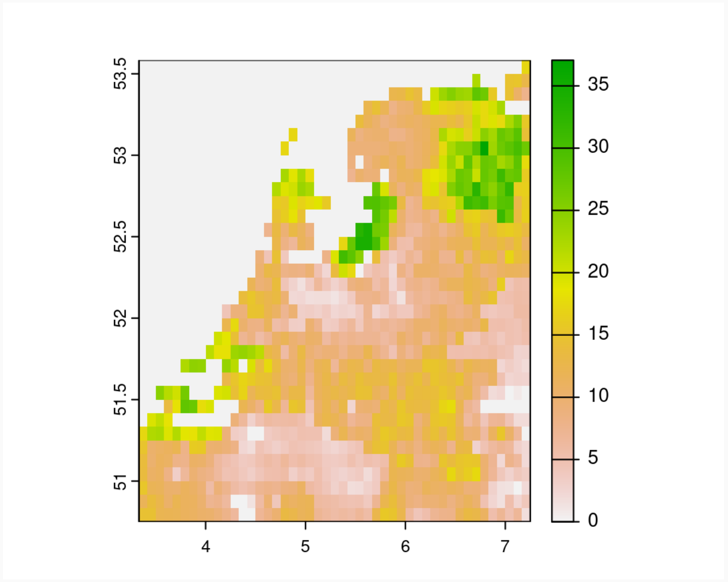
High-level Functions

- `aggregate` and `disagg` allow for changing the resolution (cell size) of a SpatRaster object.
    - In the case of `aggregate`, you need to specify a function determining what to do with the grouped cell values mean. It is possible to specify different (dis)aggregation factors in the x and y direction.
- `crop` lets you take a geographic subset of a larger raster object. You can crop a raster object by providing an extent object or another spatial object from which an extent can be extracted
    - An easy way to get an extent object is to plot a raster and then use `drawExtent` to visually determine the bounding box
- `trim` crops a raster by removing the outer rows and columns that only contain `NA` values.
    - In contrast, `extend` adds new rows and/or columns with NA values. The purpose of this could be to create a new raster with the same Extent of another, larger, raster so that they can be used together in other functions.

High-level Functions Example

- Example: I take the `sqrt_potato` raster data.frame and crop it to match the size of the Netherlands

```
crop(sqrt_potato, netherlands) |> terra::plot()
```



Part 3: Aggregate Raster to Vector

Raster Data

- We will use the MODIS *(Moderate Resolution Imaging Spectroradiometer)* satellite data source to acquire a raster data of the Netherlands
- MODIS is an instrument aboard the Terra and Aqua satellites, which orbits the entire Earth every 1-2 days, acquiring data at different spatial resolutions.
- The data acquired by MODIS describes features of the land, oceans and the atmosphere.
    - A complete list of MODIS data products can be found on the MODIS website
    - The website contains *codes* that you use to query the corresponding data
    - I use the *vegetation index products*, available here

Get Vegetation Data

- We start off by gathering the Normalized Difference Vegetation Index (NDVI) data for the Netherlands
    - It is a widely used vegetation index in remote sensing and geospatial analysis to assess and monitor the health and vitality of vegetation
    - Values close to -1 represent non-vegetated or barren surfaces, such as water bodies or urban areas, values close to 0 correspond to to bare soil, rocks, or other non-vegetated surface, and values closer to 1 represent various stages of vegetation, with higher values indicating healthier and denser vegetation.

```
library(MODIStsp)
information <- MODIStsp_get_prodlayers("M*D13Q1")

information[3]
```

```
## $bandfullnames
##  [1] "16 day NDVI average"              "16 day EVI average"               "VI q
##  [4] "Surface Reflectance Band 1"       "Surface Reflectance Band 2"       "Surf
##  [7] "Surface Reflectance Band 7"       "View zenith angle of VI pixel"    "Sun 
## [10] "Relative azimuth angle of VI pixel" "Day of year of VI pixel"        "Qual
```

Downloading Data

- Now, we send a long query to the MODIS database:

```r
source('password.R')
MODIStsp(
  gui = FALSE,
  out_folder = "./",
  out_folder_mod = "./",
  selprod = "Vegetation_Indexes_16Days_1Km (M*D13A2)",
  bandsel = "NDVI",
  user = "basm92",
  password = password_here,
  start_date = "2020.06.01",
  end_date = "2020.06.01",
  verbose = FALSE,
  spatmeth = "file",
  spafile = "gadm/netherlands.shp",
  out_format = "GTiff"
)
```

Import Data to R

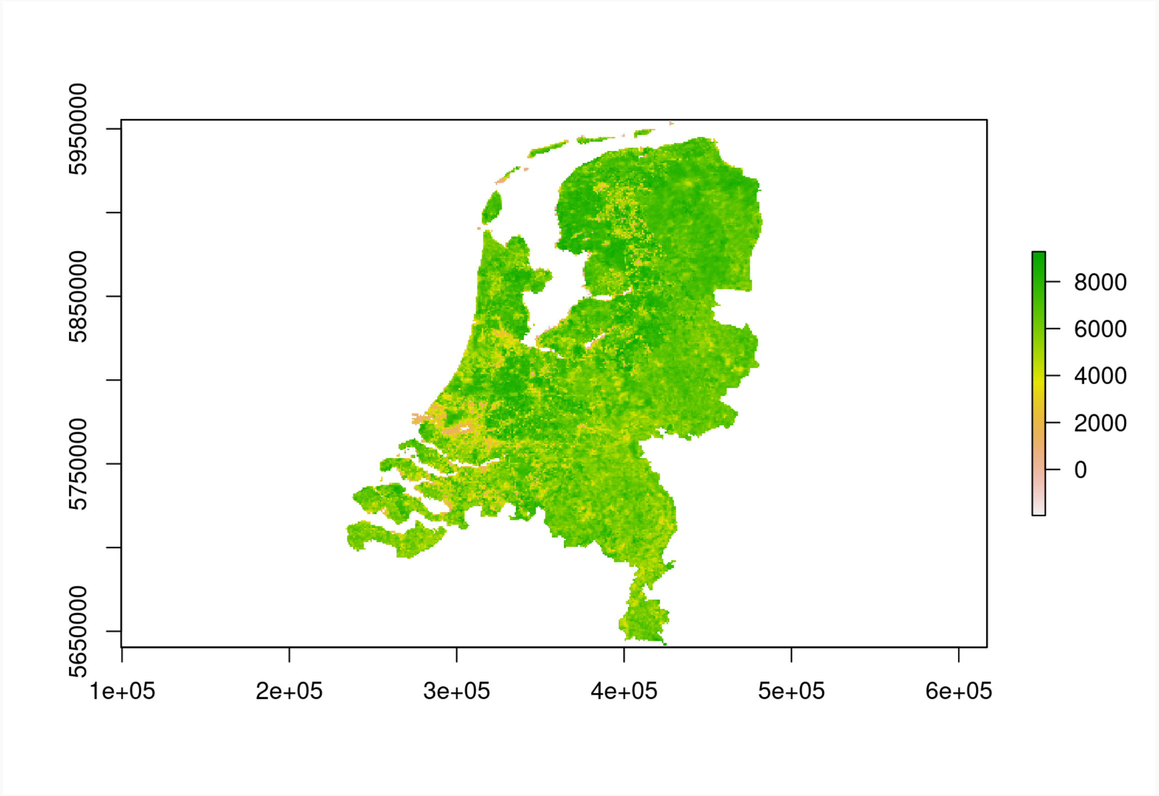
- ..And we import the data in R:

```r
library(raster); library(terra)
# Import the file
ndvi_raster ← raster('./MYD13A2.A2020153.h18v03.061.2020336102901.hdf')
# Set the CRS of the Netherlands to this file
crs_raster ← st_crs(ndvi_raster)
netherlands_transformed ← st_transform(netherlands, crs= crs_raster)
# Isolate only the relevant part overlapping the netherlands
ndvi_raster ← raster::mask(ndvi_raster, netherlands_transformed)
# Crop it
ndvi_raster ← terra::crop(ndvi_raster, netherlands_transformed)
```

Plot Output

- Our file looks like this:

```
raster::plot(ndvi_raster)
```



Main Task

- Now we can proceed to our main task: computing average vegetation per polygon in our `netherlands` shapefile
  - Or rather, let's do this by municipality!

```
# Import
netherlands_munip ← geodata::gadm("Netherlands", level=2, path="./") ▷ st_as_sf(

# Set the CRS
netherlands_munip ← st_transform(netherlands_munip, crs = crs_raster)
```

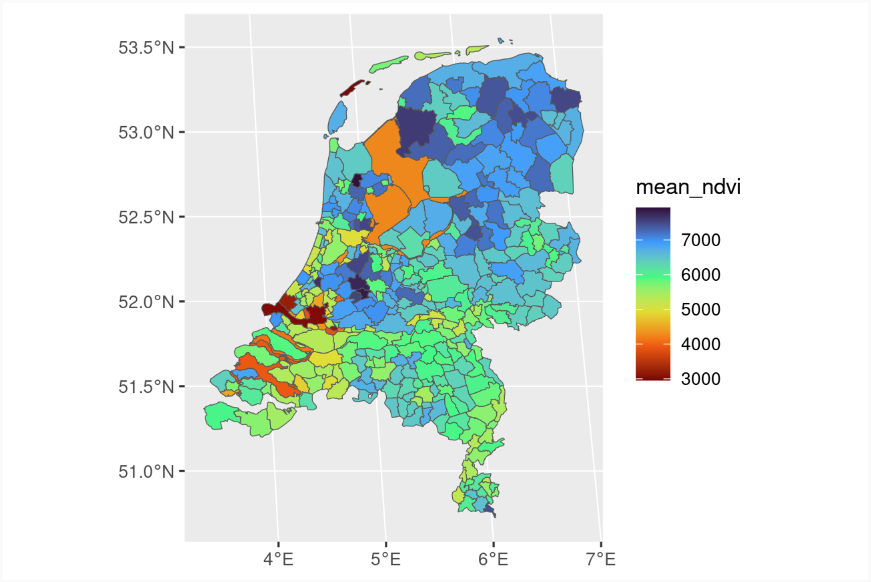
- This can be done *very easily* through `raster`'s `extract` function:
  - This extracts a list of values from the raster for each polygon in the sf data frame:
  - Followed by some data wrangling:

```
values ← extract(ndvi_raster, netherlands_munip)
netherlands_munip ← netherlands_munip ▷
  mutate(mean_ndvi = map_dbl(values, mean, na.rm=T))
```

Inspect Output

- Finally, we can plot this:
  - Some of the highly urbanized areas have a low vegetation index and some of the more rural areas have a higher vegetation index

```
netherlands_munip ▷
  ggplot(aes(fill=mean_ndvi)) +
  geom_sf() +
  scale_fill_viridis_c(direction = -1, option = 'turbo')
```



Part 4: Aggregate Vector to Raster

Download Map to Start With

- We first download the map of the Netherlands through the `giscoR` package

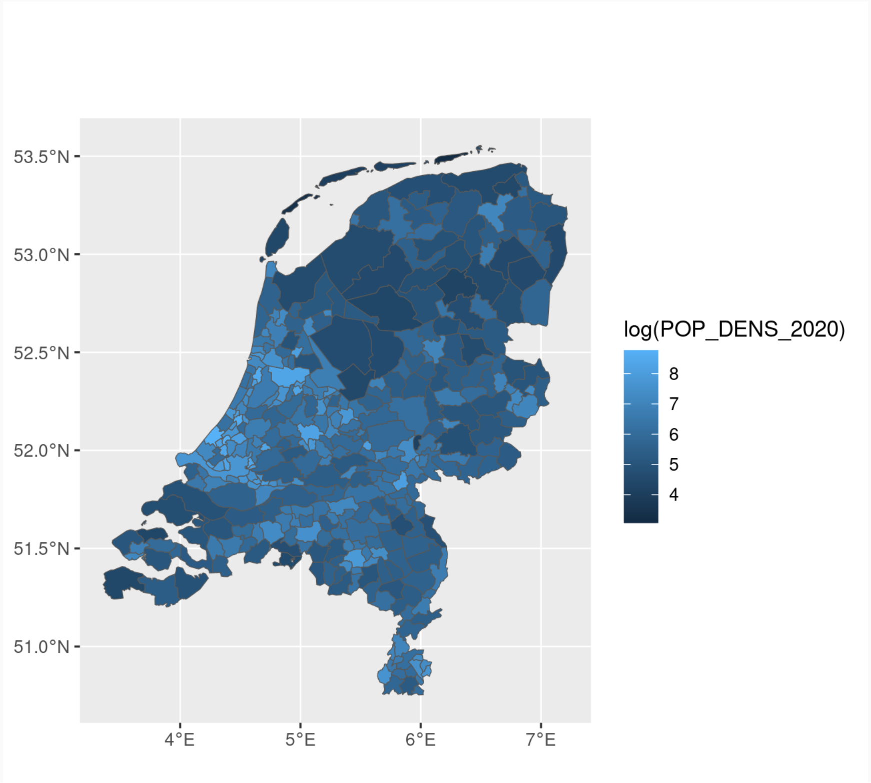
```r
netherlands <- giscoR::gisco_get_lau(year='2020', country='Netherlands')

netherlands |> head(5)
```

```
## Simple feature collection with 5 features and 10 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 3.433871 ymin: 51.20128 xmax: 5.021941 ymax: 52.3025
## Geodetic CRS:  WGS 84
##          id  GISCO_ID CNTR_CODE LAU_ID       LAU_NAME POP_2020 POP_DENS_2020
## 1 NL_GM0715 NL_GM0715        NL GM0715      Terneuzen    54438      208.5815
## 2 NL_GM0716 NL_GM0716        NL GM0716         Tholen    25758      157.1897
## 3 NL_GM0717 NL_GM0717        NL GM0717          Veere    21885      154.9336
## 4 NL_GM0718 NL_GM0718        NL GM0718     Vlissingen    44365     1259.4066
## 5 NL_GM0736 NL_GM0736        NL GM0736 De Ronde Venen    44457      380.1674
##    AREA_KM2 YEAR       FID                _ogr_geometry_
## 1 260.99149 2020 NL_GM0715 MULTIPOLYGON (((3.99667 51....
## 2 163.86569 2020 NL_GM0716 MULTIPOLYGON (((4.15222 51....
## 3 141.25409 2020 NL_GM0717 MULTIPOLYGON (((3.708876 51 ...
## 4  35.22691 2020 NL_GM0718 MULTIPOLYGON (((3.716321 51 ...
## 5 116.94059 2020 NL_GM0736 MULTIPOLYGON (((4.909582 52 ...
```

Plot The Map

- Let us have a look at `POP_DENS_2020`, the population density in 2020
    - This is but one of the variables in this spatial data.frame

```
netherlands ▷
  ggplot(aes(fill=log(POP_DENS_2020))) +
  geom_sf()
```



Make Raster of The Map

- Next, we make a grid dividing the Netherlands into areas of $x \times y$ latitude x longitude
  - Say 0.05×0.05 in this case
- The grid literally consists of pieces of area of 0.05 latitude by 0.05 longitude
  - This can be the unit of our analysis

```r
library(sf); library(stars); library(starsExtra)

grid <- st_make_grid(netherlands, square=T, cellsize=c(0.05, 0.05))
```

Aggregate Original Vector Data to Raster

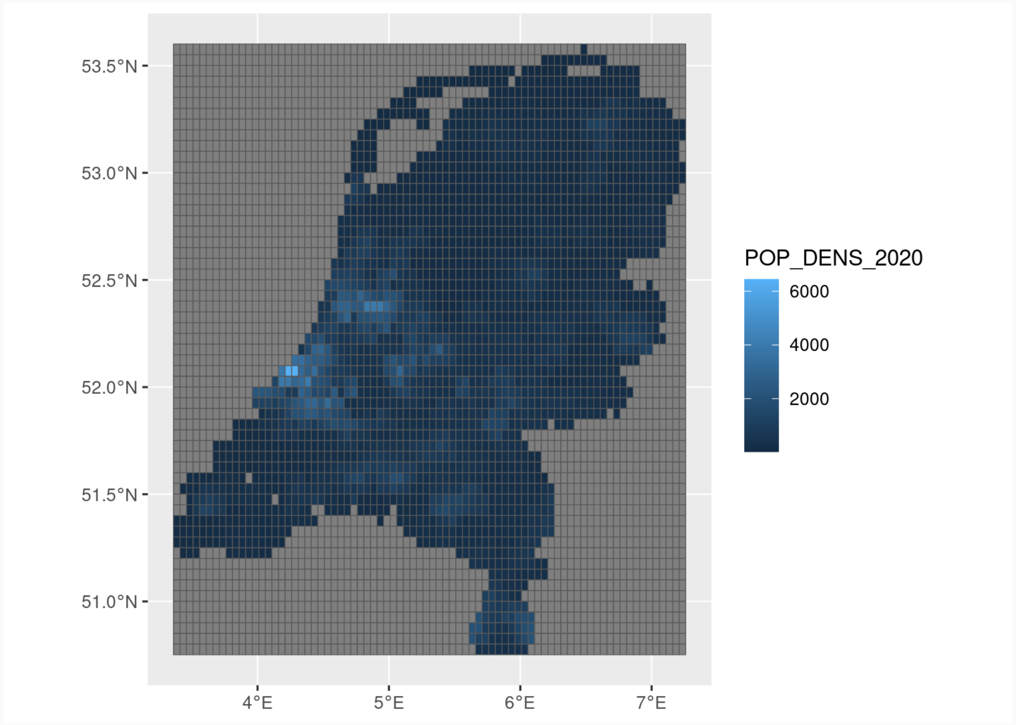
- Let us now use `raster::aggregate` to `aggregate` the variables of `netherlands` to the grid-level
- We can choose an *aggregation function*: this time, we use the mean
  - That is, for each box in the grid, we compute the corresponding `mean` values of the polygons and compute a geographical "weighted average" of our population density and other variables

```r
per_grid <- raster::aggregate(netherlands, grid, FUN=mean)
```

Look at Plot

- Let us look at the output:

```
per_grid ▷
  st_as_sf() ▷
  ggplot(aes(fill=POP_DENS_2020)) + geom_sf()
```



Extra: `geodata`

The `geodata` package

- The `geodata` package is a package through which you can get all kinds of open-sourced georeferenced data
- Install it and load it through `pacman`:

```r
p_load(geodata)
```

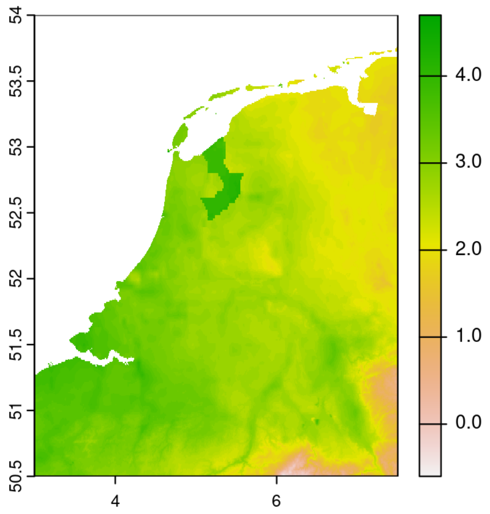
- For example, geographical data can be downloaded through:

```r
nl <- geodata::gadm("Netherlands", level = 1, path = "./") |> st_as_sf()
```

The `geodata` package

- You can also download other kinds of (raster and vector) data:
  - This can again be aggregate to our raster or vector data!

```
temperature <- geodata::worldclim_country("Netherlands",var = "tavg", path = "./")
temperature$NLD_wc2.1_30s_tavg_1 |> raster::plot()
```


54
53.5
53
52.5
52
51.5
51
50.5
4
6
4.0
3.0
2.0
1.0
0.0