Historical Persistence & Geospatial Economics

Applied Economics Research Course

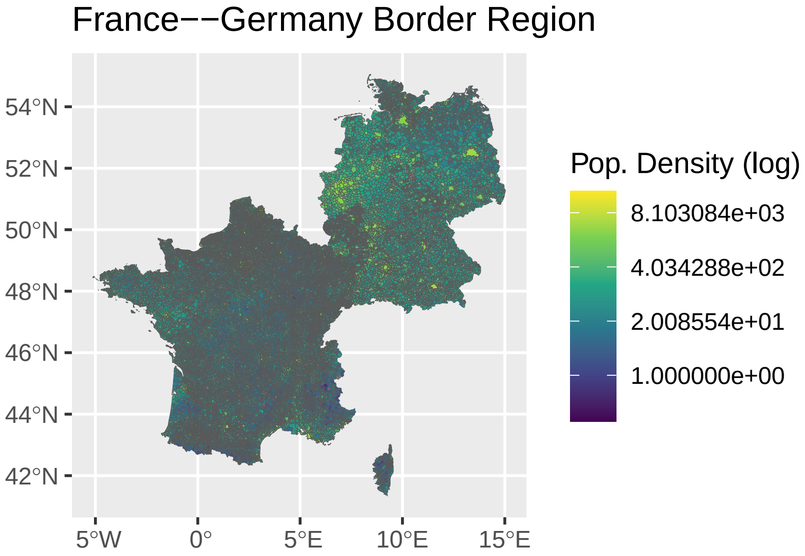
Bas Machielsen

Introduction

Why Spatial Data Wrangling?

- Your research project starts with `.geojson` / `.shp` files
- Before any regression, you need to be able to:
  - **Load** spatial data and understand its structure
  - **Clean and filter** observations
  - **Join** datasets that live in different files
  - **Aggregate** fine-grained data to your unit of analysis
- This lecture walks through all four steps using the **course example datasets**

Example Dataset



Two Data Models

Feature	**Vector**	**Raster**
Representation	Points / Lines / Polygons	Regular grid of cells
Format	`sf` data.frame	Image-like matrix
Example	Municipality boundaries	Satellite imagery, climate
R packages	`sf`	`stars`, `terra`, `raster`

- **Vector** suits analysis where your unit is a well-defined geographic entity (municipality, country)
- **Raster** suits analysis where your unit is a cell at fixed resolution ($0.05^\circ \times 0.05^\circ$)
- In practice: you often *convert* between the two

Coordinate Reference Systems

- Every spatial object has a **CRS** — a mathematical system that maps coordinates to locations on Earth
- CRS matters because:
  - Distance calculations assume a specific unit (meters vs. degrees)
  - Two datasets in different CRS produce nonsense when combined
- Use `st_crs()` to inspect, `st_transform()` to reproject

```r
st_crs(fg)$input   # inspect
```

```
[1] "WGS 84"
```

```r
fg_wgs84 <- st_transform(fg, crs = 4326)   # reproject to WGS84
st_crs(fg_wgs84)$input
```

```
[1] "EPSG:4326"
```

- Rule of thumb: always check CRS before joining two spatial datasets

Part 1: Computing on Vector Data

Loading Data

- Use `st_read()` + `here()` to load course GeoJSON files
- `here()` always resolves from the project root — no hard-coded paths

```
fg  <- st_read(here("data", "france_germany", "france_germany_updated.geoj
               quiet = TRUE)
nl  <- st_read(here("data", "netherlands", "netherlands_roman_updated.geoj
               quiet = TRUE)
```

- `class()` confirms the object is both `sf` **and** a `data.frame`:

```
class(fg)
```

```
[1] "sf"         "data.frame"
```

Inspecting sf Objects

- sf objects behave exactly like a data.frame with one extra geometry column

```
Simple feature collection with 3 features and 23 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 4.314479 ymin: 44.57289 xmax: 4.422028 ymax: 44.91308
Geodetic CRS:  WGS 84
        id GISCO_ID CNTR_CODE LAU_ID LAU_NAME POP_2016 POP_DENS_2016  AREA_KM2
1 FR_07001 FR_07001        FR  07001   Accons      385      38.49323 10.001759
2 FR_07002 FR_07002        FR  07002   Ailhon      565      71.34928  7.918791
3 FR_07003 FR_07003        FR  07003    Aizac      160      24.56040  6.514553
  YEAR      FID   GID_1 GID_0 COUNTRY               NAME_1 VARNAME_1 NL_NAME_1
1 2017 FR_07001 FRA.1_1   FRA  France Auvergne-Rhône-Alpes      <NA>      <NA>
2 2017 FR_07002 FRA.1_1   FRA  France Auvergne-Rhône-Alpes      <NA>      <NA>
3 2017 FR_07003 FRA.1_1   FRA  France Auvergne-Rhône-Alpes      <NA>      <NA>
  TYPE_1 ENGTYPE_1 CC_1 HASC_1 ISO_1 minimum_distance treatment
1 Région    Region <NA>  FR.AR  <NA>        23490.901         0
2 Région    Region <NA>  FR.AR  <NA>         4387.666         0
3 Région    Region <NA>  FR.AR  <NA>         6738.470         0
                        geometry
1 MULTIPOLYGON (((4.372145 44...
2 MULTIPOLYGON (((4.329545 44...
3 MULTIPOLYGON (((4.349086 44...
```

Inspecting sf Objects (cont.)

- Geometry-specific helpers give spatial metadata:

```
st_geometry_type(fg) |> unique()
```

```
[1] MULTIPOLYGON
18 Levels: GEOMETRY POINT LINESTRING POLYGON MULTIPOINT ... TRIANGLE
```

```
st_bbox(fg)
```

```
     xmin      ymin      xmax      ymax 
-5.140199 41.333860 15.039865 55.058121 
```

```
glimpse(fg)
```

```
Rows: 49,018
Columns: 24
$ id                <chr> "FR_07001", "FR_07002", "FR_07003", "FR_07004", "F
$ GISCO_ID          <chr> "FR_07001", "FR_07002", "FR_07003", "FR_07004", "F
```

dplyr Verbs Work as Normal

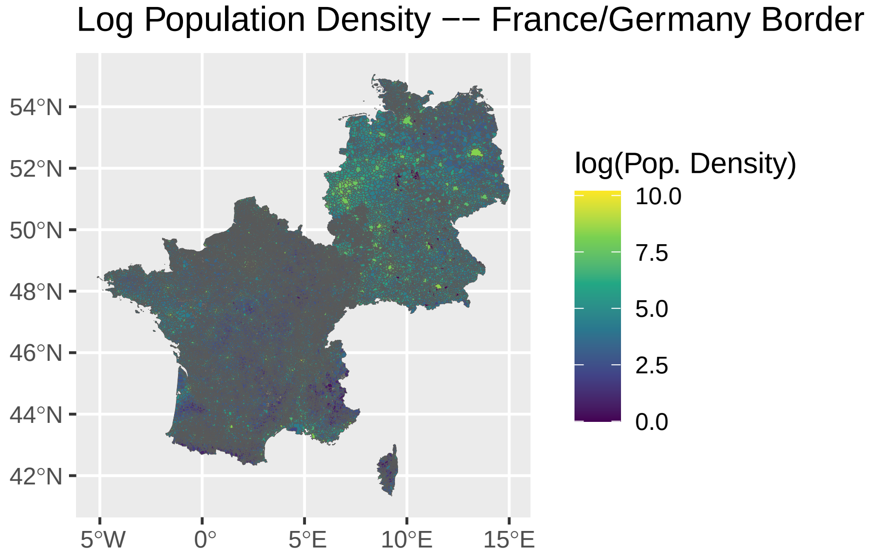
- `filter()`, `mutate()`, `select()` all work unchanged on sf objects

```
# Keep only French municipalities, add log population density
france <- fg |>
  filter(treatment == 0 | treatment == 1) |>     # all rows - just to show
  mutate(log_pop = log(POP_DENS_2016 + 1)) |>
  select(log_pop, treatment, geometry)

france |> head(3)
```

```
Simple feature collection with 3 features and 2 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 4.314479 ymin: 44.57289 xmax: 4.422028 ymax: 44.91308
Geodetic CRS:  WGS 84
   log_pop treatment                       geometry
1 3.676129         0 MULTIPOLYGON (((4.372145 44...
```

dplyr Verbs: Plot Result



CRS Operations

- Always check CRS before joining datasets:

```
st_crs(fg)$input
```

```
[1] "WGS 84"
```

```
st_crs(nl)$input
```

```
[1] "WGS 84"
```

- Reproject with `st_transform()`:
- **Warning**: mixing CRS silently produces wrong geometries and distances

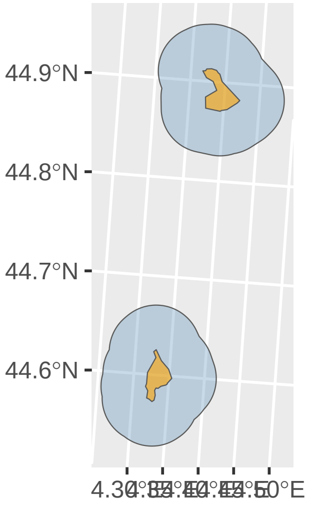
Geometric Operations

- sf supports set operations on geometries:
  - `st_buffer(x, dist)` — expand each geometry outward by `dist` metres
  - `st_union(x)` — merge all geometries into one
  - `st_intersection(x, y)` — keep the overlapping parts
  - `st_difference(x, y)` — remove the part of x that overlaps y

```r
# Small illustrative subset: two neighbouring municipalities
sub <- fg_metric |> slice(1:2)
buf <- st_buffer(sub, dist = 5000)      # 5 km buffer
uni <- st_union(sub)                    # merge the two polygons
```

Geometric Operations Plot

Buffer (blue) around two municipalit



Computing Distances

- `st_centroid()` converts polygons to their centre points
- `st_distance()` returns a matrix of pairwise distances

```
# Centroids of first 5 municipalities
cents <- fg_metric |> slice(1:5) |> st_centroid()
# Distance matrix (meters, in metric CRS)
st_distance(cents) |> round(0)
```

```
Units: [m]
      1     2     3     4     5
1     0 32560 19369 16605 40536
2 32560     0 13488 21151 21489
3 19369 13488     0 12911 28038
4 16605 21151 12911     0 23966
5 40536 21489 28038 23966     0
```

Distance to Roman Roads — Course Context

- The `france_germany_updated.geojson` already contains a distance-to-road variable
- Here is how one would build it from scratch:

```
# Pseudocode (road layer not included here)
centroids <- fg_metric |> st_centroid()
dist_matrix <- st_distance(centroids, roman_roads_metric)
fg_metric <- fg_metric |>
  mutate(min_dist = apply(dist_matrix, 1, min))
```

- The `treatment` variable in the course data is a binary indicator: `treatment == 1` if a Roman road passes through the municipality

Spatial Joins: Concept

- `st_join(x, y, join = <predicate>)` enriches `x` with attributes from `y`
    - based on a **geometric relationship** between their geometries, not a shared key

Predicate	Meaning
`st_intersects`	x and y share any point (default)
`st_within`	x lies entirely inside y
`st_covered_by`	x is covered by y (boundary included)
`st_nearest_feature`	match x to closest y

- Result: one row per match (may duplicate rows if many-to-many)
- Use `largest = TRUE` to keep only the y with largest overlap per x row

Spatial Joins: Example

- Add a NUTS-2 region label to each municipality in our course data

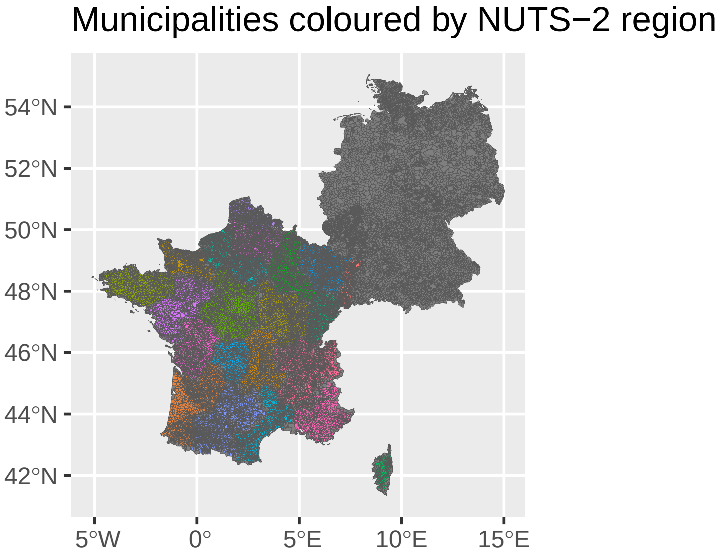
```r
# Download NUTS-2 regions for France (small file, ~200 KB)
library(giscoR)
nuts2 <- giscoR::gisco_get_nuts(
  year = "2021", nuts_level = "2",
  country = "FR", resolution = "20"
)
nuts2 <- nuts2 |> select(NUTS_ID, NAME_LATN)
```

Spatial Joins: Example

```r
fg_nuts <- st_join(
  st_transform(fg, st_crs(nuts2)),
  nuts2,
  join = st_within
)
fg_nuts |> select(NUTS_ID, NAME_LATN, POP_DENS_2016) |> head(3)
```

```
Simple feature collection with 3 features and 3 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 4.314479 ymin: 44.57289 xmax: 4.422028 ymax: 44.91308
Geodetic CRS:  WGS 84
  NUTS_ID   NAME_LATN POP_DENS_2016                       geometry
1    FRK2 Rhône-Alpes      38.49323 MULTIPOLYGON (((4.372145 44...
2    FRK2 Rhône-Alpes      71.34928 MULTIPOLYGON (((4.329545 44...
3    FRK2 Rhône-Alpes      24.56040 MULTIPOLYGON (((4.349086 44...
```

Spatial Joins: Plot



Merging with Non-Spatial Data

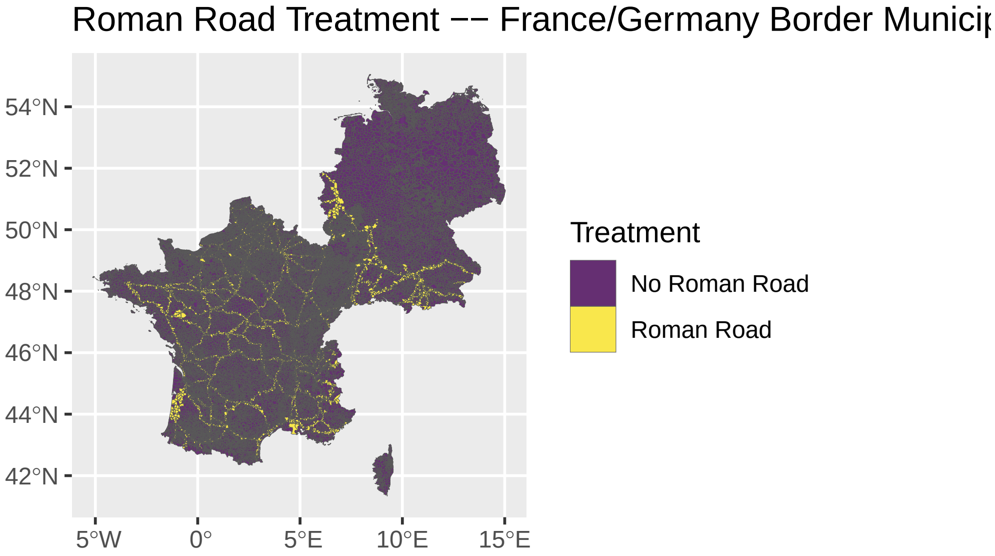
- An sf object **is** a data.frame — left_join() works exactly as expected

```r
# Simulate a CSV with per-municipality GDP data
fake_gdp <- fg |>
  st_drop_geometry() |>
  slice(1:20) |>
  select(LAU_ID) |>
  mutate(gdp_index = runif(20, 80, 130))

fg_gdp <- left_join(fg |> slice(1:20), fake_gdp, by = "LAU_ID")
fg_gdp |> select(LAU_ID, POP_DENS_2016, gdp_index) |> head(3)
```

```
Simple feature collection with 3 features and 3 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 4.314479 ymin: 44.57289 xmax: 4.422028 ymax: 44.91308
Geodetic CRS:  WGS 84
```

Putting It Together



Part 2: Computing on Raster Data

What Is Raster Data?

- A raster is a **regular grid** of cells, each containing a value
- **Resolution** = the geographic size of one cell (e.g. 100 m $\times$ 100 m)
- Each cell has implicit coordinates determined by the grid's origin + resolution + CRS

Common raster sources:

- **Satellite imagery**: RGB bands + derived indices (NDVI, NDWI)
- **Climate**: temperature, precipitation (WorldClim, ERA5)
- **Air quality**: NOx, PM2.5 concentration grids

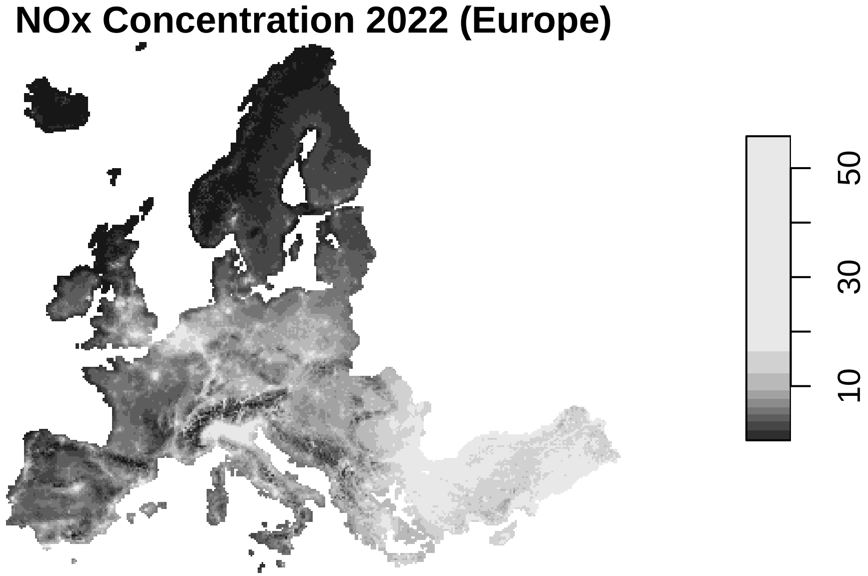
Loading Raster Data

- `stars::read_stars()` is the modern approach; `raster::raster()` is the classic

```r
nox <- read_stars(here("data", "schools", "nox_avg_22.tif"))
nox
```

```
stars object with 2 dimensions and 1 attribute
attribute(s), summary of first 1e+05 cells:
                Min. 1st Qu. Median       Mean 3rd Qu.      Max.  NA's
nox_avg_22.tif  0.05    0.05   0.05 0.05150361    0.05 0.4653052 94720
dimension(s):
  from   to  offset delta          refsys point x/y
x    1 2355 2620000  2000 ETRS_1989_LAEA FALSE [x]
y    1 2035 5440000 -2000 ETRS_1989_LAEA FALSE [y]
```

Plot Raster Data



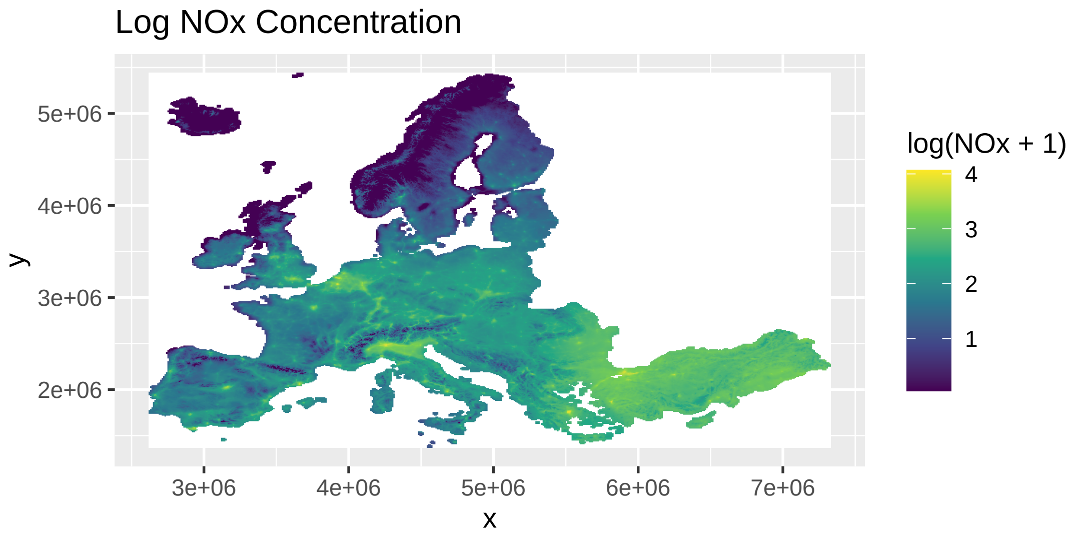
Raster Algebra

- Arithmetic works element-wise on the entire grid
- Supported: `+`, `-`, `*`, `/`, `sqrt()`, `log()`, `abs()`, logical operators

```r
# Scale NOx values (divide by 1000 to convert units) and take log
nox_log <- log(nox + 1)

# Threshold: flag cells with high NOx
nox_high <- nox > quantile(nox[[1]], 0.9, na.rm = TRUE)
```

Raster Algebra Result



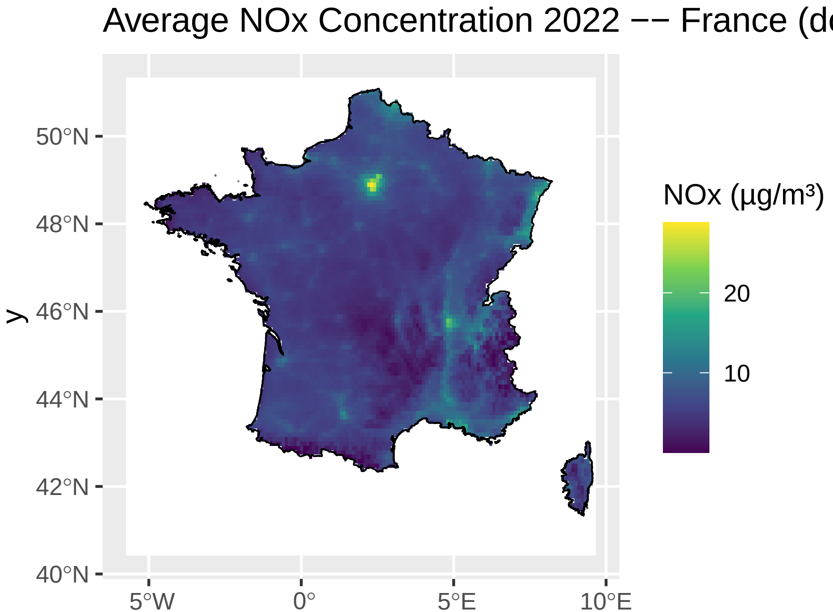
High-Level Functions

- `st_warp()` — reproject a stars raster to a new CRS or resolution
- `st_crop()` — clip to a bounding box or polygon
- `aggregate()` — reduce resolution (e.g. average 10 $\times$ 10 cells $\rightarrow$ 1 cell)

```r
# Download France outline to tempdir
france_outline <- geodata::gadm("France", level = 0, path = tempdir()) |>
  st_as_sf() |>
  st_transform(st_crs(nox))

# Crop and downsample
nox_crop   <- st_crop(nox, france_outline)
nox_coarse <- st_warp(nox_crop, cellsize = c(0.1, 0.1), crs=4326)
```

Visualizing Raster with ggplot



Using stars Idioms: Crop to Polygon

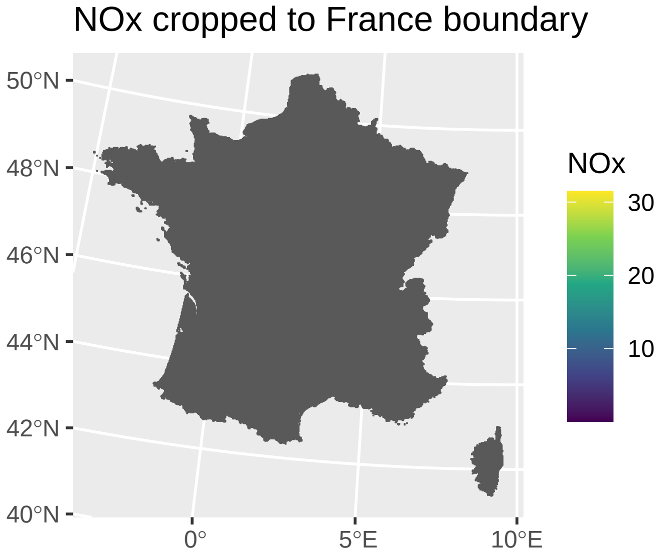
- `st_crop()` with a polygon clips the raster to its spatial extent
- Combine with `st_union()` to dissolve internal boundaries first

```r
nox_france <- nox |>
  st_transform(st_crs(france_outline)) |>
  st_crop(st_union(france_outline))

nox_france
```

```
stars object with 2 dimensions and 1 attribute
attribute(s), summary of first 1e+05 cells:
                Min. 1st Qu. Median Mean 3rd Qu. Max.  NA's
nox_avg_22.tif    NA      NA     NA  NaN      NA   NA 1e+05
dimension(s):
  from   to          refsys point                          values x/y
x    1 2355 ETRS_1989_LAEA FALSE [2355x2035] 2621000,...,7329000 [x]
y    1 2035 ETRS_1989_LAEA FALSE [2355x2035] 1371000,...,5439000 [y]
```

Crop Result


NOx cropped to France boundary
50°N
48°N
46°N
44°N
42°N
40°N
0°
5°E
10°E
NOx
30
20
10

Part 3: Aggregate Raster to Vector

The Task

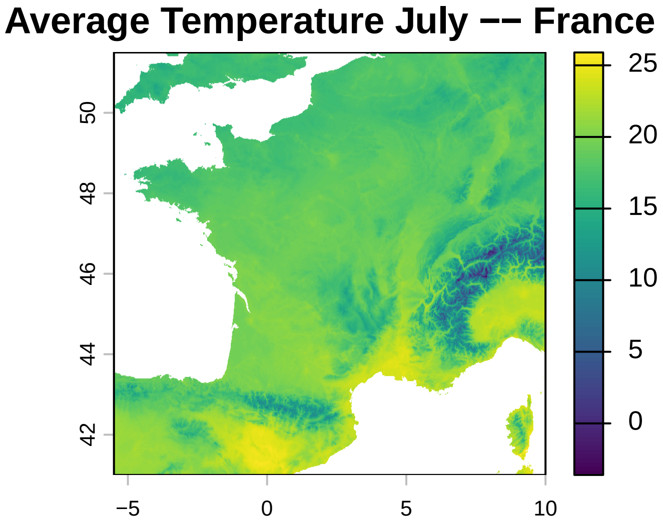
- **Goal**: Assign one climate value per municipality polygon
- **Why**: Municipalities are your unit of analysis; temperature or precipitation is a control variable stored as a raster
- **Workflow**:
    1. Load municipality polygons (vector)
    2. Download climate raster
    3. Align CRS between vector and raster
    4. Extract raster values per polygon and summarise (e.g. mean)

This is how you would add a “mean annual temperature” control to the France/Germany dataset.

Download Climate Raster

```
class       : SpatRaster
size        : 1260, 1860, 12  (nrow, ncol, nlyr)
resolution  : 0.008333333, 0.008333333  (x, y)
extent      : -5.5, 10, 41, 51.5  (xmin, xmax, ymin, ymax)
coord. ref. : lon/lat WGS 84 (EPSG:4326)
source      : FRA_wc2.1_30s_tavg.tif
names       : FRA_w~avg_1, FRA_w~avg_2, FRA_w~avg_3, FRA_w~avg_4, FRA_w~av
min values  :       -16.1,       -17.1,       -17.5,       -16.3,       -1
max values  :        11.2,        10.9,        12.6,        14.7,        1
```

Climate Plot



Align CRS

- Before extraction, **both objects must share a CRS**
- Common mistake: skip this step $\rightarrow$ values assigned to wrong polygons

```r
# Check CRS
crs(tavg_rast)                # terra SpatRaster
```

```
[1] "GEOGCRS[\"WGS 84\",\n    DATUM[\"World Geodetic System 1984\",\n
```

```r
st_crs(fg)$input              # sf data.frame
```

```
[1] "WGS 84"
```

```r
# Transform vector to match raster CRS
fg_proj <- st_transform(fg, crs = crs(tavg_rast))

# Verify they match
identical(st_crs(fg_proj)$proj4string,
          as.character(crs(tavg_rast, proj = TRUE)))
```

Using terra::extract()

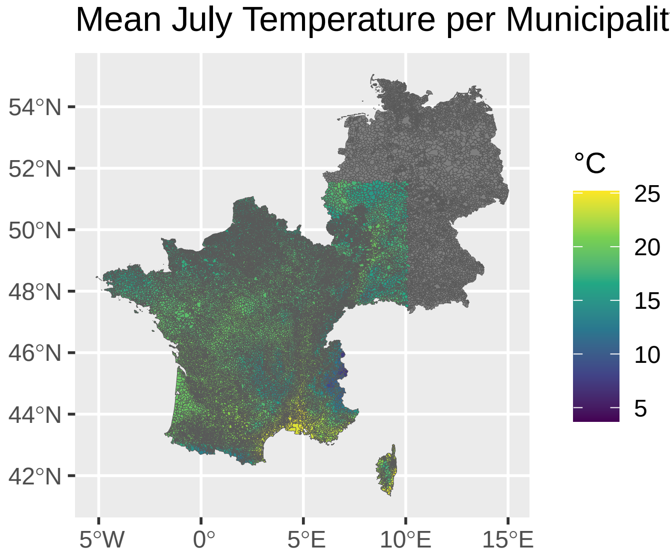
```r
# Extract mean July temperature per municipality polygon
values_july <- terra::extract(
  tavg_rast[[7]],         # raster layer
  vect(fg_proj),          # sf → SpatVector for terra
  fun = mean,
  na.rm = TRUE
)

fg_temp <- fg_proj |>
  mutate(mean_temp_july = values_july[, 2])

fg_temp |> select(mean_temp_july) |> head(4)
```

```
Simple feature collection with 4 features and 1 field
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 4.314479 ymin: 44.57289 xmax: 4.520519 ymax: 44.91308
```

`terra` Plot



Mean July Temperature per Municipality -- France/Germany
54°N
52°N
50°N
48°N
46°N
44°N
42°N
5°W
0°
5°E
10°E
15°E
°C
25
20
15
10
5

Using stars::aggregate()

- Alternative: convert raster to stars, then aggregate with sf polygons

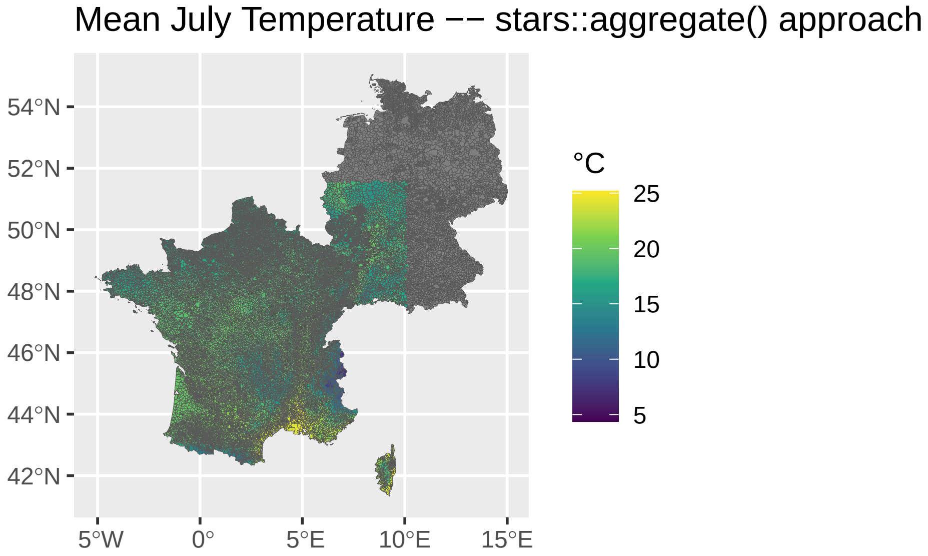
```r
tavg_stars <- st_as_stars(tavg_rast[[7]])
tavg_stars <- st_transform(tavg_stars, st_crs(fg))

agg_result <- stats::aggregate(tavg_stars, fg, FUN = mean, na.rm = TRUE) |
  st_as_sf() |>
  rename(mean_temp = 1)    # rename auto-generated layer column

agg_result |> head(3)
```

```
Simple feature collection with 3 features and 1 field
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 4.314479 ymin: 44.57289 xmax: 4.422028 ymax: 44.91308
Geodetic CRS:  WGS 84
  mean_temp                             geometry
```

stars Plot


Mean July Temperature -- stars::aggregate() approach
54°N
52°N
50°N
48°N
46°N
44°N
42°N
5°W
0°
5°E
10°E
15°E
°C
25
20
15
10
5

Part 4: Aggregate Vector to Raster

The Task

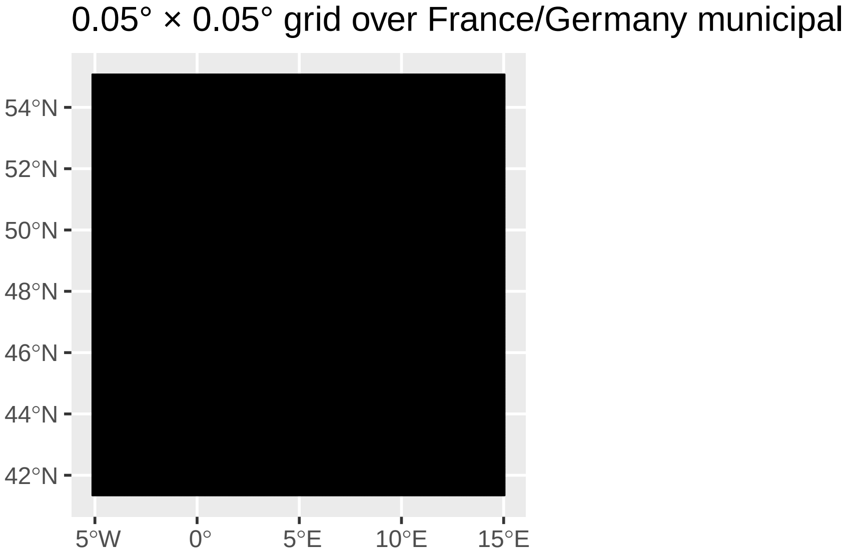
- **Goal**: Replace irregular municipality polygons with a **regular spatial grid**
- **Why**: Some research designs use grid cells as the unit of analysis
  - Easier to merge datasets from different sources
  - Avoids Modifiable Areal Unit Problem (MAUP) from administrative boundaries
- **Workflow**:
  1. Create a regular grid over the study area
  2. Assign polygon attributes to overlapping grid cells (spatial join / aggregate)
  3. Plot and verify

Creating a Grid

- `st_make_grid()` creates a grid of square (or hexagonal) cells

```
Simple feature collection with 3 features and 1 field
Geometry type: POLYGON
Dimension:     XY
Bounding box:  xmin: -5.140199 ymin: 41.33386 xmax: -4.990199 ymax: 41.383
Geodetic CRS:  WGS 84
                               x cell_id
1 POLYGON ((-5.140199 41.3338...       1
2 POLYGON ((-5.090199 41.3338...       2
3 POLYGON ((-5.040199 41.3338...       3
```

Grid Plot


0.05° × 0.05° grid over France/Germany municipal
54°N
52°N
50°N
48°N
46°N
44°N
42°N
5°W
0°
5°E
10°E
15°E

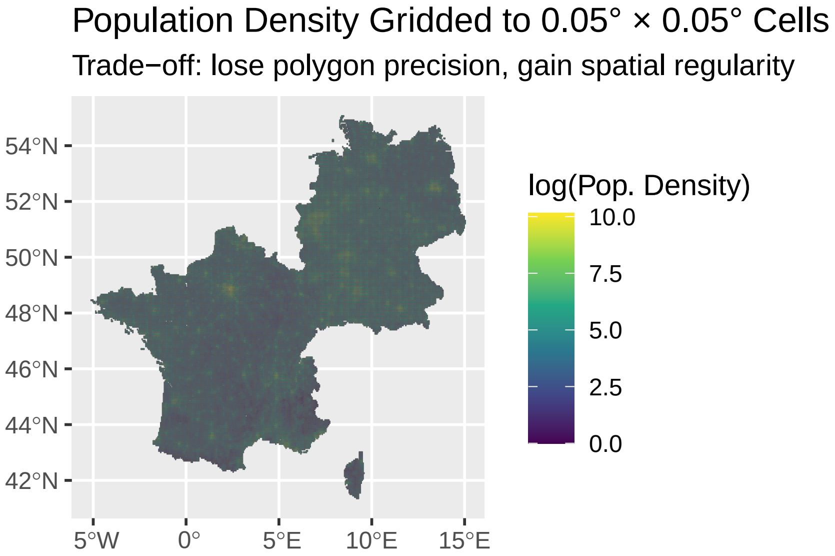
Aggregating to Grid Cells

- Spatial join: for each grid cell, find which municipalities overlap, then average

```
# Spatial join: attach municipality attributes to intersecting grid cells
fg_grid_join <- st_join(grid, fg |> select(POP_DENS_2016, treatment))

# Average POP_DENS_2016 per grid cell (some cells overlap multiple municipa
fg_gridded <- fg_grid_join |>
  group_by(cell_id) |>
  summarise(
    POP_DENS_2016 = mean(POP_DENS_2016, na.rm = TRUE),
    treatment     = mean(treatment,     na.rm = TRUE),
    .groups       = "drop"
  )
```

Plot the Gridded Result


Population Density Gridded to 0.05° × 0.05° Cells
Trade−off: lose polygon precision, gain spatial regularity
54°N
52°N
50°N
48°N
46°N
44°N
42°N
5°W
0°
5°E
10°E
15°E
log(Pop. Density)
10.0
7.5
5.0
2.5
0.0

- **Trade-off**: grid cells ignore administrative boundaries → some cells mix