Berlin Preprocessing

Nicole Dwenger and Orla Mallon

2021/06/01

Information

This document pre-processing spatial and attribute data of BERLIN for the final project of Spatial Analytics. Data for the following variables will be processed in the following:

1. Borough Shapes (spatial, vector)
2. Population and Age data (only attribute)
3. Country of Origin (pre-processed in `berlin_dot_density.rmd`)
4. Places of Worship (pre-processed in `berlin_osm_culture_religion.Rmd`)
5. Museums, Theatres & Nightlife (pre-processed in `berlin_osm_culture_religion.Rmd`)
6. Travel Time (spatial, vector and attribute)
7. Crime (attribute)
8. Rent (attribute)
9. Tree cover (spatial, raster)
10. Imperviousness (spatial, raster)

In a last step all pre-processed data will be joined in one data frame.

Loading Libraries

```
library(tidyverse)
library(sf)
library(leaflet)
library(glue)
library(readxl)
library(XML)
library(here)
library(raster)

# Function to load data from FIS broker, online data service for berlin
sf_fisbroker <- function(x) {
      u1 <- glue("http://fbinter.stadt-berlin.de/fb/wfs/data/senstadt/{x}")
      u2 <- glue("?service=wfs&version=2.0.0&request=GetFeature&TYPENAMES={x}")
      url <- paste0(u1, u2)
      print(url)
      s <- sf::st_read(url)
      s
}
```

1. Bezirke Berlin

Loading shapefiles for the Bezirke boundaries of Berlin, as vector data of polygons. The raw data is in 4326, so we transform it into a projected CRS (ETRS89 / UTM zone 33N, EPSG:25833), which fits for Germany, to extract centroids of each of the polygons (Bezirke) and area measurements. Lastly, we extract the relevant columns, and transform both the coordinates of the polygons and the coordinates of the centroids back to EPSG 4326, as this is the required CRS for Leaflet (which will be used in the shiny app) and calculate the area.

- Data source: https://daten.odis-berlin.de/de/dataset/bezirksgrenzen/

```
# Loading the shapefiles of the Berliner Bezirke
berlin_raw = st_read(here("berlin", "raw_data", "bezirke", "bezirksgrenzen.shp"))
```

```
## Reading layer ‘bezirksgrenzen’ from data source ‘/Users/nicoledwenger/Documents/University/6SEMESTER/
## Simple feature collection with 12 features and 6 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 13.08835 ymin: 52.33825 xmax: 13.76116 ymax: 52.67551
## Geodetic CRS:  WGS 84
```

```
# Checking the CRS: unprojected, 4326
st_crs(berlin_raw)
```

```
## Coordinate Reference System:
##   User input: WGS 84
##   wkt:
## GEOGCRS["WGS 84",
##     DATUM["World Geodetic System 1984",
##         ELLIPSOID["WGS 84",6378137,298.257223563,
##             LENGTHUNIT["metre",1]]],
##     PRIMEM["Greenwich",0,
##         ANGLEUNIT["degree",0.0174532925199433]],
##     CS[ellipsoidal,2],
##         AXIS["latitude",north,
##             ORDER[1],
##             ANGLEUNIT["degree",0.0174532925199433]],
##         AXIS["longitude",east,
##             ORDER[2],
##             ANGLEUNIT["degree",0.0174532925199433]],
##     ID["EPSG",4326]]
```

```
# To get centroids and area measurements the CRS needs to be projected
# Here choosing 25844 (ETRS89 / UTM zone 33N)
berlin_proj = berlin_raw %>% st_transform(crs = 25833)
# Checking CRS
st_crs(berlin_proj)
```

```
## Coordinate Reference System:
##   User input: EPSG:25833
##   wkt:
## PROJCRS["ETRS89 / UTM zone 33N",
```

```
##     BASEGEOGCRS["ETRS89",
##         DATUM["European Terrestrial Reference System 1989",
##             ELLIPSOID["GRS 1980",6378137,298.257222101,
##                 LENGTHUNIT["metre",1]]],
##         PRIMEM["Greenwich",0,
##             ANGLEUNIT["degree",0.0174532925199433]],
##         ID["EPSG",4258]],
##     CONVERSION["UTM zone 33N",
##         METHOD["Transverse Mercator",
##             ID["EPSG",9807]],
##         PARAMETER["Latitude of natural origin",0,
##             ANGLEUNIT["degree",0.0174532925199433],
##             ID["EPSG",8801]],
##         PARAMETER["Longitude of natural origin",15,
##             ANGLEUNIT["degree",0.0174532925199433],
##             ID["EPSG",8802]],
##         PARAMETER["Scale factor at natural origin",0.9996,
##             SCALEUNIT["unity",1],
##             ID["EPSG",8805]],
##         PARAMETER["False easting",500000,
##             LENGTHUNIT["metre",1],
##             ID["EPSG",8806]],
##         PARAMETER["False northing",0,
##             LENGTHUNIT["metre",1],
##             ID["EPSG",8807]]],
##     CS[Cartesian,2],
##         AXIS["(E)",east,
##             ORDER[1],
##             LENGTHUNIT["metre",1]],
##         AXIS["(N)",north,
##             ORDER[2],
##             LENGTHUNIT["metre",1]],
##     USAGE[
##         SCOPE["unknown"],
##         AREA["Europe - 12°E to 18°E and ETRS89 by country"],
##         BBOX[46.4,12,84.01,18.01]],
##     ID["EPSG",25833]]
```

```
# Getting centroids of each of the polygons / bezirke
berlin_proj$center_coords = st_centroid(berlin_proj$geometry)
# Getting area for each polygon / bezirk
berlin_proj$area_m2 = st_area(berlin_proj$geometry)
# Transforming into ha
berlin_proj$area_ha = as.numeric(berlin_proj$area_m2)/10000
# Transforming into km2
berlin_proj$area_km2 = as.numeric(berlin_proj$area_m2)/1000000

# Turing the bezirk geometry back into into 4326 for Leaflet
berlin_geo = berlin_proj %>%
  st_transform(crs = 4326) %>%
  # Selecting relevant columns
  dplyr::select("name" = Gemeinde_n,
                "area_ha" = area_ha,
                "area_km2" = area_km2,
```

```
                "center_coords" = center_coords, # coordinates, but not "geometry" of sf object
                "geometry" = geometry) # coordinates in 4326 for boroughs, "geometry" of sf object

# Separately also transforming the centroids again, the above step only transformed the "geometry"
berlin_geo$center_coords = st_transform(berlin_geo$center_coords, 4326)

# Checking CRS of center coordinates again
st_crs(berlin_geo$center_coords)$input
```

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

```
# Checking CRS of boroughs geometry again
st_crs(berlin_geo$geometry)$input
```

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

```
# Saving bezrik names
bezirk_names = berlin_geo$name
```

2. Population and Age

This preprocessed population and age data. First, we load a data frame, which contains population counts for age ranges of 10 years for 31.12.2019, for each borough. We sum all the population values to get a total count for each borough, and calculate percentages using the total and age-range population values. Lastly, we calculate a mean value using the middle value of each of the age ranges.

- Data source: Einwohnerregisterstatistik, 31.12.2019: https://www.statistik-berlin-brandenburg.de/webapi/jsf/tableView/tableView.xhtml

```
# Loading the age data as xlsx file, while specifying which columns to use
age_raw = read_excel(here("berlin", "raw_data", "age_population", "table_2021-04-30_16-43-38.xlsx"), col
```

```
## New names:
## * `` -> ...1
## * `` -> ...2
## * `` -> ...3
## * `` -> ...4
## * `` -> ...5
## * ...
```

```
# Cleaning up column names
colnames(age_raw) = c("rm", "name", "0_9", "10_19", "20_29", "30_39", "40_49", "50_59", "60_69", "70_79
# Dropping unimportant rows and columns
age_raw = age_raw[-c(1:3, 16:22),-1]
# Take out digits in the names
age_raw$name = gsub(pattern = "[0-9]+ ", "", age_raw$name)

# Calculating the total population for each Bezirk
age_raw[,2:11] <- sapply(age_raw[,2:11],as.numeric)
age_raw$total_population = rowSums(age_raw[,2:11])
```

```r
# Calculating the age range percentages for each Bezirk
age_perc = age_raw
# Getting the age groups
ranges = colnames(age_perc[2:11])
# Looping through the age groups and getting the percentage
for (range in ranges){
  # Calculating percentage
  percentage = age_perc[range]/age_perc["total_population"]
  # Creating corresponding column name
  col_name = paste0("perc_age_", range)
  # Saving value in column
  age_perc[col_name] = percentage
}

# Calculating the mean age for each borough
# Create copy of data frame
age_sumaries = age_perc
# Create empty age_mean column
age_sumaries$age_mean = NA
# For each row (Bezirk), calculate the mean by taking the population count and center value of the age
for(i in 1:nrow(age_sumaries)) {
  mean_age = (age_sumaries[i,2]*5.5 + age_sumaries[i,3]*15.5 + age_sumaries[i,4]*25.5 +
              age_sumaries[i,5]*35.5 + age_sumaries[i,6]*45.5 + age_sumaries[i,7]*55.5 +
              age_sumaries[i,8]*65.5 + age_sumaries[i,9]*75.5 + age_sumaries[i,10]*85.5 +
              age_sumaries[i,11]*95.5)/age_sumaries$total_population[i]
  # Round and save as age mean in data frame
  age_sumaries$age_mean[i] = round(mean_age,2)
}

# Select only relevant variables
age = age_sumaries[-c(2:11)]

# Turn to numeric
age$age_mean = as.numeric(age$age_mean)
```

3. Country of Origin

This data was processed in berlin_dot_density.rmd. Here only the preprocessed data is loaded.

```r
# Loading preprocessed data file
origin = read.csv(here("berlin", "preprocessed_data", "berlin_percentages.csv"))
colnames(origin)[1] = "name"
```

4. Places of Worship

This data was extraced and processed in a script called berlin_osm_culture_religion.Rmd.

```r
# Loading the preprocessed religion count data
religion = read.csv(here("berlin", "preprocessed_data", "berlin_religion_counts.csv"))
```

5. Museums, Theatres and Nightlife

This data was extraced and processed in a script called berlin_osm_culture_religion.Rmd.

```
# Loading the preprocessed culture count data
culture = read.csv(here("berlin", "preprocessed_data", "berlin_culture_counts.csv"))
```

6. Transit Travel Time

Here, we extract transit travel times from Google Maps, using the gmapsdistance package, and the Google Maps Distance Matrix API. We decided to define a set of points of interests of train stations and airports, and calculate the time it takes to travel from the district centroids to these POI. To get the data, it requires coordinates for "origin" and coordinates for "destination" in the format lat+lng, and in the CRS that Google Maps required the coordinates to be in (4326). The data is extracted for using transit, for UCT 6.00 (which is 8.00) in Berlin The extracted data in seconds is converted to minutes and stored in a dataframe. As it is not allowed to create content with data extracted from Google Maps, the data should not be further used, read more here: https://cloud.google.com/maps-platform/terms/.

- For more info on the gmapsdistance package: https://github.com/rodazuero/gmapsdistance

```
# Installing gmaps distance
#devtools::install_github("rodazuero/gmapsdistance")
library(gmapsdistance)
# Setting API key (removed for security)
set.api.key("")

# Getting centroids data
centers = berlin_geo %>%
  dplyr::select(name, center_coords) %>%
  # Dropping borough shape geometry
  st_drop_geometry() %>%
  # Turning into sf, so that centroids become the geometry
  st_as_sf() %>%
  # Mutate empty column
  mutate(origin = NA)

# For each row (centroid), bring the coordinates in the right format
# They should be in the same format as Google Maps data (4326, WGS84)
for (row in 1:nrow(centers)){
  # Extract the lat and long coordinates
  lat = st_coordinates(centers)[row, "Y"]
  lng = st_coordinates(centers)[row, "X"]
  # Paste together for gmapsdistance query
  centers$origin[row] = paste0(lat, "+", lng)
}

# Define locations of points of interest, coordinates were extracted manually from GoogleMaps
POI = rbind(c("train_central", 52.525041631523685, 13.36947054088901),
            c("train_westkreuz", 52.500967565415856, 13.283907398559895),
            c("train_ostkreuz", 52.50329675780353, 13.469369213903255),
            c("train_südkreuz", 52.47571455978377, 13.365571641664372),
            c("air_berlin_brandenburg", 52.36522784326457, 13.509633657421132))
```

```
# Saving the POI as a data frame to use in shiny app
POI = as.data.frame(POI)
# Fixing column names
colnames(POI) = c("id", "lat", "lng")
# Save csv of the coordinates for the PoI
write_csv(POI, "berlin/preprocessed_data/berlin_poi.csv")

# Here we are getting the gmaps travel time data, for a monday morning in May
# For each Point of Interest
for (i in 1:nrow(POI)){
  # Turn the coordinates into the correct format
  dest = paste0(POI[i,2], "+", POI[i,3])
  # Extract values from gmaps distance (6.00 UCT is 8.00 in Berlin)
  gmaps_values = gmapsdistance(origin = centers$origin, destination = dest,
                               dep_date = "2021-06-07", dep_time = "06:00:00", mode = "transit")
  # Append to data frame with column name of time_POI, and divide by 60 to get in minutes
  centers[,paste0("time_", POI[i,1])] = gmaps_values$Time[,2]/60
}

# Remove the center coordinates from the data frame, to only keep the travel time
travel_time = centers %>% st_drop_geometry() %>% dplyr::select(-origin) %>% as.data.frame()

# Save rds
saveRDS(travel_time, "berlin/preprocessed_data/berlin_travel_time.rds")
```

```
# Read rds
travel_time = readRDS(here("berlin", "preprocessed_data", "berlin_travel_time.rds"))
```

7. Crime

Here we simply load data from the Berlin Police, select those for 2020, and save them in a dataframe.

- Data source: https://daten.berlin.de/datensaetze/kriminalitätsatlas-berlin

```
# Loading the raw crime data as xlsx file and selecting relevant columns
crime_raw <- read_excel(here("berlin", "raw_data", "crime", "Fallzahlen&HZ 2012-2020.xlsx"), sheet = "Fa

# Filtering out only the relevant columns for the bezirke
crime_bezirke = crime_raw %>% dplyr::filter(`Bezeichnung (Bezirksregion)` %in% bezirk_names)
# Renaming column to reflect crime total
colnames(crime_bezirke)[3] <- "crime_total"

# Selecting relevant columns and renaming, to later add to final dataframe
crime = crime_bezirke %>%
  dplyr::select("name" = `Bezeichnung (Bezirksregion)`,
         "crime_total" = crime_total)
```

8. Rent

Here, we download data from a server called FISBROKER. A function to load data, is defined above, so we are using it here, to get the rent data for 2019 for the Prognoseräume (which is a smaller division of the

Berlin Bezirke). Thus, we also get the shapefiles for the Prognoseräume, and then match the Prognoseräume with the Bezirke to calcualte a median for each of the Bezirke.

- Data source:
  - Miete per m2, 2018, fisbroker: https://fbinter.stadt-berlin.de/fb/berlin/service_intern.jsp?id=s_wohnatlas2019@senstadt&type=WFS
  - Prognoseräume: fisbroker: https://fbinter.stadt-berlin.de/fb/berlin/service_intern.jsp?id=s_lor_prog@senstadt&type=WFS

```
# Loading the rent data from Fisbroker, online service
# This data is not on the "Bezirke"-Level, but is divided into smaller areas, but we'll convert them to
miete_raw = sf_fisbroker("s_wohnatlas2019")
```

```
## [1] "http://fbinter.stadt-berlin.de/fb/wfs/data/senstadt/s_wohnatlas2019?service=wfs&version=2.0.0&r
## Reading layer 's_wohnatlas2019' from data source 'http://fbinter.stadt-berlin.de/fb/wfs/data/senstad
```

```
## Warning: no simple feature geometries present: returning a data.frame or tbl_df
```

```
# Selecting the relevant columns of the dataframe
miete = miete_raw %>% dplyr::select("prog_name" = PROGNOSERAUM_BEZEICHNUNG,
                                    "miete" = ANGEBOTSMIETEN_2018) # median euro/m2

# Fixing some of the Prognoseräume names to match with other berlin data
miete$prog_name[miete$prog_name == "Kreuzberg-Nord"] = "Kreuzberg Nord"
miete$prog_name[miete$prog_name == "Schöneberg-Nord"] = "Schöneberg Nord"
miete$prog_name[miete$prog_name == "Heiligensee-Konradshöhe"] = "Heiligensee/Konradshöhe"
miete$prog_name[miete$prog_name == "Reinickendorf-Ost"] = "Reinickendorf Ost"
miete$prog_name[miete$prog_name == "Frohnau-Hermsdorf"] = "Frohnau/Hermsdorf"

# Making a value of 0 na
miete$miete[miete$miete == 0] = NA

# Getting the shapefiles of the Progoseräume
prog_raw = sf_fisbroker("s_lor_prog")
```

```
## [1] "http://fbinter.stadt-berlin.de/fb/wfs/data/senstadt/s_lor_prog?service=wfs&version=2.0.0&reques
## Reading layer 's_lor_prog' from data source 'http://fbinter.stadt-berlin.de/fb/wfs/data/senstadt/s_l
## Simple feature collection with 60 features and 5 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 370000.7 ymin: 5799520 xmax: 415785.3 ymax: 5837259
## CRS:           NA
```

```
colnames(prog_raw)
```

```
## [1] "gml_id"                "PROGNOSERAUM"          "BEZIRKSNAME"
## [4] "DATUM_GUELTIG_AB"      "FLAECHENGROESSE_IN_M2" "geometry"
```

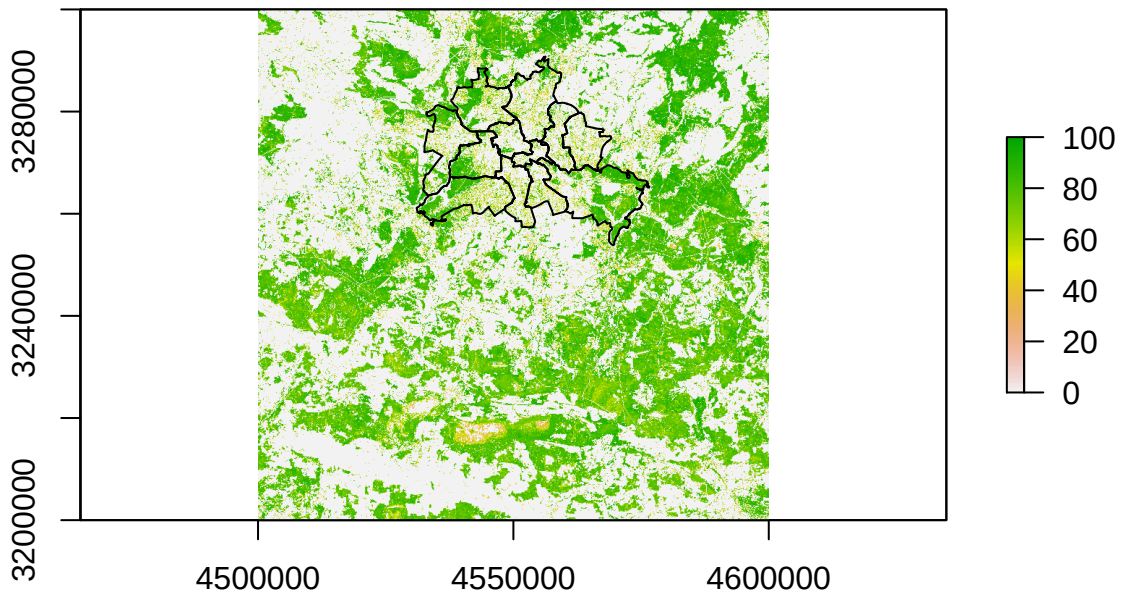
```r
# Setting correct CRS (info from from website)
st_crs(prog_raw) = 25833
# Getting the info of which Progrnoseraum belongs to which Bezirk, to match it up
prog = prog_raw %>% dplyr::select("prog_name" = PROGNOSERAUM,
                                  "name" = BEZIRKSNAME)

# Joining and summarising, by calculating the median of Prognoseräume for each Bezirk (euro/m2)
rent = left_join(miete, prog, "prog_name") %>%
  group_by(name) %>%
  summarise(median_rent = median(miete, na.rm = T))
```

9. Tree Cover (Raster)

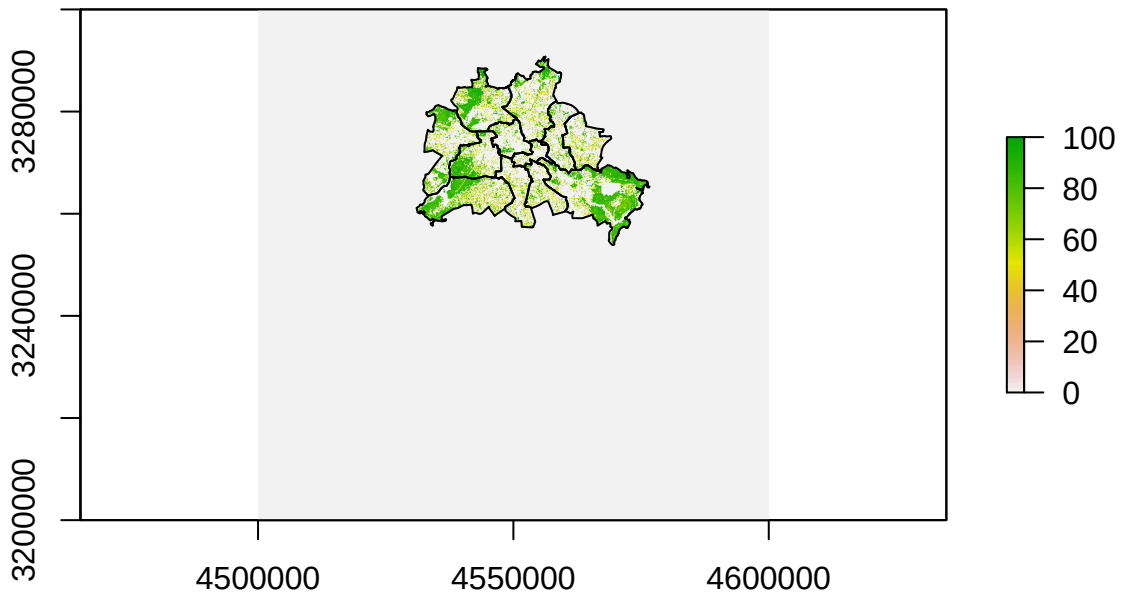
The aim was a value of mean percentage of green cover for each of the Bezirke Thus, we extracted raster data from the Corpernicus Land Monitoring Service, which contains percentages of green cover on a high resolution. The data is split into tiles. Thus, the relevant tiles which covered Berlin were found and loaded. Then, we transform the vector data of the boroughs to the crs of the raster data, we mask and crop the raster data using the transformed vector data, and lastly extract mean values of the raster data (green cover percentage) for each of the Bezirke Note that here some of the code is commented out, to be able to knit the document. If you are running the code, a warning message might occur about the datum and projection. This was not deemed to have an affect on results, and thus ignored. You can read more about the warning here: https://stackoverflow.com/questions/63374488/since-update-of-sp-package-i-get-a-warning-by-calling-a-spcrs-definition.

- Data source: https://land.copernicus.eu/pan-european/high-resolution-layers/forests/tree-cover-density/status-maps/tree-cover-density-2018

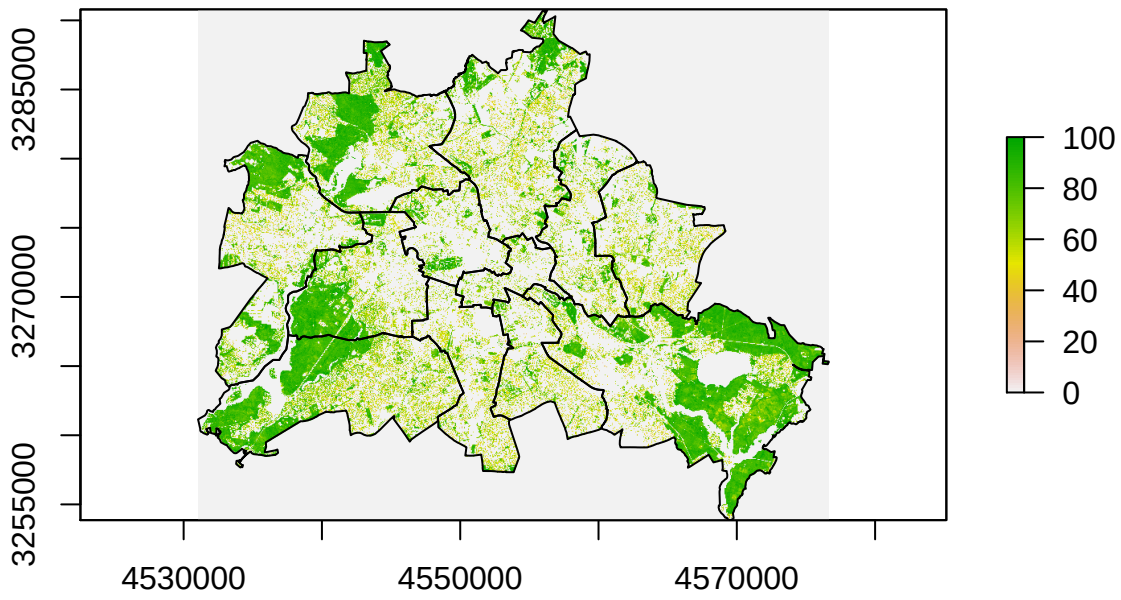
```r
# Loading the relevant raster file which covers Berlin
treecover_raw = raster::raster(here("berlin", "raw_data", "treecover", "TCD_2018_010m_E45N32_03035_v020

# Getting the CRS of the raster file: EPSG 3035
crs_3035 = crs(treecover_raw, asText=TRUE)
# Transforming the berlin geo to the crs
berlin_3035 = st_transform(berlin_geo, crs = crs_3035)

# Plotting the raster and geometry data
plot(treecover_raw, col=rev(terrain.colors(100)));
plot(berlin_3035$geometry, add = T)
```




```
# Masking the raster data to only keep Berlin area
# tree_mask = mask(treecover_raw, berlin_3035)
# Saving masked raster
# writeRaster(tree_mask, "berlin/raw_data/treecover/masked_treecover.tif")
# Read masked raster for plotting and knitting
tree_mask = raster(here("berlin", "raw_data", "treecover", "masked_treecover.tif"))
# Plotting
plot(tree_mask, col=rev(terrain.colors(100)));
plot(berlin_3035$geometry, add = T)
```




```
# Cropping the raster data to the polygons / bezirke of Berlin
# tree_cropped = crop(tree_mask, berlin_3035)
# Save cropped raster
# writeRaster(tree_cropped, "berlin/raw_data/treecover/cropped_treecover.tif")
# Read raster for plotting and knitting
tree_cropped = raster(here("berlin", "raw_data", "treecover", "cropped_treecover.tif"))
# Plotting
plot(tree_cropped, col=rev(terrain.colors(100)));
plot(berlin_3035$geometry, add = T)
```




```
# Print information about raster that will be used for extracting values
tree_cropped
```

```
## class      : RasterLayer
## dimensions : 3693, 4561, 16843773  (nrow, ncol, ncell)
## resolution : 10, 10  (x, y)
## extent     : 4531040, 4576650, 3253860, 3290790  (xmin, xmax, ymin, ymax)
## crs        : +proj=laea +lat_0=52 +lon_0=10 +x_0=4321000 +y_0=3210000 +ellps=GRS80 +units=m +no_defs
## source     : /Users/nicoledwenger/Documents/University/6SEMESTER/Spatial Analytics/FINALPROJECT/cdss
## names      : cropped_treecover
## values     : 0, 100  (min, max)
```

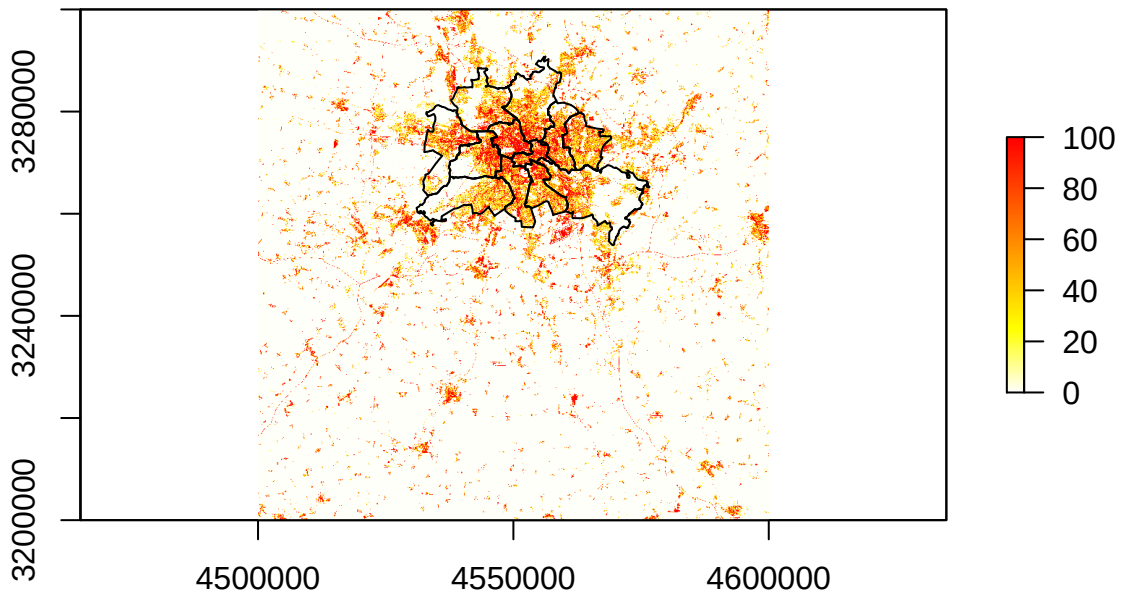
```
# Extract the values, using the mean, commented out for knitting
# bezirke_treecover = raster::extract(x=tree_cropped, y=berlin_3035, fun=mean)

# Save as a dataframe together with the names of the bezirke to add to final data
# treecover = as.data.frame(bezirk_names) %>% mutate("treecover" = bezirke_treecover)
# colnames(treecover)[1] = "name"

# Save rds
# saveRDS(treecover, "berlin/preprocessed_data/berlin_tree_extracted.rds")
# Load rds for knitting
treecover = readRDS(here("berlin", "preprocessed_data", "berlin_tree_extracted.rds"))
```

10. Imperviousness (Raster)

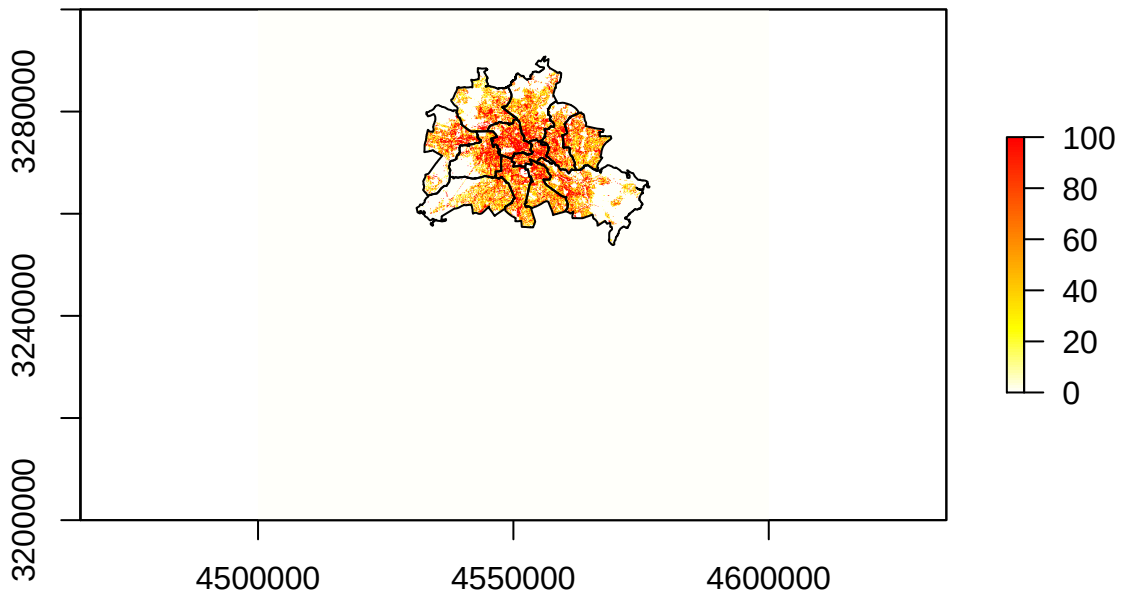
The aim was a value of mean percentage of imperviousness for each of the Bezrike Thus, we extracted raster data from the Corpernicus Land Monitoring Service, which contains percentages of imperviousness on a high resolution. The data is split into tiles. Thus, the relevant tiles which covered Berlin were found and loaded. Then, we transform the vector data of the boroughs to the crs of the raster data, we mask and crop the raster data using the transformed vector data, and lastly extract mean values of the raster data (imperviousness percentage) for each of the Bezirke. Note that here some of the code is commented out, to be able to knit the document.If you are running the code, a warning message might occur about the datum and projection. This was not deemed to have an affect on results, and thus ignored. You can read more about the warning here: https://stackoverflow.com/questions/63374488/since-update-of-sp-package-i-get-a-warning-by-calling-a-spcrs-definition.

- Data source: https://land.copernicus.eu/pan-european/high-resolution-layers/imperviousness/status-maps/imperviousness-density-2018

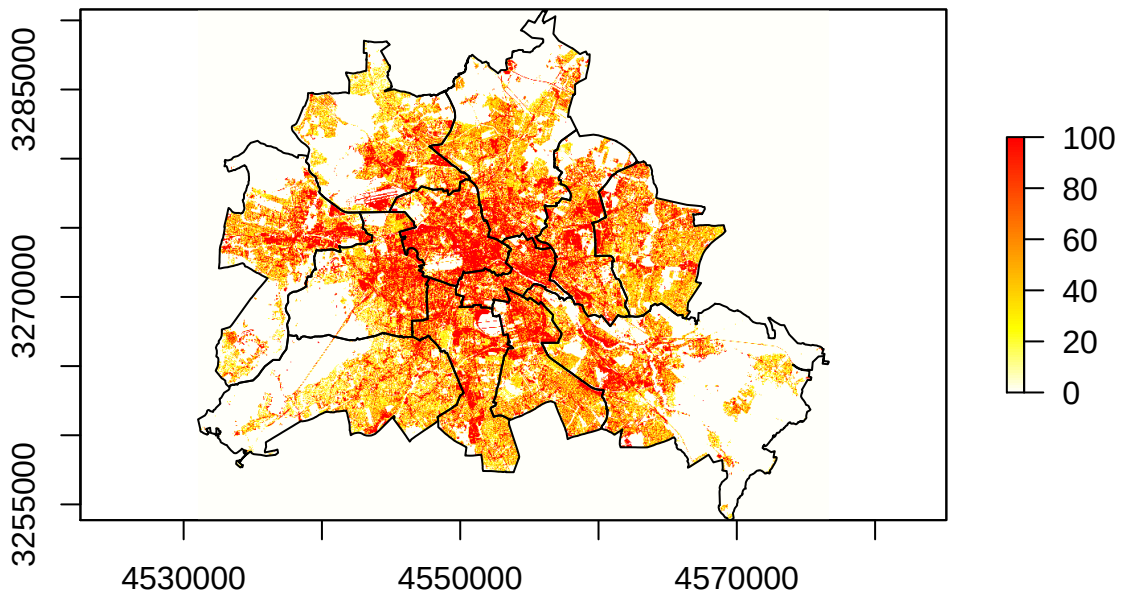
```
# Loading the relevant raster files which cover Berlin
imperviousness_raw = raster(here("berlin", "raw_data", "imperviousness", "IMD_2018_010m_E45N32_03035_v0

# Getting the CRS of the raster file: EPSG 3035
crs_3035 = crs(imperviousness_raw, asText=TRUE)
# Transforming the berlin geo to this crs
berlin_3035 = st_transform(berlin_geo, crs = crs_3035)

# Plotting the raster and geometry data
plot(imperviousness_raw, col=rev(heat.colors(100)));
plot(berlin_3035$geometry, add = T)
```




```
# Masking the raster data to only keep Berlin area
# imperv_mask = mask(imperviousness_raw, berlin_3035)
# Saving masked raster
# writeRaster(imperv_mask, "berlin/raw_data/imperviousness/masked_imperv.tif")
# Read masked raster for plotting and knitting
imperv_mask = raster(here("berlin", "raw_data", "imperviousness", "masked_imperv.tif"))
# Plotting
plot(imperv_mask, col=rev(heat.colors(100)));
plot(berlin_3035$geometry, add=TRUE)
```




```
# Cropping the raster data to the polygons / bezirke of Berlin
# imperv_cropped = crop(imperv_mask, berlin_3035)
# Saving cropped raster
# writeRaster(imperv_cropped, "berlin/raw_data/imperviousness/cropped_imperv.tif")
# Read cropped raster for plotting and knitting
imperv_cropped = raster(here("berlin", "raw_data", "imperviousness", "cropped_imperv.tif"))
# Plotting
plot(imperv_cropped, col=rev(heat.colors(100)));
plot(berlin_3035$geometry, add=TRUE)
```




```
# Print information of the raster layer which will be used for extracting
imperv_cropped
```

```
## class      : RasterLayer
## dimensions : 3693, 4561, 16843773  (nrow, ncol, ncell)
## resolution : 10, 10  (x, y)
## extent     : 4531040, 4576650, 3253860, 3290790  (xmin, xmax, ymin, ymax)
## crs        : +proj=laea +lat_0=52 +lon_0=10 +x_0=4321000 +y_0=3210000 +ellps=GRS80 +units=m +no_defs
## source     : /Users/nicoledwenger/Documents/University/6SEMESTER/Spatial Analytics/FINALPROJECT/cdssp
## names      : cropped_imperv
## values     : 0, 100  (min, max)
```

```
# Extract the values, using the mean, commented out for knitting
# bezirke_imperv = raster::extract(x=imperv_cropped, y=berlin_3035, fun=mean)

# Save as a dataframe together with the names of the bezirke to add to final data
# imperviousness = as.data.frame(bezirk_names) %>% mutate("imperviousness" = bezirke_imperv)
# colnames(imperviousness)[1] = "name"

# Save rds
# saveRDS(imperviousness, "berlin/preprocessed_data/berlin_imperv_extracted.rds")
# Read rds for knititng
imperviousness = readRDS(here("berlin", "preprocessed_data", "berlin_imperv_extracted.rds"))
```

FINAL: Joining everything together

```
# Joining all varaibles together
berlin = left_join(berlin_geo, age, by = "name") %>%
         mutate(population_dens_ha = (total_population/area_ha)) %>%
         mutate(population_dens_km2 = (total_population/area_km2)) %>%
         left_join(., origin, by = "name") %>%
         left_join(., religion, by = "name") %>%
         left_join(., culture, by = "name") %>%
         left_join(., travel_time, by = "name") %>%
         left_join(., crime, by = "name") %>%
         mutate(crime_rate = (crime_total/total_population)*1000) %>%
         left_join(., rent, by = "name") %>%
         left_join(., treecover, by = "name") %>%
         left_join(., imperviousness, by = "name")

# Prining columnnames
colnames(berlin)
```

```
##  [1] "name"                          "area_ha"
##  [3] "area_km2"                      "total_population"
##  [5] "perc_age_0_9"                  "perc_age_10_19"
##  [7] "perc_age_20_29"                "perc_age_30_39"
##  [9] "perc_age_40_49"                "perc_age_50_59"
## [11] "perc_age_60_69"                "perc_age_70_79"
## [13] "perc_age_80_89"                "perc_age_90+"
## [15] "age_mean"                      "population_dens_ha"
## [17] "population_dens_km2"           "origin_perc_africa"
## [19] "origin_perc_asia"              "origin_perc_europe"
## [21] "origin_perc_north_america"     "origin_perc_oceania"
## [23] "origin_perc_south_america"     "origin_perc_unknown"
## [25] "n_buddhist"                    "n_hindu"
## [27] "n_jewish"                      "n_muslim"
## [29] "n_protestant"                  "n_catholic"
## [31] "n_museum"                      "n_theatre"
## [33] "n_nightlife"                   "time_train_central"
## [35] "time_train_westkreuz"          "time_train_ostkreuz"
## [37] "time_train_südkreuz"           "time_air_berlin_brandenburg"
## [39] "crime_total"                   "crime_rate"
## [41] "median_rent"                   "treecover"
## [43] "imperviousness"                "center_coords"
## [45] "geometry"
```

```
# Save as rds
#saveRDS(berlin, "berlin/preprocessed_data/berlin.rds")
# Save as csv
#write.csv(berlin, "berlin/preprocessed_data/berlin.csv")
```