Handout – Working with Geo-Data

NaWi-Workshop: Obtaining, linking and plotting geographic data

Markus Konrad markus.konrad@wzb.eu

May 6, 2019

Plotting with *ggplot2*

The following will only a cover some basic explanations and recipes. I will only show scatterplots as examples, but the basic concepts can be applied to all other kinds of plots. Specifically, geographic plots with `geom_sf()` work in a very similar fashion, only that the coordinate system usually uses geo-locations in longitude / latitude degrees on the x- and y-axis.

For a more thorough introduction to ggplot2, see chapter 3 in "R for Data Science".

An example dataset

We'll use the built-in `airquality` dataset:

```
head(airquality)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

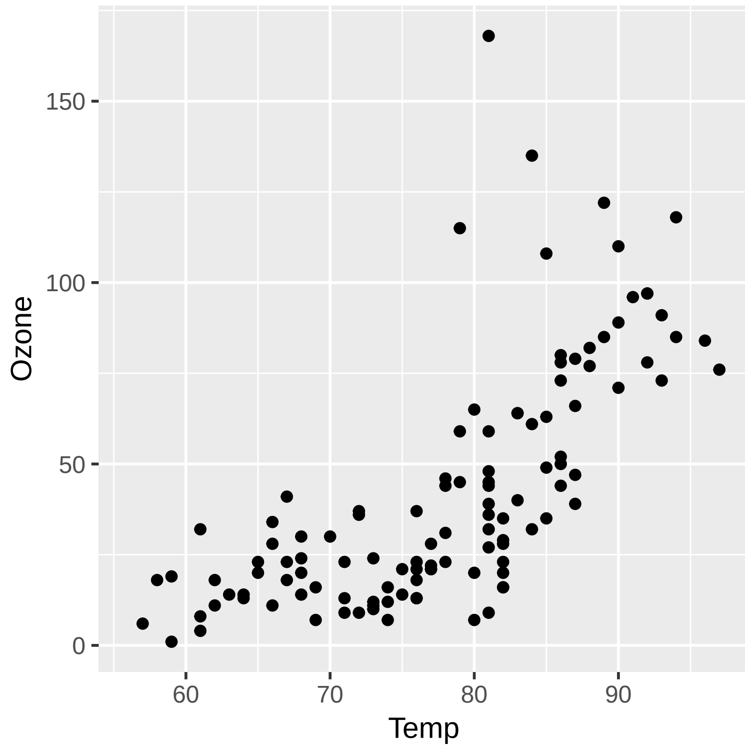
In its very essence, you make a plot by:

1. setting the dataset to use for plotting
2. specifying an ***aesthetics* mapping** that defines which visual properties of the plot are controlled by which variables in your dataset (e.g. variable `Temp` is mapped to the x-coordinate in your plot, `Ozone` is mapped to the y-coordinate)
3. setting a graphical primitive (e.g. a line, points, etc.) to use for plotting one layer of the data; this is called a ***geom***; you may add several layers of different graphical primitives to your plot, e.g. a layer for a line showing a trend and a layer of points that display the individual data points

The first two steps can be done by using the `ggplot()` function. It accepts the dataset to use and an aesthetic mapping which is defined in the function `aes()`. You then add a layer of points via the *geom*-function `geom_point()`. You combine all these steps by using the `+` operator.

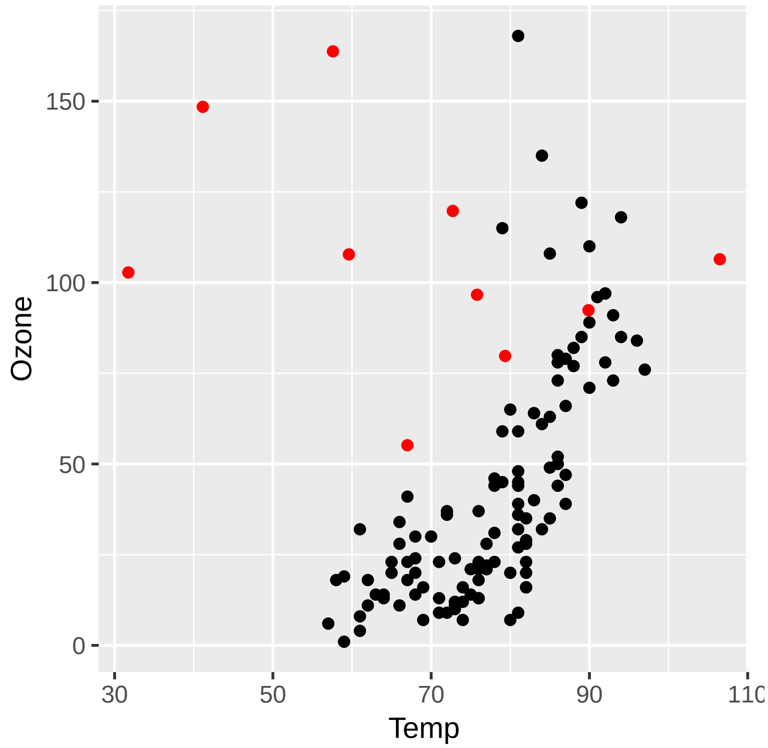
An example of a scatterplot of `Ozone` on the y-axis against `Temp` on the x-axis:

```
library(ggplot2)

ggplot(airquality, aes(x = Temp, y = Ozone)) + geom_point()
```



Be aware that the dataset and the mapping that you pass to `ggplot()` affects all layers unless they define their own data to use and/or aesthetic mapping. For example, we could also refrain from setting a dataset and aesthetic mapping in the `ggplot()` function and instead create two layers of points, each using a different dataset and aesthetic mapping:

```
random_data <- data.frame(rand_x = rnorm(10, mean = 75,
                                         sd = 20),
                          rand_y = rnorm(10, mean = 100,
                                         sd = 30))

ggplot() +
    geom_point(aes(x = Temp, y = Ozone), data = airquality) +
    geom_point(aes(x = rand_x, y = rand_y), color = 'red',
               data = random_data)
```

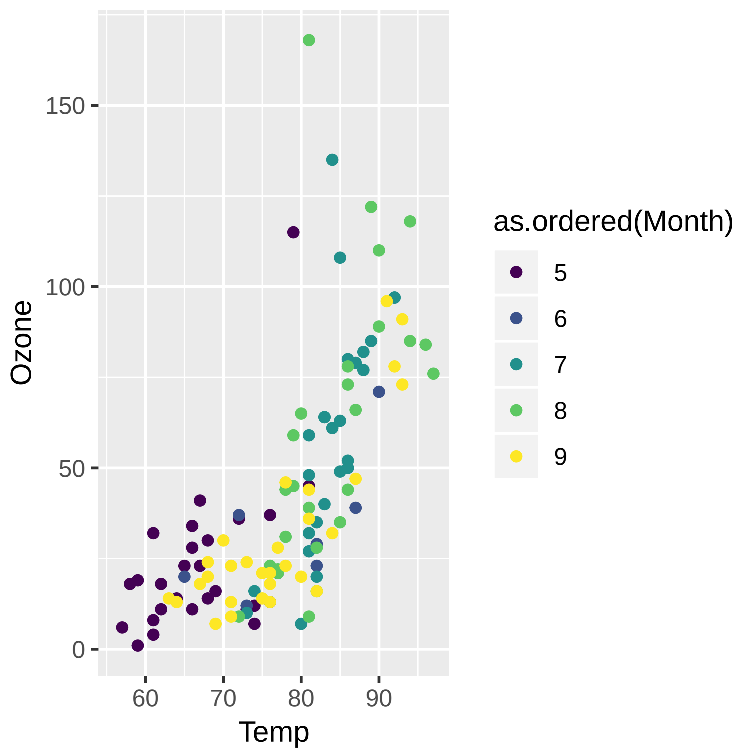


The first `geom_point()`-layer creates points for the `airquality` data just like before. The second layer uses a dataset with randomly generated values `rand_x` and `rand_y`. Additionally, the color for those points is set to red.

You can see that `geom_point()` has several visual properties (i.e. "aesthetics"): The position on the x and y scale; the color of the points. Some of these properties, like x and y position, are *required*. They must be mapped to a variable or set to a static value, because otherwise you can't draw a point. Others, like `color` are optional and they have reasonable default values.

All aesthetics can be mapped to a variable, so we can also map `color` to a variable (note that the variable is converted to an ordered factor, because otherwise the color would be on a continuous scale):

```
# here we only pass the dataset to ggplot() and define the aesthetic mapping in the geom layer
ggplot(airquality) +
    geom_point(aes(x = Temp, y = Ozone,
                   color = as.ordered(Month)))
```

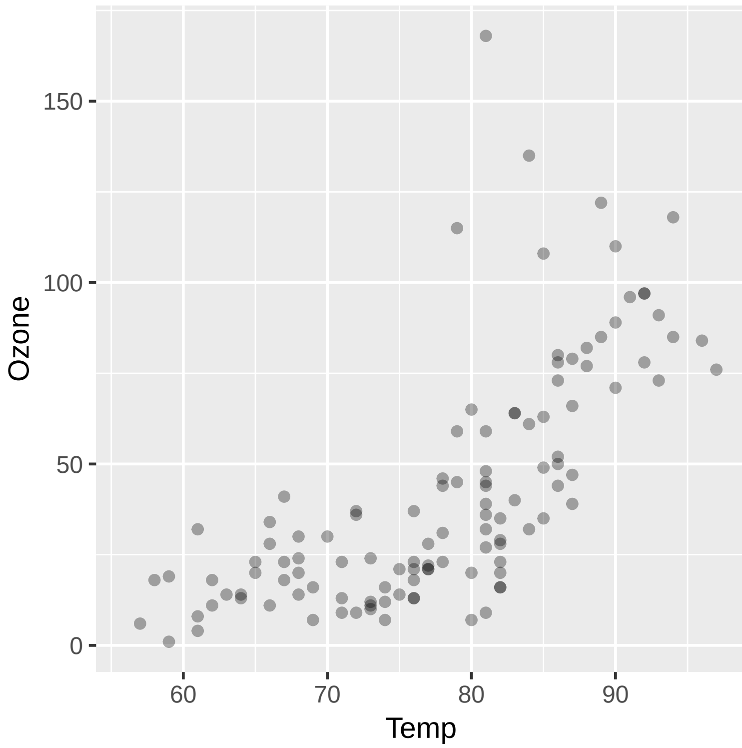


In order to find out which visual properties of a geom can be controlled, you can use the help document of the respective function, e.g. `?geom_point` or `?geom_line` and have a look at the section "Aesthetics".

Overplotting can easily occur, especially with large data sets.

- happens when multiple data points are drawn on the same spot
- fix it with setting a semi-transparent fill color or apply *jittering*

```
ggplot(airquality, aes(x = Temp, y = Ozone)) +
  geom_point(alpha = 0.33)  # alpha of 0 is invisible
```

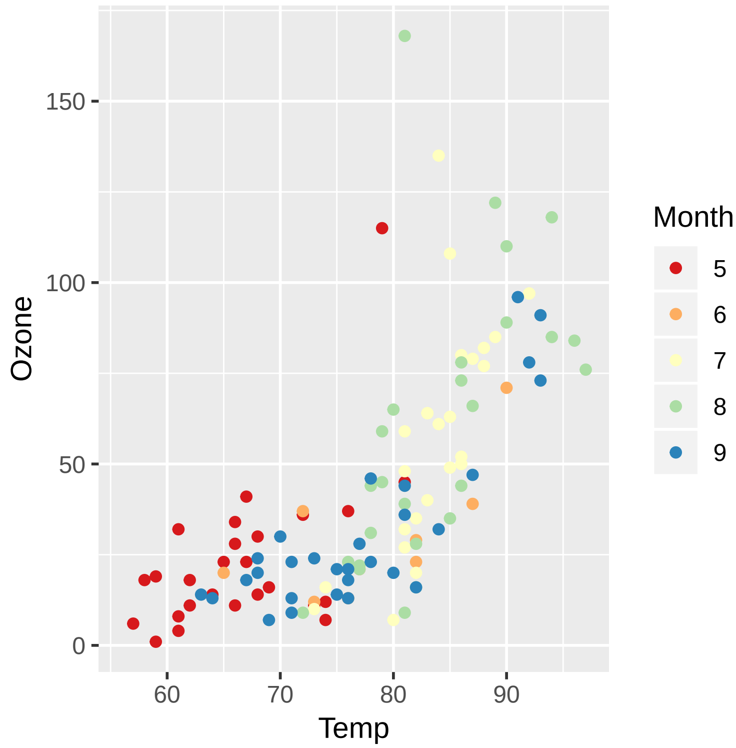



```
# 1 is opaque
```

Scales

Each visual property that a variable maps to, belongs to a scale that you can further adjust. For example, you might apply a log-transformation to the x- and/or y-axis. Or you can also change the color palette that is used for the color of points in a scatterplot. If a scale uses a legend, you can adjust its appearance, change the title or its position, among other things:

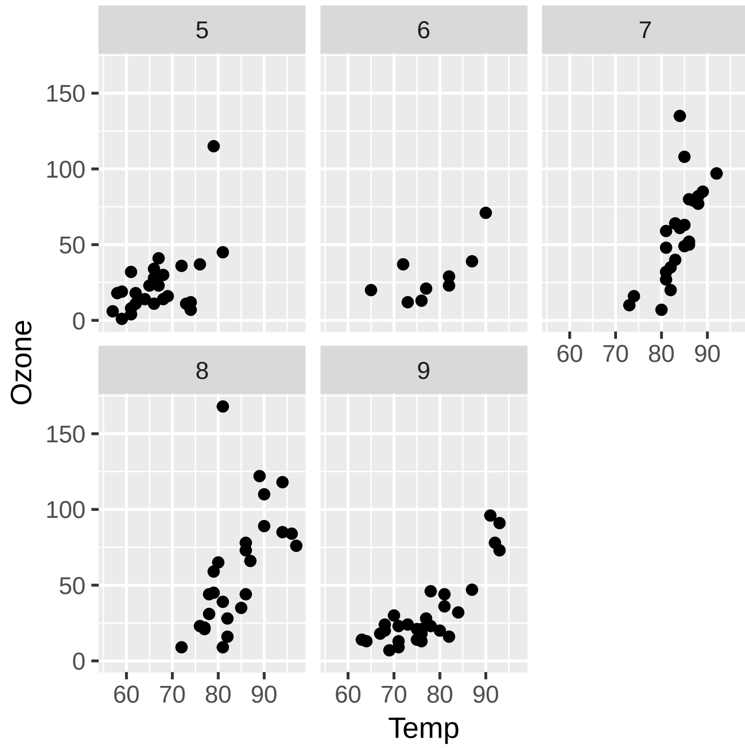
```
ggplot(airquality) +
    geom_point(aes(x = Temp, y = Ozone,
                   color = as.ordered(Month))) +
    scale_color_brewer(palette = 'Spectral',
                       guide = guide_legend(title = 'Month'))
```



Facets

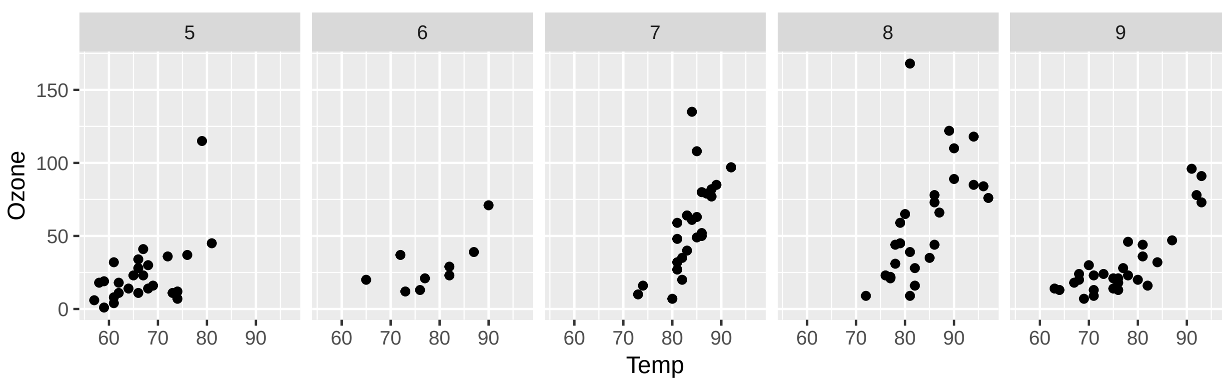
Facets allow you to create small multiples of your plots. Instead of projecting all data points into a single plot, split them into groups depending on a variable and then create a small plot for each group. You can use `facet_wrap()` to do this and specify the variable for splitting as "R formula", e.g. `~ X` where `X` is the variable to split by:

```
# make a plot for each month. convert to ordered factor before
ggplot(airquality) +
    geom_point(aes(x = Temp, y = Ozone)) +
    facet_wrap(~ as.ordered(Month))
```



You can specify more options in `facet_wrap()`, e.g. restrict the number of rows or columns.

```
ggplot(airquality) +
    geom_point(aes(x = Temp, y = Ozone)) +
    facet_wrap(~ as.ordered(Month), nrow = 1)
```



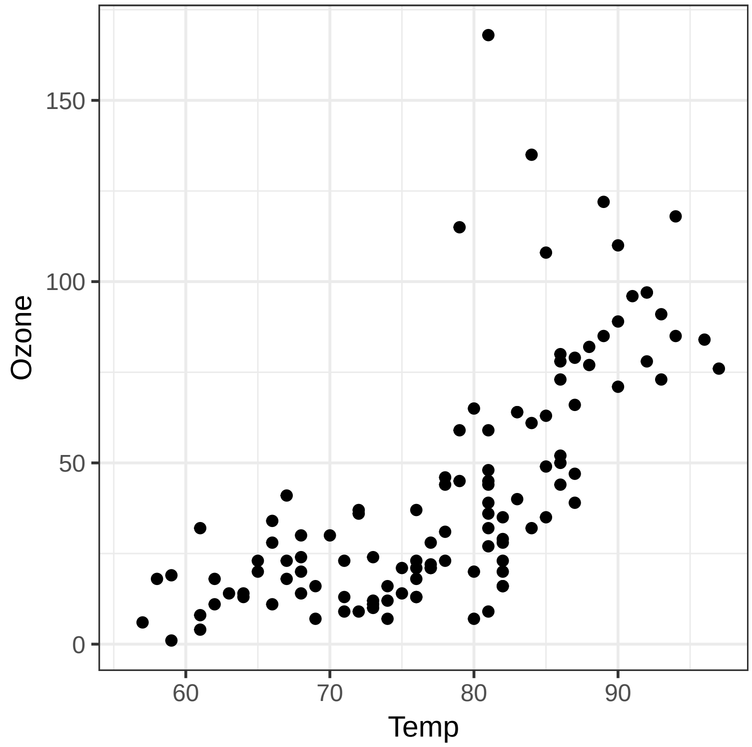
By default, all plots share the same scales on the axes which ensures better comparability. However, you can change that using the `scales` argument.

Themes

Themes control the overall appearance of a plot. There are several predefined themes which define a "plot style". They are all prefixed by `theme_`.

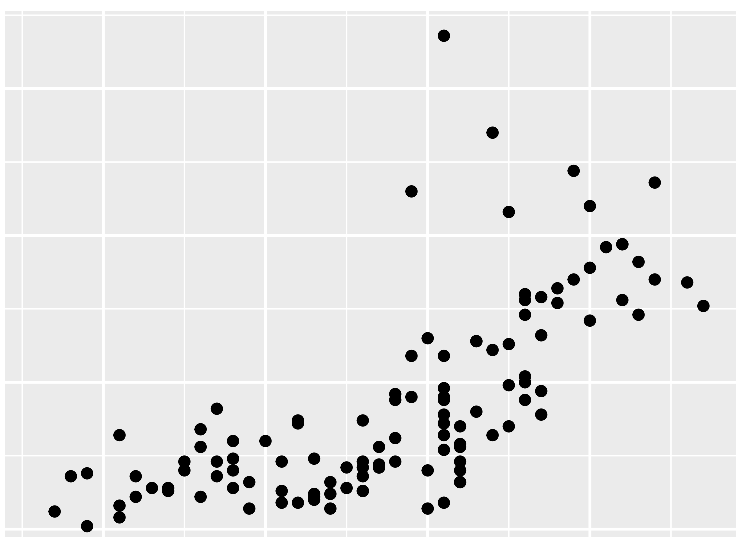
For example, `theme_bw()` which is optimized for black and white prints:

```r
ggplot(airquality, aes(x = Temp, y = Ozone)) + geom_point() +
    theme_bw()
```



For geo-spatial plots, you can often remove the axis labels and ticks, which can be done like this:

```r
ggplot(airquality, aes(x = Temp, y = Ozone)) + geom_point() +
    theme(axis.title = element_blank(),
          axis.ticks = element_blank(),
          axis.text = element_blank())
```



An alternative is `theme_void()`, which removes all elements around the actual plot.

Plots as objects

A ggplot object can be assigned a name just as any other object in R:

```
myplot <- ggplot(airquality, aes(x = Temp, y = Ozone))
myplot   # shows an "empty" plot
```

You can re-use the ggplot object and try out different layers or themes:

```
myplot + geom_point()
```

```
myplot + geom_point(position = position_jitter()) +
    theme_minimal()
```

You can eventually save the plot to disk with `ggsave()`:

```
final_plot <- myplot + geom_point(position = position_jitter())
ggsave('example_saved_figure.png',
       plot = final_plot,
       width = 4,
       height = 3)
```

There are several options to configure the output file (see `?ggsave`):

- plot dimensions (by default in inch)
- plot resolution
- format (PNG, PDF, etc.) – determined by file extension

Common mistakes

A very common mistake is to accidentally put `+` on a new line:

```
ggplot(airquality, aes(x = Temp, y = Ozone))
  + geom_point()
```

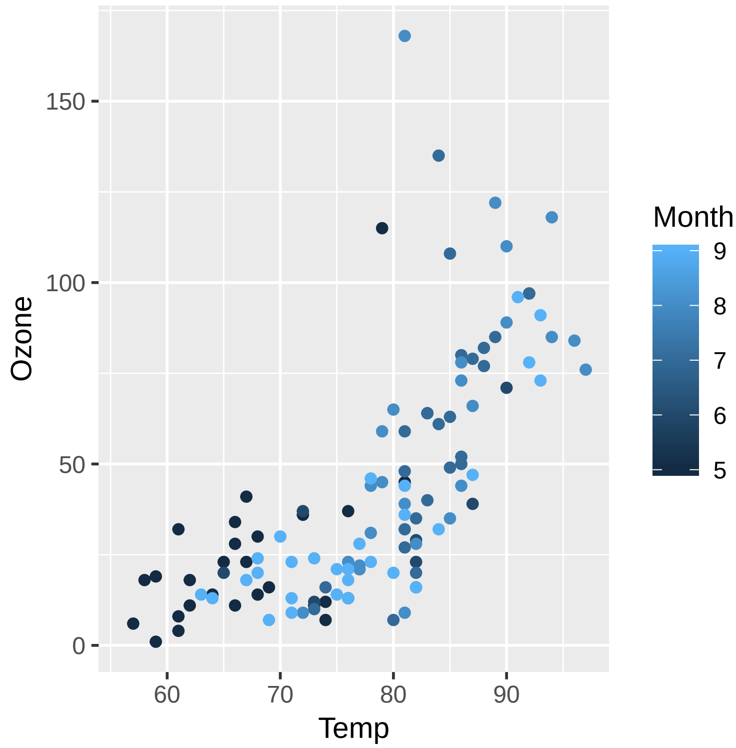
```
Error: Cannot use "+.gg()" with a single argument. Did you accidentally put + on a new
line?
```

The `+` operator must appear before the line break (the same is true for other operators like `%>%` used in *dplyr*):

```
ggplot(airquality, aes(x = Temp, y = Ozone)) +
  geom_point()
```

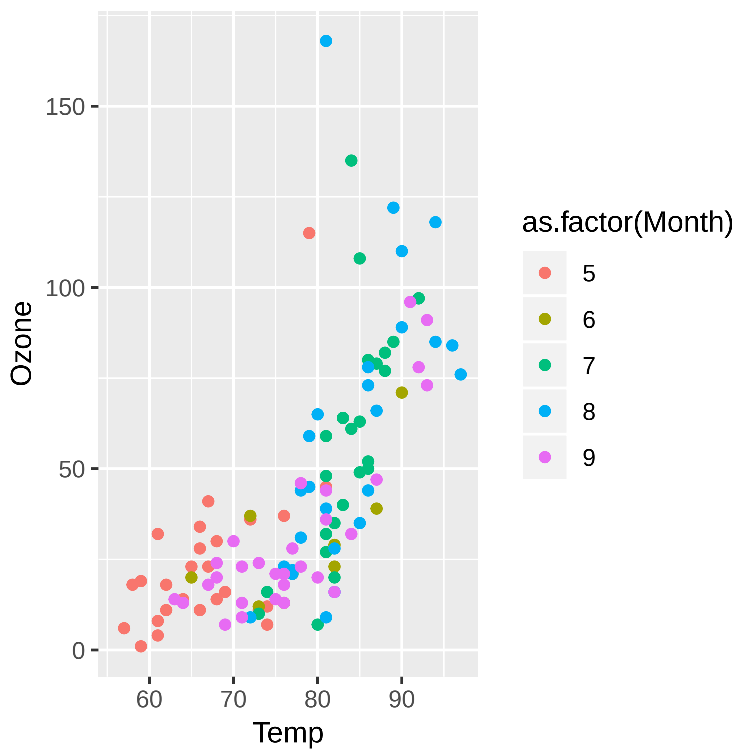
The type of your variables determines its scale for plotting. E.g. here you might want to use a discrete scale:

```
ggplot(airquality, aes(x = Temp, y = Ozone, color = Month)) +
  geom_point()
```



Converting the numerical to a factor tells ggplot that a discrete scale is appropriate:

```
ggplot(airquality, aes(x = Temp, y = Ozone,
                       color = as.factor(Month))) +
  geom_point()
```



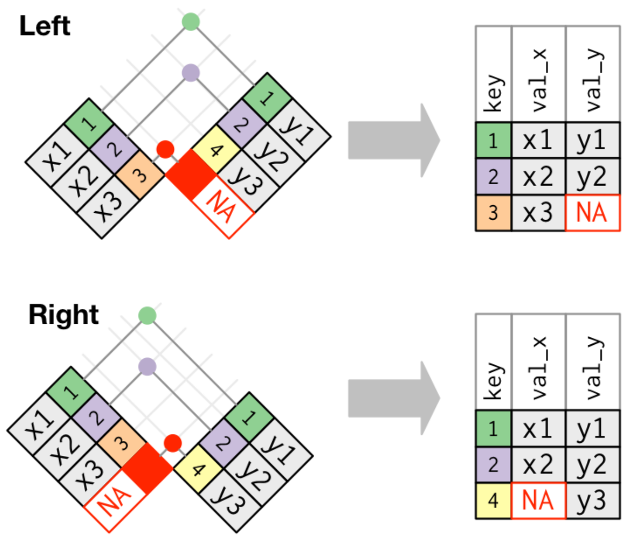


Figure 1: Left and right join. Source: Grolemund, Wickham 2017: R for Data Science

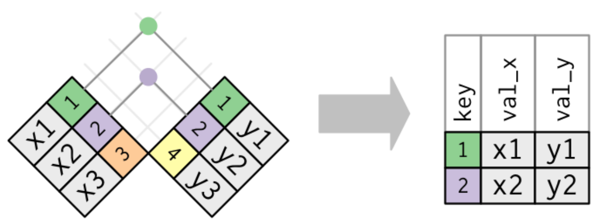


Figure 2: Inner join. Source: Grolemund, Wickham 2017: R for Data Science

Data linkage with *dplyr*

Left and right (outer) joins

Left and right outer joins keep all observations on the left-hand or right-hand side data sets respectively. Unmatched rows are filled up with *NA*s:

Syntax: `inner_join(a, b, by = <criterion>)`

Inner joins

An *inner join* matches keys that appear in both data sets and returns the combined observations:

Syntax: `inner_join(a, b, by = <criterion>)`

Specifying matching criteria

Parameter `by` can be:

1. a character string specifying the key for both sides, e.g.: `inner_join(pm, city_coords, by = 'city')` will match `city` column in `pm` with `city` column in `city_coords`;

2. a vector of character strings specifying several keys to match both sides, e.g.: `inner_join(pm, city_coords, by = c('city', 'country')` will match those rows, where `city` *and* `country` columns match;
3. a *named* character string vector like `inner_join(pm, city_coords, by = c('cityname' = 'id')`, which will match the column `cityname` in `pm` with the column `id` in `city_coords`

Specific hints / further information for excercises

Exercise 2

Finding out geo-coordinates

We will later learn how to use the Google Maps API to geocode (i.e. get the geo-coordinates) places programmatically. For the purpose of this excerise, it's enough to do it manually.

There are several websites that offer free manual geocoding, e.g.:

- https://google-developers.appspot.com/maps/documentation/utils/geocoder/
- https://www.mapdevelopers.com/geocode_tool.php

Both work the same way: You enter a request (i.e. an address, city name, restaurant name, etc.) and it spits out the result, including the longitude and latitude. **Please be aware that the first service returns the geo-coordinate with latitude first, followed by longitude ("Location: ...").**

Constructing a dataset quickly from within R

You can construct the small dataset directly within R by passing place labels, longitude and latitudes as separate column vectors:

```
places <- data.frame(
    label = c('born', 'living', 'neven been there'),
    lng = c(  12.590, 13.402,   8.0456),
    lat = c(  51.279, 52.520,   52.276)
)
```

Loading the worldmap dataset

The following loads the world map dataset from the `maps` package as *Simple Features* spatial dataset:

```
library(maps)
library(sf)

worldmap_data <- st_as_sf(map('world', plot = FALSE,
                              fill = TRUE))
```

Filtering the worldmap dataset

You can filter the worldmap data from the `maps` package by using the "ID" column:

```
sweden <- worldmap_data[worldmap_data$ID == 'Sweden',]
```

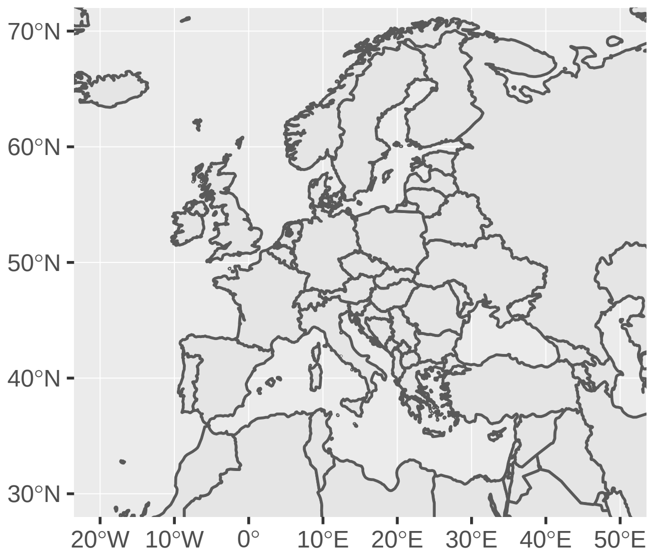
Using the `%in%` operator when selecting several countries:

```
sel_cntrs <- c('Sweden', 'Denmark', 'Finland', 'Norway', 'Iceland')
scandinavia <- worldmap_data[worldmap_data$ID %in% sel_cntrs,]
```

Restricting the display window

You can specify a "display window" (i.e. "zooming in" to a certain region) by setting a limit on the displayed longitude range (`xlim`) and latitude range (`ylim`) in the `coord_sf()` function:

```
ggplot() + geom_sf(data = worldmap_data) +
    coord_sf(xlim = c(-20, 50), ylim = c(30, 70))
```



We will learn more options on how to specify display windows in the second part of the workshop.

Exercise 3

Loading and filtering the worldmap dataset

See hints in exercise 2.

Grouping and counting

You can use the function `group_by()` from the package `dplyr` to group the dataset and then pass the groups to the function `count()` to count observations in each group. For example, if you want to group the dataset `msleep` by variable `vore` and count the observations in each group, you can do so as follows:

```
library(dplyr)
library(ggplot2)  # for the "msleep" example dataset

group_by(msleep, vore) %>% count()
```

```
## # A tibble: 5 x 2
## # Groups:   vore [5]
##   vore        n
##   <chr>   <int>
```

```
## 1 <NA>        7
## 2 carni      19
## 3 herbi      32
## 4 insecti     5
## 5 omni       20
```

You can also group by several variables, for example by each combination of the variables `vore` and `order` that exist in the dataset:

```
# 1. group and count by variables "vore" and "order"
# 2. ungroup and show only a sample of 5 rows
group_by(msleep, vore, order) %>% count() %>%
    ungroup() %>% sample_n(5)
```

```
## # A tibble: 5 x 3
##   vore  order               n
##   <chr> <chr>           <int>
## 1 herbi Artiodactyla        5
## 2 herbi Pilosa              1
## 3 herbi Perissodactyla      3
## 4 <NA>  Soricomorpha        1
## 5 omni  Didelphimorphia     1
```

For the given task, you must group the observations by city **and** city longitude/latitude.

Restricting the display window

See hints in exercise 2.

Exercise 4

When loading the `bln_plr_sozind_data.csv` dataset, make sure that the variable `SCHLUESSEL` is loaded as character string, **not** as integer (use `colClasses = c('SCHLUESSEL' = 'character')` in `read.csv()`).

After loading the spatial dataset `bln_plr.geojson` make sure to set the CRS: `st_crs(<DATASET>) <- 25833`.

More information on the `bln_plr_sozind_data.csv` dataset:

- source: Berlin Senate Dept. for Urban Dev. and Housing, *Monitoring Soziale Stadtentwicklung 2017* via FIS-Broker
- variables:
  - `STATUS1`: Unemployment rate 2016 in percent
  - `STATUS2`: Long term unemployment rate 2016 in percent
  - `STATUS3`: Pct. of households that obtain social support ("Hartz IV") 2016
  - `STATUS4`: Portion of children under 15 living in household that obtains social support ("Hartz IV") 2016
  - `DYNAMO1` to 4: Change in the above indicators from the previous year

Exercise 5

After loading the spatial dataset `nutsrg_2_2016_epsg3857_20M.json` make sure to set the CRS: `st_crs(<DATASET>) <- 3857`.

More information on the `tgs00010_unempl_nuts2.csv` dataset:

- source: Eurostats / Regions & cities
- variables:
  - `sex`: `F` means unemployment rate for women, `M` for men, `T` for both
  - `nuts`: NUTS level-2 region code
  - `year`: year when the data was collected
  - `unempl_pct`: unemployment rate in percent

In case you want to use a different Eurostats dataset or a different NUTS map, you can download these resources here:

- for the datasets: https://ec.europa.eu/eurostat/data/browse-statistics-by-theme
- for the NUTS maps: https://github.com/eurostat/Nuts2json

Sources for geo-data

R packages

The following packages come directly with geo-data or provide means to download them programmatically:

- maps: World, USA, US states, US counties and more
- mapdata: World in higher resolution, China, Japan and more
- rnaturalearth: *R package to hold and facilitate interaction with natural earth vector map data.* → see next section
- OpenStreetMap: Access to the OpenStreetMap API → see next section

Natural Earth Data

naturalearthdata.com: *Natural Earth is a **public domain map dataset** available at 1:10m, 1:50m, and 1:110 million scales. Featuring tightly integrated vector and raster data, with Natural Earth you can make a variety of visually pleasing, well-crafted maps with cartography or GIS software.*

Provides vector data for:

- countries and provinces, departments, states, etc.
- populated places (capitals, major cities and towns)
- physical features such as lakes, rivers, etc.

You can either download the data directly from the website or use the package rnaturalearth.

Open Street Map

- provides even more detail than *Natural Earth Data*: streets, pathways, bus stops, metro lines, etc.
- GeoFabrik provides downloads of the raw data
- is much harder to work with b/c of the complexity of the data

OSM Admin Boundaries Map: web-service to download administrative boundaries worldwide for different levels in different formats (shapefile, GeoJSON, etc.); contains meta-data (depending on country) such as AGS in Germany

This wiki article explains which OpenStreetMap administrative boundary levels correspond to which regional level in Germany (e.g. level 6 corresponds to “Kreise”).

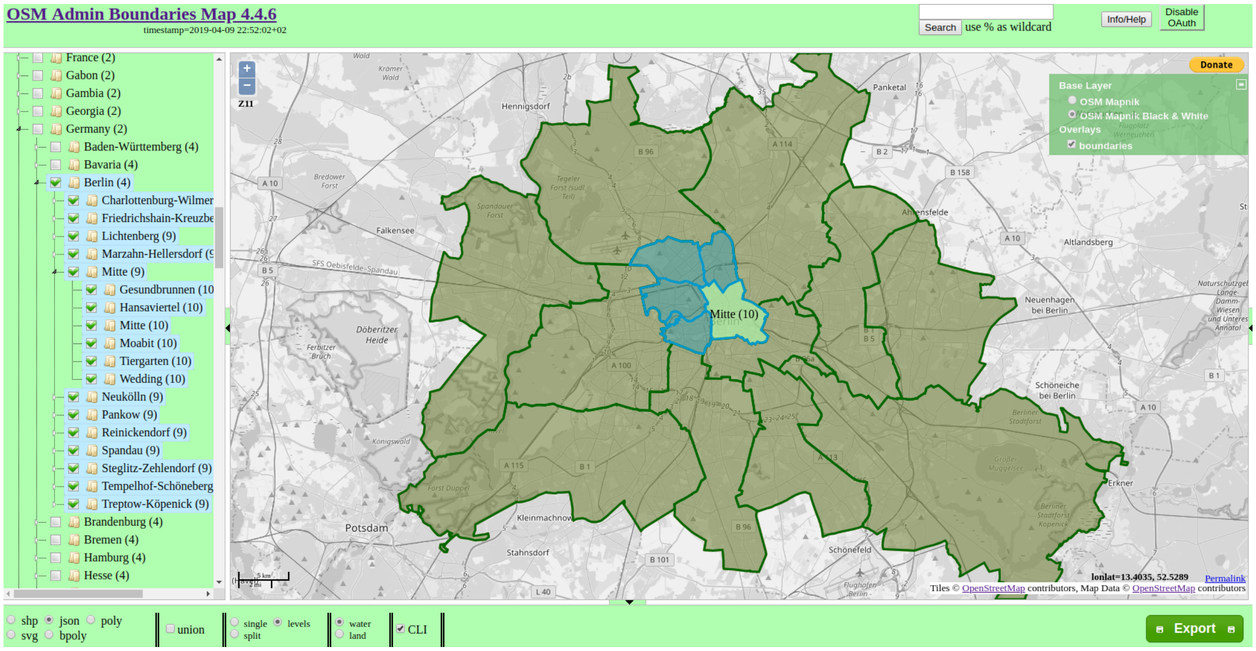


Figure 3: OSM Admin Boundaries screenshot

Administrative authorities in the EU

Administrative authorities often provide geo-data. In the EU, the main source is Eurostat which provides data referenced by NUTS code.

- main NUTS datasets as SHP, GeoJSON, TopoJSON, SVG
- Nuts2json provides another overview for GeoJSON and TopoJSON datasets
- correspondence tables map national structures and postcodes to NUTS regions

Administrative authorities in Germany

Statistisches Bundesamt provides geo-referenced data, such as:

- Gemeindeverzeichnis: AGS, area, population, etc.
- Regionaldatenbank: GDP, building land value, etc.
- govdata.de: Open data portal for Germany – lots of data, but not very well curated and documented

Berlin:

- Senate Department for Urban Development and Housing for example provides datasets based on LOR units
- FIS Broker is a web-service providing all publicly available geo-referenced data – this post shows how to use it

What about historical data?

Geographic areas such as administrative borders change. Identifiers may change, too. Make sure to use the version that matches your dataset!

- *Eurostat* provides historical NUTS areas back to 2003
- *Statistisches Bundesamt* also provides an archive

Glossary

AGS: *Amtlicher Gemeindeschlüssel* – municipality identificator in Germany.

CRS: Coordinate reference system – defines the coordinate system (spherical, ellipsoid, cartesian, etc.), unit of measurment (degrees, meters, etc.) and map projection of points in a spatial dataset in order to locate geographical entities

CRAN: *Comprehensive R Archive Network* – repository of packages that extend the statistical software suite R.

EPSG: *European Petroleum Survey Group* – a scientific organization tied to European petroleum industry. Created the *EPSG Geodetic Parameter Set*, which among other things contains a database of →CRS identified by EPSG →SRID code

ETRS89: *European Terrestrial Reference System 1989* – EU-recommended frame of reference for geodata for Europe; defines a →CRS.

GIS: *Geographic information system* – a system such as a software like →QGIS designed to work with geographic data.

Lat / Latitude: Geographic coordinate that defines the north-south position of a point on Earth as an angle between -90° (south pole) and 90° (north pole). The equator is located at 0° latitude.

Lon / Long / Lng / Longitude: Geographic coordinate that defines the east-west position of a point on Earth as an angle between -180° (westward) and 180° (eastward). The Prime Meridian is located at 0° longitude.

LOR: *Lebensweltlich orientierte Räume* – structures the city area of Berlin into sub-regions at three different levels; each area is identified by a LOR code.

NUTS: *Nomenclature of Territorial Units for Statistics* – divides the EU territory into regions at 3 different levels for socio-economic analyses of the regions; each area is identified by a NUTS code.

QGIS: free and open-source →GIS application.

SRID: *Spatial Reference System Identifier* – identifies a →CRS by a unique code number which is listed in the →EPSG database. Because of this, it is often also called EPSG code or number. Examples: EPSG:4326 refers to →WGS84; EPSG:4258 refers to →ETRS89.

SRS: *Spatial Reference System* – see →CRS.

WGS84: *World Geodetic System 1989* – defines a →CRS at global scale. Coordinates are defined in degrees as →longitude and →latitude.