Tutorial - Spatial Analysis in R with Open Geodata

Egge-Jan Pollé - Tensing GIS Consultancy B.V.
Willy Tadema - Provincie Groningen

Version 0.7.7 - September 6, 2018

Introduction

This is the manual for a tutorial session to be delivered at the 6th International Conference on the Use of R in Official Statistics (uRos2018) at the Dutch Office for National Statistics (CBS) in the Hague.

- Date: **Wednesday, September 12, 2018**
- Time: **9:00 - 12:30**
- Location: **Tinbergenzaal** (at CBS, Henri Faasdreef 312, 2492JP The Hague, The Netherlands)

Traning data

All datasets used in this training manual do come from publicly accessible sources - so, are really *Open Data*. Loading those data directly into R is part of the exercises. At the time of writing all links used in this book to access the data were valid.

Course material available on GitHub

This training manual is available on our GitHub repository - both in **PDF** and in **Rmd** file format.

The whole project is a work in progress, so comments, suggestions, code samples and/or useful links are always welcome: just drop us a line or fork us on GitHub.

Prerequisites

- Attendees should bring their own laptop with (64-bit) R, and preferably also RStudio, installed.
- No specific prior experience (with R) is required for this introductory session, though some basic experience in one or more of the following fields is certainly helpful: Data Science, Programming/Software Development and/or GIS.

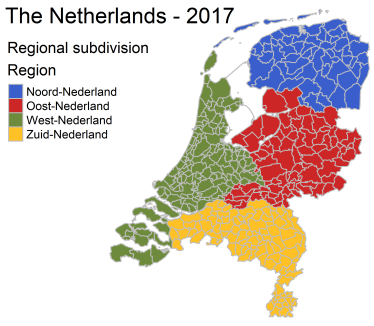
Packages to install

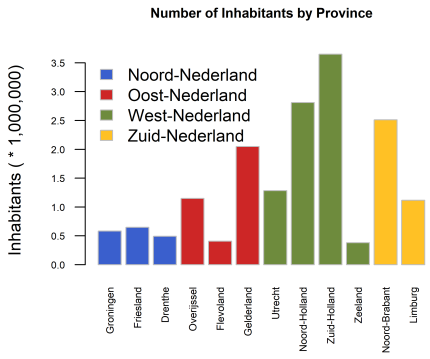
These are the packages you will at least need during the course:
`cbsodataR`, `data.table`, `dplyr`, `httr`, `knitr`, `jsonlite`, `mapview`, `sf`, `stringr`, `tmap` and `units`.

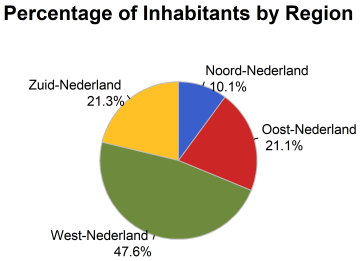
Pro tip: use the function `purl()` to extract all the R code from the manual:

```
library(knitr)
purl("Tutorial Spatial Analysis in R with Open Geodata - uRos2018.Rmd")
```

This will definitely save you a lot of copying and pasting :-)







Contents

Managing Geospatial Vector Data in R

1 The package `sf` (Simple Features for R)

To manage spatial data in R in this manual we will be using the library `sf`, a relativle new addition to the R universe. The package was released on CRAN in January 2017.

This package provides support for simple features, which is a standardized way to encode spatial vector data.

`sf` links directly to three important geospatial libraries, to unlock their power for use in R:

- GDAL for reading and writing data
- GEOS for geometrical operations
- Proj.4 for projection conversions and datum transformations.

The package `sf` on CRAN:
https://cran.r-project.org/package=sf

If the package is not yet installed, you can install it with the following command:

```
install.packages("sf")
```

To be able to manage your spatial data you will first have to load the package `sf`:

```
library(sf)
```

```
## Linking to GEOS 3.6.1, GDAL 2.2.3, proj.4 4.9.3
```

The real geospatial powers behind sf

- GDAL: the **Geospatial Data Abstraction Library** is a translator library for raster and vector geospatial data formats. Website: http://www.gdal.org/
- GEOS: the **Geometry Engine, Open Source** contains the complete functionality of the OpenGIS Simple Features for SQL spatial predicate functions and spatial operators. Website: https://trac.osgeo.org/geos
- Proj.4: **PROJ** is a generic coordinate transformation software, that transforms coordinates from one coordinate reference system (CRS) to another. This includes cartographic projections as well as geodetic transformations. Website: http://proj4.org/

1.1 `st_as_sf()`: converting a `data.frame` into an `sf` object

In this first exercise we will try to get some Dutch airports on the map. Let's start with defining some variables:

```
ap <- "Lelystad Airport"
cd <- "LEY"
class(cd)
```

```
## [1] "character"
```

```
lat <- 52.460278
lon <- 5.527222
class(lon)
```

```
## [1] "numeric"
```

Now we will combine these variables into a `data.frame`:

```
airport <- data.frame(ap, cd, lat, lon, stringsAsFactors = FALSE)
```

Wait, we have found a CSV file with some more airports:

```
NL_Airports <-
  read.csv("http://www.twiav.nl/files/NL_Airports.csv", stringsAsFactors = FALSE)

NL_Airports
```

```
##                       airport iata latitude longitude
## 1  Amsterdam Airport Schiphol  AMS 52.30833  4.760833
## 2 Rotterdam The Hague Airport  RTM 51.95000  4.433333
## 3     Groningen Airport Eelde  GRQ 53.12500  6.583333
## 4           Eindhoven Airport  EIN 51.45028  5.374444
## 5   Maastricht Aachen Airport  MST 50.91583  5.776944
```

Let's add our single airport to this dataset:

```
NL_Airports <- rbind(NL_Airports, airport)
```

```
## Error in match.names(clabs, names(xi)): names do not match previous names
```

```
# Oh yeah, of course: this will generate an error...
```

We can easily solve this issue setting the names equal to eachother:

```
names(airport) <- names(NL_Airports)
```

Now we can `rbind` the two data.frames:

```
NL_Airports <- rbind(NL_Airports, airport)
NL_Airports
```

```
##                       airport iata latitude longitude
## 1  Amsterdam Airport Schiphol  AMS 52.30833  4.760833
## 2 Rotterdam The Hague Airport  RTM 51.95000  4.433333
## 3     Groningen Airport Eelde  GRQ 53.12500  6.583333
## 4           Eindhoven Airport  EIN 51.45028  5.374444
## 5   Maastricht Aachen Airport  MST 50.91583  5.776944
## 6            Lelystad Airport  LEY 52.46028  5.527222
```

```
class(NL_Airports)
```

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

Now, how do we convert this `data.frame` into an `sf` object? We have got information on the latitude and the longitude of our airports, and we can use this to create geometry, in this case: points.

Please note: the Coordinate Reference System (CRS) of these lat\lon coordinates is WGS84, which has been given the EPSG code 4326. In chapter 7 we will have a closer look at spatial reference systems.

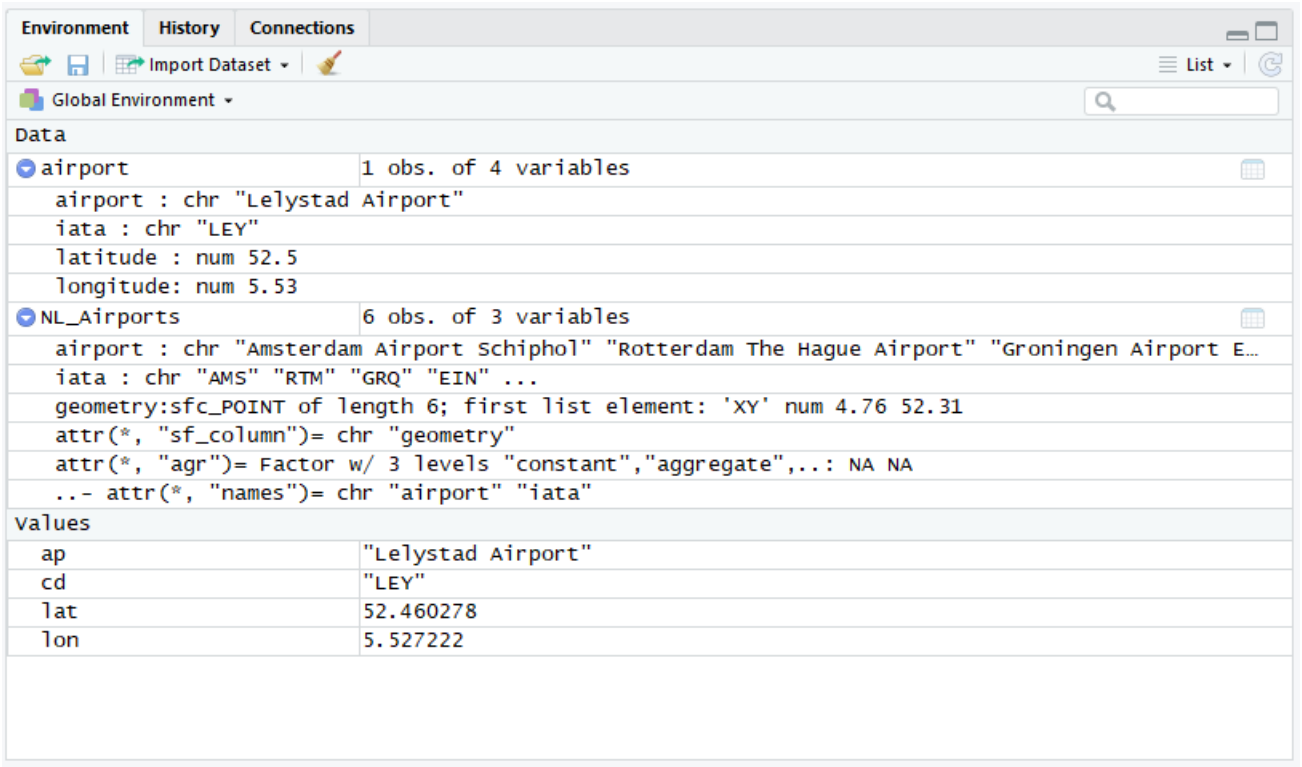
Now we can use the function `st_as_sf()` to convert our `data.frame` - with `longitude` first:

```
library(sf)

NL_Airports <- st_as_sf(NL_Airports, coords = c("longitude","latitude"), crs = 4326)

class(NL_Airports)
```

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

Our dataset is now both a `data.frame` and an `sf` object, with all the conveniences which come with this dual status!

Figure 1: Our first **sf** object in the **RStudio Environment pane**



And when we plot this relatively simple dataset to the screen we can see the sheer beauty of our **Simple Features**: the original columns containing the lat\lon information have been used to create a column containing points. And this geometry column is stored next to the attribute data, in the very same `data.frame`. Wonderful, isn't it?

```
NL_Airports
```

```
## Simple feature collection with 6 features and 2 fields
## geometry type:  POINT
## dimension:      XY
## bbox:           xmin: 4.433333 ymin: 50.91583 xmax: 6.583333 ymax: 53.125
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
##                       airport iata                  geometry
## 1  Amsterdam Airport Schiphol  AMS POINT (4.760833 52.30833)
## 2 Rotterdam The Hague Airport  RTM    POINT (4.433333 51.95)
## 3     Groningen Airport Eelde  GRQ   POINT (6.583333 53.125)
## 4           Eindhoven Airport  EIN POINT (5.374444 51.45028)
## 5   Maastricht Aachen Airport  MST POINT (5.776944 50.91583)
## 6            Lelystad Airport  LEY POINT (5.527222 52.46028)
```

Here we are talking only points, but in later exercises we will also see geometry columns containing lines and polygons.

Now we can plot our airports like this:

```
plot(st_geometry(NL_Airports), main = "Airports in the Netherlands", pch = 17)
```

1.2 st_write()

To share the spatial data in a simple features object with non-R users we have to write it to a spatial database or file format with the function `st_write()`

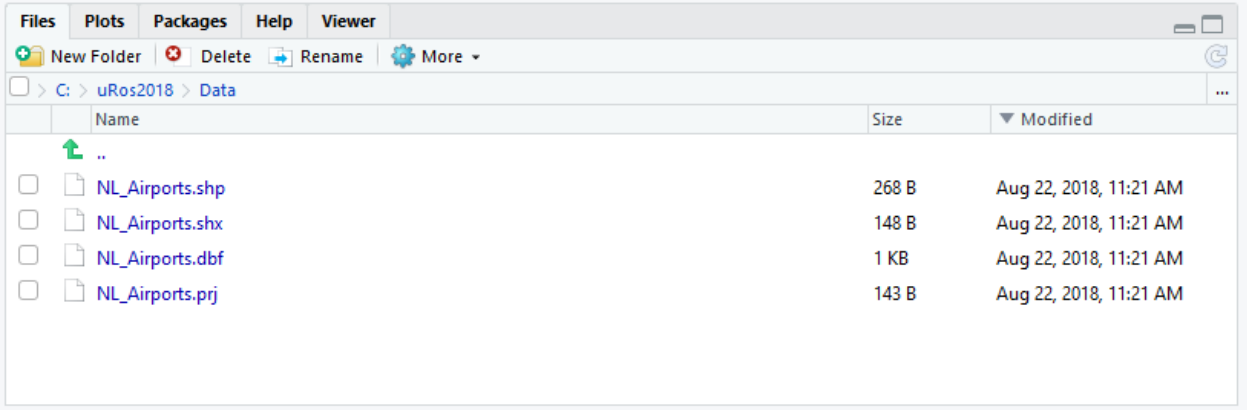
1.2.1 Export data to an ESRI Shapefile

A well-known format to write to is the ESRI Shapefile format (see Figure 2):

```
st_write(NL_Airports, "NL_Airports.shp")
```

```
## Writing layer `NL_Airports' to data source `NL_Airports.shp' using driver `ESRI Shapefile'
## features:       6
## fields:         2
## geometry type:  Point
```

Figure 2: The shapefile with the corresponding files are written to your Working Directory



The ESRI Shapefile

The shapefile format is a well-known and still rather popular geospatial vector data format for Geographic Information System (GIS) software. It spatially describes vector features - points, lines, and polygons - with attribute data attached. The shapefile is a 'classic' GIS file format in the sense that it stores geometry and attribute data in separate files (in this chapter we will discover that a single shapefile in reality consists of multiple files) as opposed to more modern spatial database or file formats, where geometry and attributes are stored together in a single table or file.

The shapefile format has been developed by Esri and over the years has become a *de facto* standard for data interoperability among Esri and other GIS software products.

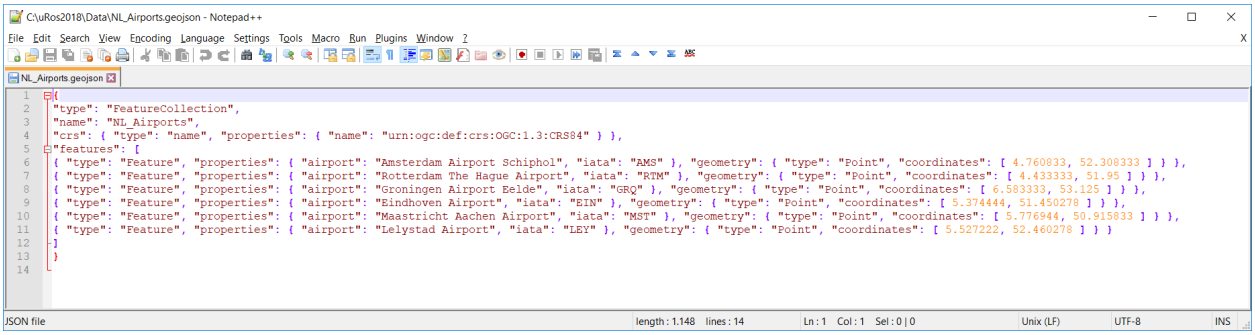
1.2.2 Export data to a GeoJSON file

Another format we may use is GeoJSON (see Figure 3):

```
st_write(NL_Airports, "NL_Airports.geojson")
```

```
## Writing layer `NL_Airports' to data source `NL_Airports.geojson' using driver `GeoJSON'
## features:       6
## fields:         2
## geometry type:  Point
```

Figure 3: The GeoJSON file can be viewed in a text editor



1.3 What about sp? (Our suggestion: Forget it!)

When you start googling about spatial analysis with R, you will surely tumble into information about the package sp. So why would we not use that one? The answer is simple: sp is the predecessor of sf.

The package sp provides classes and methods for spatial data and yes, for many years, it were these classes and methods you had to use to get your spatial data into R. For a long time, sp was at the heart of everything spatial in R. But those days are over now, as sf is rapidly taking over.

No, sp is not entirely obsolete yet. It is still actively maintained and you can find it on CRAN:

The package sp on CRAN:
https://cran.r-project.org/package=sp

So, how can we be so sure about our suggestion to try not to use sp then?

That has to do with the way the geometry is handled in both packages. In a way you could compare the way the data is stored using the sp package with the Esri shapefile mentioned in paragraph 1.2.1: in an sp Spatial* object the attribute data and the geometry are stored separately, whereas in an sf object the geometry is stored in a column in the dataframe itself (see paragraph 1.1).

Please note: the sf package is created by Eder Pebesma, together with Roger Bivand. And Pebesma and Bivand are also the main driving forces behind sp. So, also in that sense sf is really a succesor - and not a competitor - to sp. Apparently we had to go through this sp phase first, just like we needed the Esri Shapefile once...

It is Pebesma himself who stated in his first sf vignette: "The package sf aims at succeeding sp in the long term." And developments in that direction are going lightning-fast.

So, let's look forward

1.3.1 No further reading

Throughout this manual you will find suggestions for further reading, but for the title below we really suggest **no** further reading:

- **Bivand, Roger S., Edzer Pebesma and Virgilio Gómez-Rubio** (2013), *Applied Spatial Data Analysis with R*. Second Edition. Springer-Verlag New York.

Why? This book was the standard source on spatial data analysis with R, written by the creators of the `sp` package, written in a time that thinking about `simple features` had barely begun. If you manage to lay your hand on a copy, we would surely recommend you to browse through it, if only for historical reasons. But as explained above, your main focus for now should be the future-proof package `sf`.

1.4 Further reading

The big announcement:

- **Pebesma, Edzer** (January 3, 2017). *Simple Features Now on CRAN*. Blog. Retreived from: https://www.r-consortium.org/blog/2017/01/03/simple-features-now-on-cran

The `sf` vignettes:

- Simple Features for R
- Reading, Writing and Converting Simple Features
- Manipulating Simple Feature Geometries
- Manipulating Simple Features
- Plotting Simple Features
- Miscellaneous

2 Interactive Viewing of Spatial Data in R

In the previous paragraph we have seen a static map plotted on the **Plots** pane. But as a real Data Scientist we also want to be able to explore our data on an interactive map. Until a few years ago this would have meant switching back and forth between a desktop GIS and R. But in recent years some new packages have been developed to enable interactive map viewing in R.

In this paragraph we will look at two of these.

2.1 The package `mapview`

The package `mapview` on CRAN:
https://cran.r-project.org/package=mapview

If the package is not yet installed, you can install it with the following command:

```
install.packages("mapview")
```

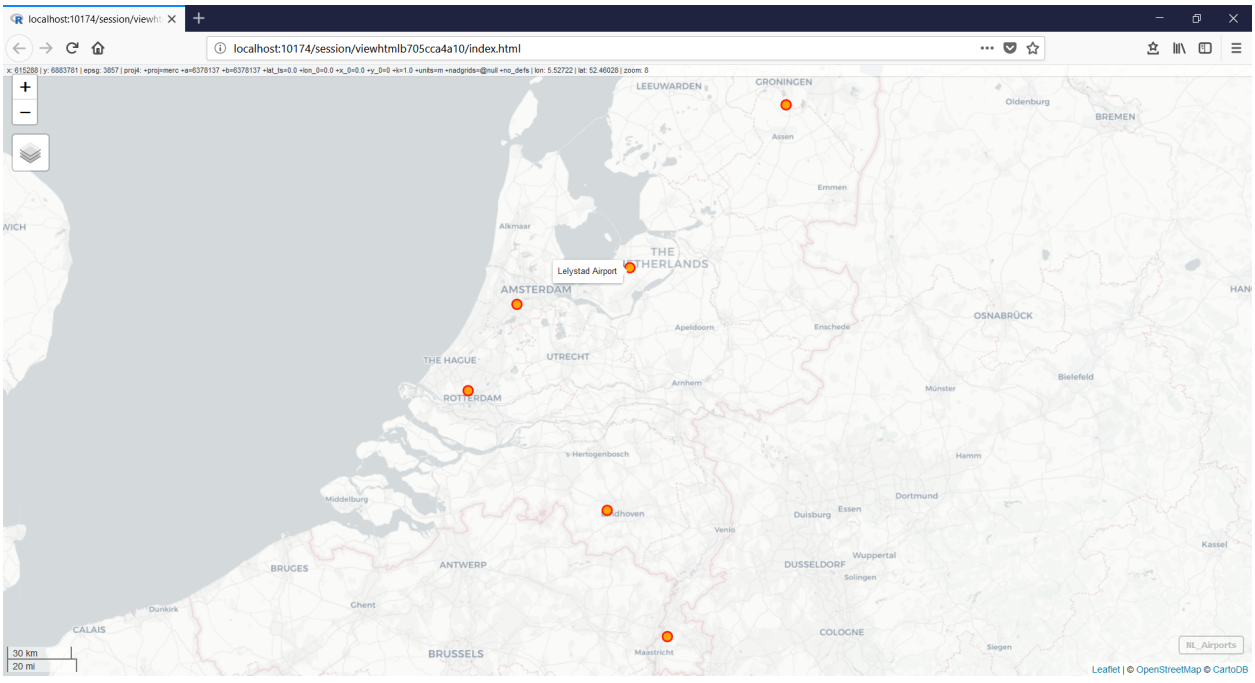
To be able to view your spatial data interactively you will first have to load the package `mapview`:

```
library(mapview)
```

Now we can open up an interactive map using a the function `mapview()` (see Figure 4):

```
mapview(NL_Airports, color = "red", col.regions = "orange", alpha.regions = 1, label =
↪  NL_Airports$airport)
```

Figure 4: An interactive map can be opened up using a simple mapview() statement



2.2 The package tmap

The package tmap on CRAN:
https://cran.r-project.org/package=tmap

If the package is not yet installed, you can install it with the following command:

```
install.packages("tmap")
```

To be able to view your spatial data interactively you will first have to load the package tmap:

```
library(tmap)
```

2.2.1 More on tmap I: Workshop Plotting spatial data in R

This afternoon you will be able to see much more thematic mapping as Martijn Tennekes, the author of the tmap package, will deliver his workshop **Plotting spatial data in R**.

- Date: Wednesday, September 12, 2018
- Time: 13:30 - 17:00
- Location: this very same Tinbergenzaal

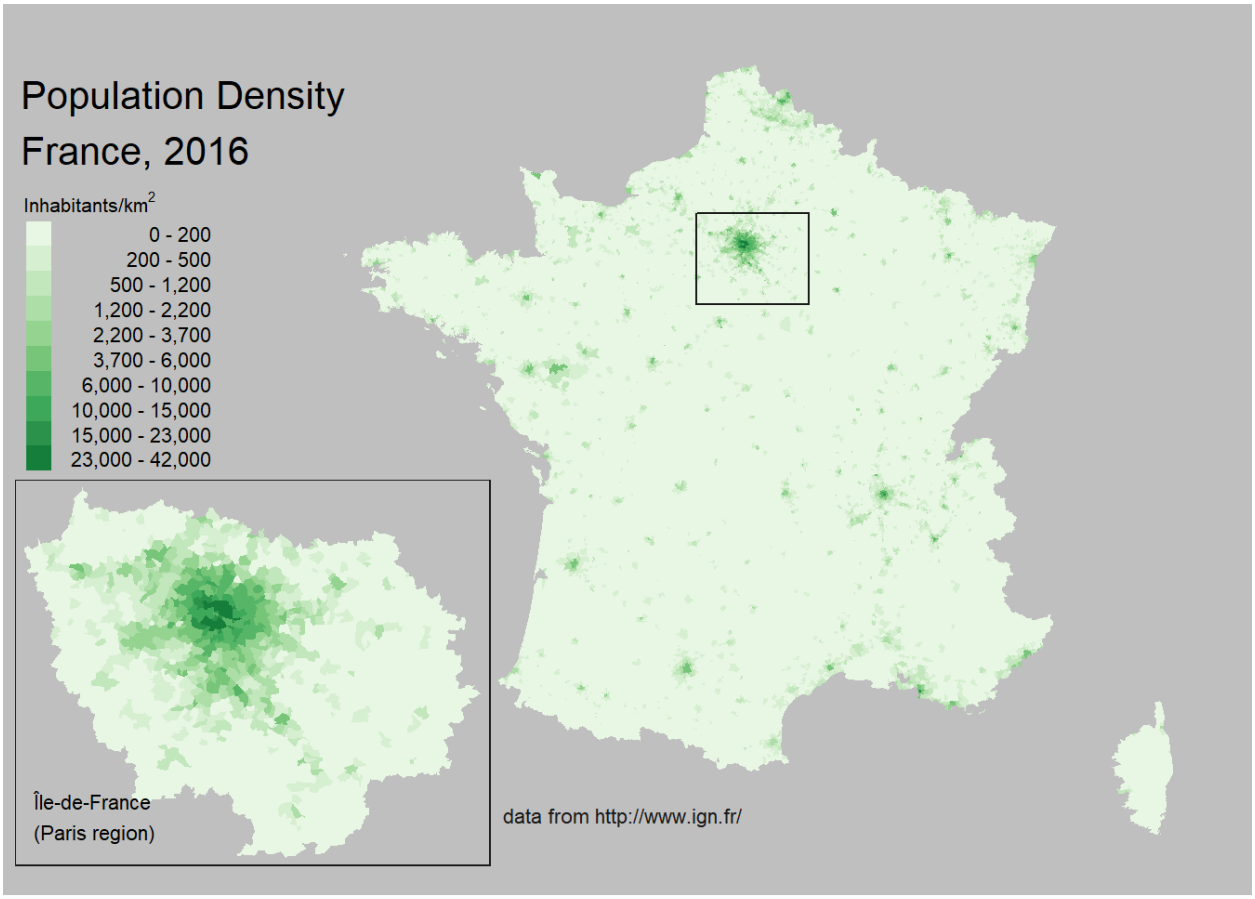
Workshop materials can be found here: https://github.com/mtennekes/tmap-workshop

2.2.2 More on `tmap` II: Visualizing population density in France (2016)

The thematic map below (see Figure 5) shows the population density in France in 2016, with an additional inset for *Île-de-France*, the region around Paris, as almost 20 percent of the French population lives there.

The code to recreate this map can be found in this blog: `tmap`: quick and easy thematic mapping in R

Figure 5: An example of a thematic map created with tmap



2.3 Further reading

The `mapview` vignettes:

- 1. mapview basics
- 2. mapview advanced controls
- 3. mapview options
- 4. mapview popups
- 5. extra functionality
- 6. extra leaflet functionality
- 7. ceci constitue la fin du pipe

The `tmap` vingette:

- tmap: get started!

Accessing Geospatial Vector Data over the Internet

3 Downloadable shapefiles

In this chapter we will learn how to download, to unzip and to load shapefiles **using R**. We have seen the ESRI Shapefile before in paragraph 1.2.1. There will be no need to use your web browser, your file explorer or a zip utility - the whole process can be completed using just a few lines of R code.

It is not uncommon for public organizations - including national statistical institutes - to distribute geographic information in this shapefile format. So, you will find shapefiles on the page with geographic data of the Dutch Office for National Statistics (CBS) and also on the STATISTIK AUSTRIA open.data Portal.

3.1 Download and unzip the shapefile

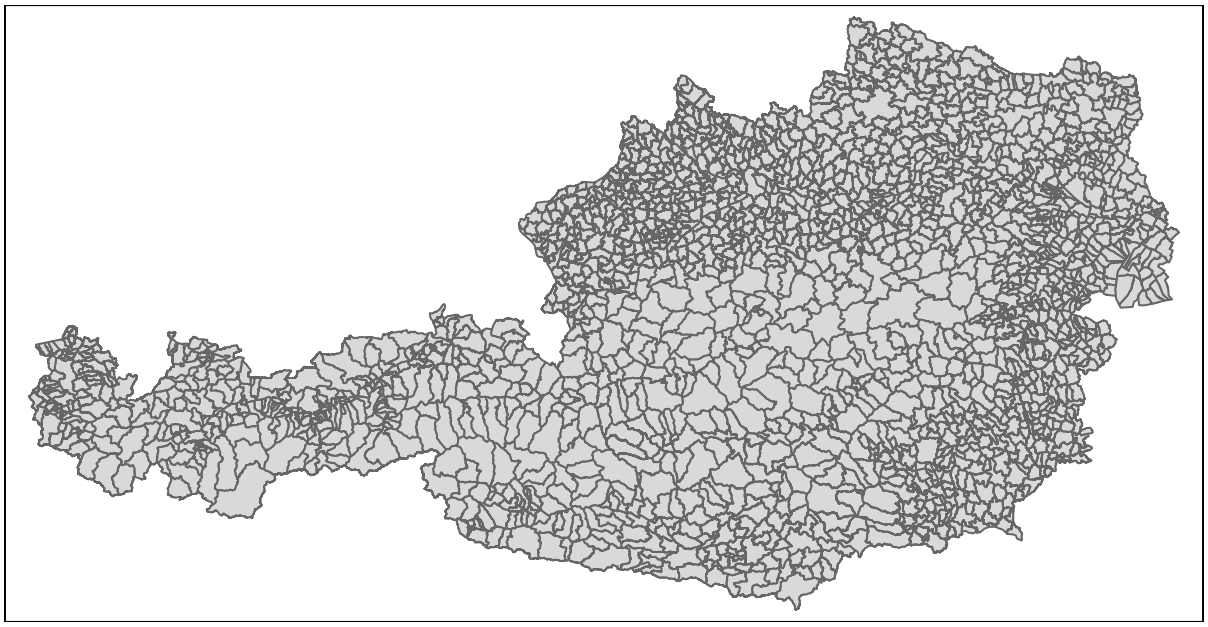
```
# Store the URL to the file to download in a variable
URL2zip <- "http://data.statistik.gv.at/data/OGDEXT_GEM_1_STATISTIK_AUSTRIA_20180101.zip"

# Create a temporary file
zip_file <- tempfile(fileext = ".zip")

# Download the file
download.file(URL2zip, destfile = zip_file, mode = "wb")

# Create a subfolder in your working directory to store the unzipped data
dir.create("./Data", showWarnings = FALSE)

# Unzip the file
unzip(zip_file, exdir = "./Data")

# After unzipping you can delete (i.e. unlink) the file
unlink(zip_file)

# Remove variables you do not longer need
rm(URL2zip, zip_file)
```

3.2 Load the shapefile

```
library(sf)
library(tmap)
AUSTRIA_GEM_20180101 <- st_read("./Data/STATISTIK_AUSTRIA_GEM_20180101.shp")
```

```
## Reading layer `STATISTIK_AUSTRIA_GEM_20180101' from data source `C:\GitHub\uRos2018\Spatial_Analysis
## Simple feature collection with 2120 features and 2 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 112518.2 ymin: 275472 xmax: 685444.5 ymax: 570431.1
## epsg (SRID):    31287
## proj4string:    +proj=lcc +lat_1=49 +lat_2=46 +lat_0=47.5 +lon_0=13.33333333333333 +x_0=400000 +y_0=4
1.3145,-5.393,-2.3887 +units=m +no_defs
```

```
qtm(AUSTRIA_GEM_20180101)
```



4 OGC Web Feature Service (WFS)

4.1 Introduction

In this chapter we will learn how **to use R as a client to access data using a Web Feature Service (WFS)**. WFS is a Data Access Standard which is defined and maintained by the Open Geospatial Consortium (OGC). The WFS Interface Standard defines a set of interfaces for accessing geographic information over the Internet. It offers the means to retrieve geographic features and their properties through a highly configurable interface and in a manner independent of the underlying data stores they publish.

The standard is used - both in the public and private sector and in the academic world - to publish vector geospatial datasets in a way that makes it easy for receiving organisations to conduct analysis on the data supplied.

In a short chapter like this it will not be possible to cover all ins and outs of the WFS Interface Standard. The main goal here is to get you up and running and to whetten your appetite for more. It might all look a bit technical and intimidating at first, but as soon as you have the syntax of your request right, you will be able to retrieve valuable spatial data in the blink of an eye.

4.2 WFS Outputformat: GML vs. GeoJSON

By default, a WFS returns data in Geography Markup Language (GML) which is written as eXtensible Markup Language (XML). However, many WFS services also offer the option to request the ouput in GeoJSON, a geospatial data interchange format based on JavaScript Object Notation (JSON).

By it's very nature, GML data is difficult to process, because - as with most XML based grammars - there are two parts to the grammar: the schema that describes the document and the instance document that contains the actual data.

On the contrary, the GeoJSON standard clearly defines several types of JSON objects and the manner in which they are combined to represent data about geographic features, their properties, and their spatial extents.

In general **GDAL**, the translator library behind the `sf` functions `st_read()` and `st_write` (see chapter 1), gives better results with GeoJSON as opposed to GML.

So, in the exercises in this manual, when accessing a WFS service to retrieve data, we will always add the parameter `outputFormat=application/json`.

4.3 Access a WFS service: `request=GetCapabilities`

In general, organisations publishing data using WFS, will provide you with a URL to their WFS server which also contains a `GetCapabilities` request. Once you know the **capabilities** of the service (i.e. once you know what is on offer) you can start building your own request to the server.

In the exercises below we will use data on the municipal division in the Netherlands. These data are offered by the host of this conference, the Dutch statistical office, through Publieke Dienstverlening op de Kaart (PDOK) and the Nationaal Georegister (NGR).

Some background information on the dataset used, can be found here: Dataset: CBS Gebiedsindelingen.

The actual URL with the `GetCapabilities` reguest is:
https://geodata.nationaalgeoregister.nl/cbsgebiedsindelingen/wfs?request=GetCapabilities

When you click this link the server response will be shown in your browser in XML format.

We are not going to study this XML response line-by-line. Not now. But please leave the web page open in your browser for later reference.

We will have a closer look at the capabilities document in paragraph 4.7

4.4 Retrieve data from a WFS service: `request=GetFeature`

4.4.1 `request=GetFeature` - an example from the Netherlands

A WFS server responding to a *GetFeature* request returns a collection of geographic feature instances filtered according to a criteria set by the requesting client. We will start with a simple request to download a full feature collection without any constraints to filter the content by. The GetFeature request queries the server with a set of parameters, which are concatenated to the URL with an *ampersand* (&).

In the script below we use the `httr` package. This allows us to store the different parameters of our request in a list, only to build the full URL at the end. We do so for readability reasons and to allow for easy modification of our request at a later stage if necessary. And the function `build_url()` will return a properly encoded URL.

A WFS service can offer one or more feature collections, see the <FeatureTypeList> section in the XML response to the `GetCapabilities` request. The service we are accessing here offers quite some feature collections, i.e. multiple regional divisions for the years 1995 up to the current year. What we are interested in here, is the municipal division for the year 2017. After some browsing through the XML response, we have found a <FeatureType> with the <Name> **cbsgebiedsindelingen:cbs_gemeente_2017_gegeneraliseerd**. And that's the value we are giving to the `typename` parameter. (This `typenames` parameter determines the collection of feature instances to return.)

Also, do not forget to add a parameter to ask for output in GeoJSON format (as discussed in paragraph 4.2).

This is the script to populate the full request:

```
# NL: Example with Dutch data
library(sf)
library(tmap)
library(httr)
library(data.table)

url <- list(hostname = "geodata.nationaalgeoregister.nl/cbsgebiedsindelingen/wfs",
            scheme = "https",
            query = list(service = "WFS",
                         version = "2.0.0",
                         request = "GetFeature",
                         typename =
                           "cbsgebiedsindelingen:cbs_gemeente_2017_gegeneraliseerd",
                         outputFormat = "application/json")) %>%
       setattr("class","url")
request <- build_url(url)
```

The variable `request` will now contain this link. (When you click this link the server response will be shown in your browser in GeoJSON format.)

Now we want to feed this response directly into R, like this:

```
NL_Municipalities2017 <- st_read(request)
```

```
## Reading layer `OGRGeoJSON' from data source `https://geodata.nationaalgeoregister.nl/cbsgebiedsindeli
## Simple feature collection with 388 features and 5 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 13565.4 ymin: 306846.9 xmax: 277992.8 ymax: 619291
## epsg (SRID):    28992
## proj4string:    +proj=sterea +lat_0=52.15616055555555 +lon_0=5.38763888888889 +k=0.9999079 +x_0=1550
0.406857,0.350733,-1.87035,4.0812 +units=m +no_defs
```

Have a look at the result:

```
qtm(NL_Municipalities2017)
```

```
head(NL_Municipalities2017)
```

```
## Simple feature collection with 6 features and 5 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 226868.9 ymin: 562754.2 xmax: 276821.5 ymax: 604994.7
## epsg (SRID):    28992
## proj4string:    +proj=sterea +lat_0=52.15616055555555 +lon_0=5.38763888888889 +k=0.9999079 +x_0=1550
0.406857,0.350733,-1.87035,4.0812 +units=m +no_defs
##                                                          id statcode
## 1 cbs_gemeente_2017_gegeneraliseerd.fid-7cb4d6d8_165ba79caf5_46cd   GM0003
## 2 cbs_gemeente_2017_gegeneraliseerd.fid-7cb4d6d8_165ba79caf5_46ce   GM0005
## 3 cbs_gemeente_2017_gegeneraliseerd.fid-7cb4d6d8_165ba79caf5_46cf   GM0007
## 4 cbs_gemeente_2017_gegeneraliseerd.fid-7cb4d6d8_165ba79caf5_46d0   GM0009
## 5 cbs_gemeente_2017_gegeneraliseerd.fid-7cb4d6d8_165ba79caf5_46d1   GM0010
## 6 cbs_gemeente_2017_gegeneraliseerd.fid-7cb4d6d8_165ba79caf5_46d2   GM0014
##       statnaam jrstatcode  rubriek                      geometry
## 1   Appingedam 2017GM0003 gemeente MULTIPOLYGON (((254580.7 59...
## 2        Bedum 2017GM0005 gemeente MULTIPOLYGON (((235432.1 59...
## 3 Bellingwedde 2017GM0007 gemeente MULTIPOLYGON (((276518.1 56...
## 4     Ten Boer 2017GM0009 gemeente MULTIPOLYGON (((245194.7 59...
## 5     Delfzijl 2017GM0010 gemeente MULTIPOLYGON (((262016.6 58...
## 6    Groningen 2017GM0014 gemeente MULTIPOLYGON (((243244.4 58...
```

4.4.2 `request=GetFeature` - an example from Finland

In this paragraph we will retrieve data from Tilastokeskus (Statistics Finland). In the English section of their website we found a nice dataset: Population by municipality-based units.

And from this map it is especially the layer **Population 2017 by municipalities 2018 (kunta_vaki2017)** we would like to investigate.

We will follow the same steps as in the Dutch example above. For an explanation of the steps taken, please refer to paragraph 4.3 and 4.4.1.

The URL with `GetCapabilities` request is:
http://geo.stat.fi/geoserver/vaestoalue/wfs?request=GetCapabilities

In this XML response - in the `<FeatureTypeList>` section - we have found a `<FeatureType>` with the `<Name>` **vaestoalue:kunta_vaki2017**. That's the value we are giving to the `typename` parameter.

The script to build the full URL is similar to the one before - the only parameters we have modified are the hostname and the typename:

```
# FI: Example with Finnish data
library(sf)
library(tmap)
library(httr)
library(data.table)

url <- list(hostname = "geo.stat.fi/geoserver/vaestoalue/wfs",
            scheme = "https",
            query = list(service = "WFS",
                         version = "2.0.0",
                         request = "GetFeature",
                         typename = "vaestoalue:kunta_vaki2017",
                         outputFormat = "application/json")) %>%
       setattr("class","url")
request <- build_url(url)
```

The variable `request` will now contain this link. (When you click this link the server response will be shown in your browser in GeoJSON format.)

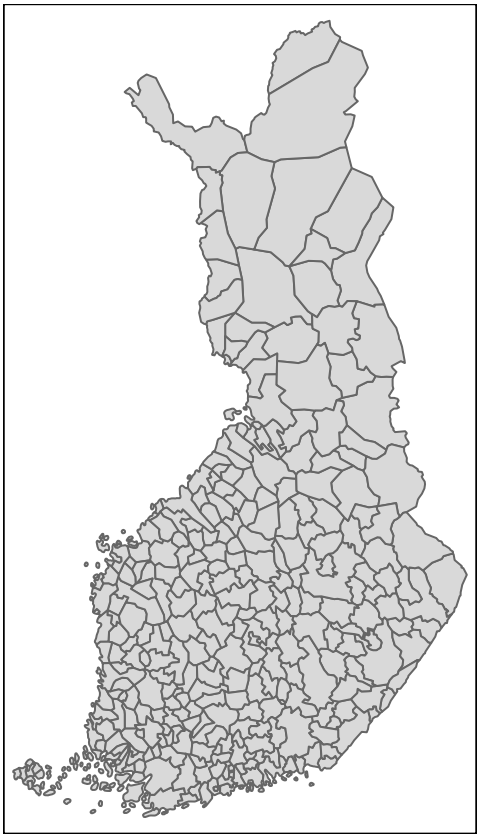
Now we want to feed this response directly into R, like this:

```
FI_Municipalities2018_Pop2017 <- st_read(request)
```

```
## Reading layer `OGRGeoJSON' from data source `https://geo.stat.fi/geoserver/vaestoalue/wfs/?service=WF
## Simple feature collection with 311 features and 20 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 83747.59 ymin: 6637032 xmax: 732907.8 ymax: 7776431
## epsg (SRID):    3067
## proj4string:    +proj=utm +zone=35 +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m +no_defs
```

Have a look at the result:

```
qtm(FI_Municipalities2018_Pop2017)
```



Optionally, you can have a look at the attribute data in the **Data Viewer**: `View(FI_Municipalities2018_Pop2017)`

With some basic knowledge of the Finnish language you should now be able to calculate that Finland had more or less 5.5 million inhabitants in 2017 - of which more or less 50 percent are males, and the other half females:

Finnish for Data Scientists: some useful words

- **Miehet** = Men
- **Naiset** = Women
- **Nimi** = Name
- **Vaesto** = Population

```
sum(FI_Municipalities2018_Pop2017$vaesto)
```

```
## [1] 5513130
```

```
sum(FI_Municipalities2018_Pop2017$miehet)
```

```
## [1] 2719131
```

```
sum(FI_Municipalities2018_Pop2017$naiset)
```

```
## [1] 2793999
```

4.5 Get a description of a dataset: `request=DescribeFeatureType`

In the previous exercises we immediately executed `GetFeature` requests to WFS services - mainly because the trainer told us to do so - without actually knowing anything about these datasets.

To get a description of the dataset, you can execute a `DescribeFeatureType` request first. This returns a description - in XML format - of the structure, including properties, of the feature type specified in the request.

For the Dutch municipalities dataset this request would look like this: https://geodata.nationaalgeoregister.nl/cbsgebiedsindelingen/wfs?service=WFS&version=2.0.0&request=DescribeFeatureType&typename=cbsgebiedsindelingen:cbs_gemeente_2017_gegeneraliseerd

And for the Finnish dataset like this: http://geo.stat.fi/geoserver/vaestoalue/wfs?service=WFS&version=2.0.0&request=DescribeFeatureType&typename=vaestoalue:kunta_vaki2017

4.6 Additional parameters to the `GetFeature` request

Additional parameters can be added to a `GetFeature` request to further filter or convert the response from the WFS.

To include additional parameters to a request, simply add an *ampersand* (&) at the end of the URL, then add the name of the parameter, an equal sign (=) and the value to assign to the parameter. Of course we will not do this manually; we will use the `httr` package to build this URL with additional parameters for us.

Below we will discuss a few of the parameters available.

4.6.1 Limit the number of records returned: the `count` parameter

To have a look at the structure of a dataset - before retrieving the full dataset - you can choose to limit the number of records returned with the `count` parameter.

The `GetFeature` request below limits the number of Finnish municipalities returned to only 5:

```
# FI: Example with Finnish data
library(httr)
library(data.table)

url <- list(hostname = "geo.stat.fi/geoserver/vaestoalue/wfs",
            scheme = "https",
            query = list(service = "WFS",
                         version = "2.0.0",
                         request = "GetFeature",
                         typename = "vaestoalue:kunta_vaki2017",
                         count = 5,
                         outputFormat = "application/json")) %>%
       setattr("class","url")
request <- build_url(url)
```

The variable `request` will now contain this link. (When you click this link the server response will be shown in your browser in GeoJSON format.)

4.6.2 Limit the number of columns returned: the `PropertyName` parameter

To restrict a `GetFeature` request by attribute rather than feature, use the `PropertyName` parameter. You can specify a single attribute, or multiple attributes separated by commas.

The `GetFeature` request below only returns the geometry and the columns *nimi* (Name) and *vaesto* (total population) for the Finnish municipalities:

```
# FI: Example with Finnish data
library(httr)
library(data.table)

url <- list(hostname = "geo.stat.fi/geoserver/vaestoalue/wfs",
            scheme = "https",
            query = list(service = "WFS",
                         version = "2.0.0",
                         request = "GetFeature",
                         typename = "vaestoalue:kunta_vaki2017",
                         propertyname = "geom,nimi,vaesto",
                         outputFormat = "application/json")) %>%
       setattr("class","url")
request <- build_url(url)
```

The variable `request` will now contain this link. (When you click this link the server response will be shown in your browser in GeoJSON format.)

4.7 Advanced topics

In paragraph 4.3 we have had a quick look at the answer to a `GetCapabilities` request. Here we will dive a little deeper into the information we can retrieve from this capabilities document.

4.7.1 outputFormat

Not every WFS service supports GeoJSON as an output format. So how can we retrieve the output formats that are available? To answer this question we take a closer look at the capabilities document.

```
library(magrittr)
library(httr)
library(data.table)
library(xml2)

url <- list(hostname = "geodata.nationaalgeoregister.nl/cbsgebiedsindelingen/wfs",
            scheme = "https",
            query = list(service = "WFS",
                         version = "2.0.0",
                         request = "GetCapabilities")) %>%
       setattr("class","url")
request <- build_url(url)
doc <- GET(request) %>% content(as = "text", encoding="UTF-8") %>% read_xml()
xpath <- paste0("//ows:Operation[@name='GetFeature']",
                "/ows:Parameter[@name='outputFormat']",
                "/ows:AllowedValues/ows:Value")
output_formats <- doc %>% xml_find_all(xpath) %>% xml_text()
output_formats
```

```
##  [1] "text/xml; subtype=gml/3.2"
##  [2] "GML2"
##  [3] "KML"
##  [4] "application/gml+xml; version=3.2"
##  [5] "application/json"
##  [6] "application/vnd.google-earth.kml xml"
##  [7] "application/vnd.google-earth.kml+xml"
##  [8] "csv"
##  [9] "gml3"
## [10] "gml32"
## [11] "json"
## [12] "text/xml; subtype=gml/2.1.2"
## [13] "text/xml; subtype=gml/3.1.1"
```

As we can see, GML and GeoJSON are not the only supported formats for this specific WFS. The service can also return the query result in a CSV or KML file.

4.7.2 FeatureType

To get a list of all available feature types in a service, we can also inspect the capabilities document. Think of a feature type as a layer or dataset.

```
xpath  <- "//wfs:FeatureType/wfs:Name"
feature_types  <- doc %>% xml_find_all(xpath) %>% xml_text()
head(feature_types)
```

```
## [1] "cbsgebiedsindelingen:cbs_arbeidsmarktregio_2014_gegeneraliseerd"
## [2] "cbsgebiedsindelingen:cbs_arbeidsmarktregio_2014_labelpoint"
## [3] "cbsgebiedsindelingen:cbs_arbeidsmarktregio_2015_gegeneraliseerd"
## [4] "cbsgebiedsindelingen:cbs_arbeidsmarktregio_2015_labelpoint"
## [5] "cbsgebiedsindelingen:cbs_arbeidsmarktregio_2016_gegeneraliseerd"
## [6] "cbsgebiedsindelingen:cbs_arbeidsmarktregio_2016_labelpoint"
```

4.7.3 maxRecordCount

Often the number of records returned by a service per request is limited to prevent performance issues caused by heavy requests. Therefore we should always check the maximum record count per request.

```
xpath <- "//ows:Constraint[@name='CountDefault']/ows:DefaultValue"
maxRecordCount <- doc %>% xml_find_first(xpath) %>% xml_integer()
maxRecordCount
```

```
## [1] 15000
```

So a request to the cbsgebiedsindelingen WFS will never return more than 15,000 records.

To determine the number of hits before we actually fetch the query result, we can add `resultType=hits` to our `GetFeature` request:

```
url$query = list(service = "WFS",
                 version = "2.0.0",
                 request = "GetFeature",
                 typename =
                  ↪  "cbsgebiedsindelingen:cbs_arbeidsmarktregio_2014_gegeneraliseerd",
                 resultType = "hits") %>%
            setattr("class","url")
request <- build_url(url)
doc <- doc <- GET(request) %>% content(as = "text", encoding="UTF-8") %>% read_xml()
xpath <- "//wfs:FeatureCollection/@numberMatched"
hits <- doc %>% xml_find_first(xpath) %>% xml_integer()
hits
```

```
## [1] 35
```

In this case the number of hits (35) is smaller than the maximum record count per request (15,000). So we can fetch the entire dataset with one request.

Let's put it all together in another example and query the Dutch Adresses en Buildings Register (Basisregistratie Adressen en Gebouwen) for the adresses in Hoek van Holland, a seaside town not far from The Hague.

```
url <- list(hostname = "geodata.nationaalgeoregister.nl/bag/wfs",
            scheme = "https",
            query = list(service = "WFS",
                         version = "2.0.0",
                         request = "GetCapabilities")) %>%
            setattr("class","url")
request <- build_url(url)
doc <- GET(request) %>% content(as = "text", encoding="UTF-8") %>% read_xml()
xpath <- "//wfs:FeatureType/wfs:Name"
feature_types <- doc %>% xml_find_all(xpath) %>% xml_text()
feature_types
```

```
## [1] "bag:ligplaats"       "bag:pand"            "bag:standplaats"
## [4] "bag:verblijfsobject" "bag:woonplaats"
```

```
xpath <- "//ows:Constraint[@name='CountDefault']/ows:DefaultValue"
maxRecordCount <- doc %>% xml_find_first(xpath) %>% xml_integer()
maxRecordCount
```

```
## [1] 1000
```

```
url$query = list(service = "WFS",
                 version = "2.0.0",
                 request = "GetFeature",
                 typename = "bag:verblijfsobject",
                 cql_filter = "bag:woonplaats='Hoek van Holland'",
                 resultType = "hits") %>%
            setattr("class","url")
request <- build_url(url)
doc <- GET(request) %>% content(as = "text", encoding="UTF-8") %>% read_xml()
xpath <- "//wfs:FeatureCollection/@numberMatched"
hits <- doc %>% xml_find_first(xpath) %>% xml_integer()
hits
```

```
## [1] 6473
```

Notice that we added `cql_filter=bag:woonplaats='Hoek van Holland'` to the request to filter out only adresses in Hoek van Holland.

Now the number of hits (6,473) is greater than the maximum record count per request. We have hit the server limit. So how do we get all adresses if a request will not return more than 1,000? The answer is simple: create more requests. This is called paging.

4.7.4 Paging

For paging the parameters `sortBy`, `startIndex` and `count` come in handy. By adding `sortBy=bag:identificatie` to our request we ensure that the features returned, are sorted by their globally unique identifier. `startIndex` specifies the index of the first record. `count` is the number of records that the request should try to fetch. It is identical to the maximum record count.

Because we know the maximum record count and the number of hits, we can calculate the number of requests required to fetch all records in the query result. By increasing the `startIndex` for each subsequent request, we can eventually retrieve all adresses in Hoek van Holland.

```
library(sf)

url$query <- list(service = "wfs",
                  version = "2.0.0",
                  request = "GetFeature",
                  typename = "bag:verblijfsobject",
                  cql_filter = "woonplaats='Hoek van Holland'",
                  outputFormat = "application/json",
                  resultType = "results",
                  count = maxRecordCount,
                  sortBy = "bag:identificatie")

requestAdresses <- function(x) {
  url$query$startIndex <- x
  request <- build_url(url)
  st_read(request, stringsAsFactors = FALSE)
}

adresses <- lapply(seq(0, hits, maxRecordCount), requestAdresses) %>% do.call(rbind, .)
```

```
nrow(adresses)
```

```
## [1] 6473
```

All 6473 adresses in Hoek van Holland are fetched!

To wrap it up, let's visualize these addresses in an interactive map (see Figure 6): .

```
library(tmap)

tmap_mode('view')
tm_shape(adresses) +  tm_dots(col = "black", scale = 0.1) +
  tm_legend(show = FALSE) + tm_view(basemaps = 'OpenStreetMap')
```

Figure 6: The BAG addresses in Hoek van Holland on an interactive map

4.8 Further reading

- The official OGC Web Feature Service (WFS) Implementation Specification can be found here: http://www.opengeospatial.org/standards/wfs
- The OGC offers an official tutorial module - OGC E-learning - covering it's activities and the several standards it maintains. The specific chapter on WFS can be found here.

Figure 7: On the overview page metadata about the **Feature Layer** *USA States (Generalized)* is presented

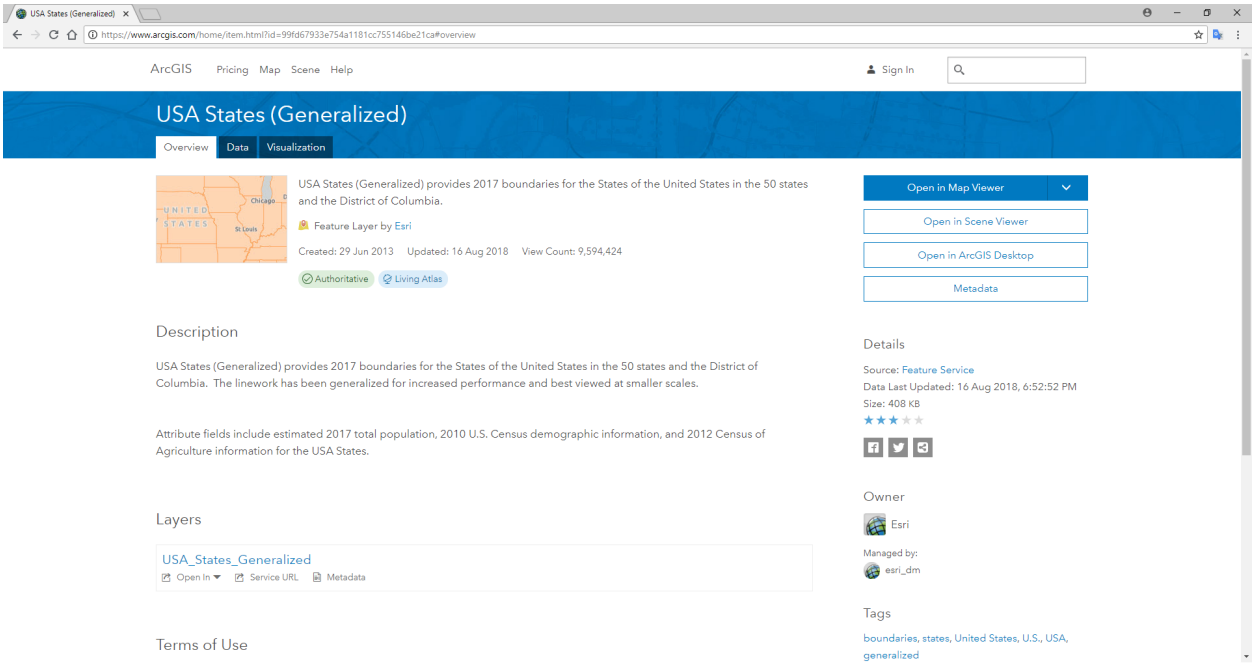
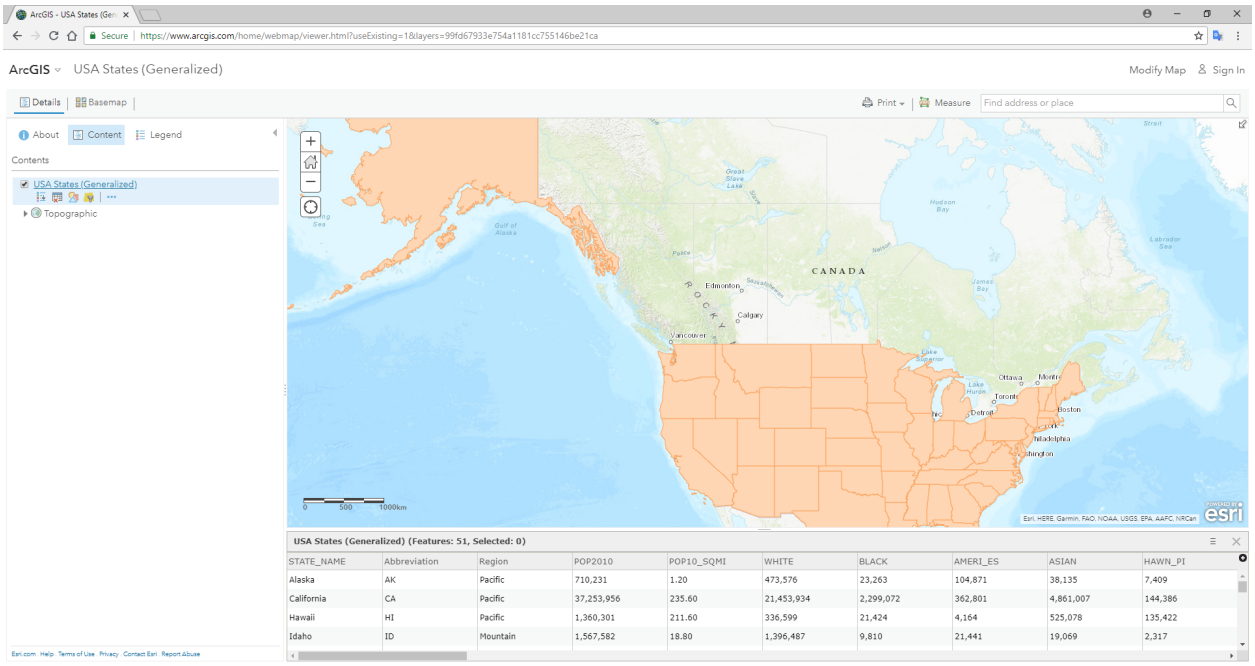


Figure 8: Here the **Feature Layer** *USA States (Generalized)* is shown in the **ArcGIS Online Map Viewer**



5 ArcGIS REST Service

5.1 Introduction

In this chapter we will learn how **to use R as a client to access data using an ArcGIS REST Service**. We will be accessing ArcGIS Server to retrieve Feature Layers using the ArcGIS REST API. This API is part of ArcGIS Online, the "complete SaaS mapping platform" by the American GIS company Esri. And as part of this platform Esri has launched the ArcGIS Living Atlas of the World[1]

Some datasets in this Living Atlas are Open Data. Some, but by no means all, as many datasets are only accessible via an ArcGIS Online organizational account. But in cases where the source data is already Open Data, Esri publishes the data under the same conditions.

5.2 Exploring data in the ArcGIS Living Atlas of the World

After some browsing through the content of the Living Atlas we have found a nice (and open) dataset for this exercise: the **Feature Layer** *USA States (Generalized)*. An overview page with metadata for the dataset we have chosen can be found via this link (see Figure 7).

Before actually importing the data into R, you can explore the dataset - both the geometry and the attribute data - in the ArcGIS Online Map Viewer (see Figure 8).

5.3 Query the Feature Service

On the overview page we have found the Service URL pointing to the layer **USA_States_Generalized** in the the Services Directory (see Figure 9):

https://services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services/USA_States_Generalized/FeatureServer/0

One of the **Supported Operations** for this layer is **Query**. The *Query* page (see Figure 10) offers an interface to construct the query:

https://services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services/USA_States_Generalized/FeatureServer/0/query

Let's try to build a request to retrieve the full dataset:

- The first parameter is `where`. It is obligatory and cannot be omitted. If there is no *where-clause* (i.e. if you want to extract the full dataset) you have to populate this parameter with the value `1=1` (which is - of course - always true…)
- To get all the attribute columns you have to give the parameter `outFields` the value `*`
- To retrieve the spatial data the parameter `returnGeomatry` should be set to `true`
- And the outputformat - `f` - is set to `geojson`

[1] Some background information on the Living Atlas of the World can be found in this article, where Esri also invites YOU to *"Become a user and a contibutor"*.

Figure 9: On this page the **Feature Layer** *USA_States_Generalized* is presented in the **ArcGIS REST Services Directory**

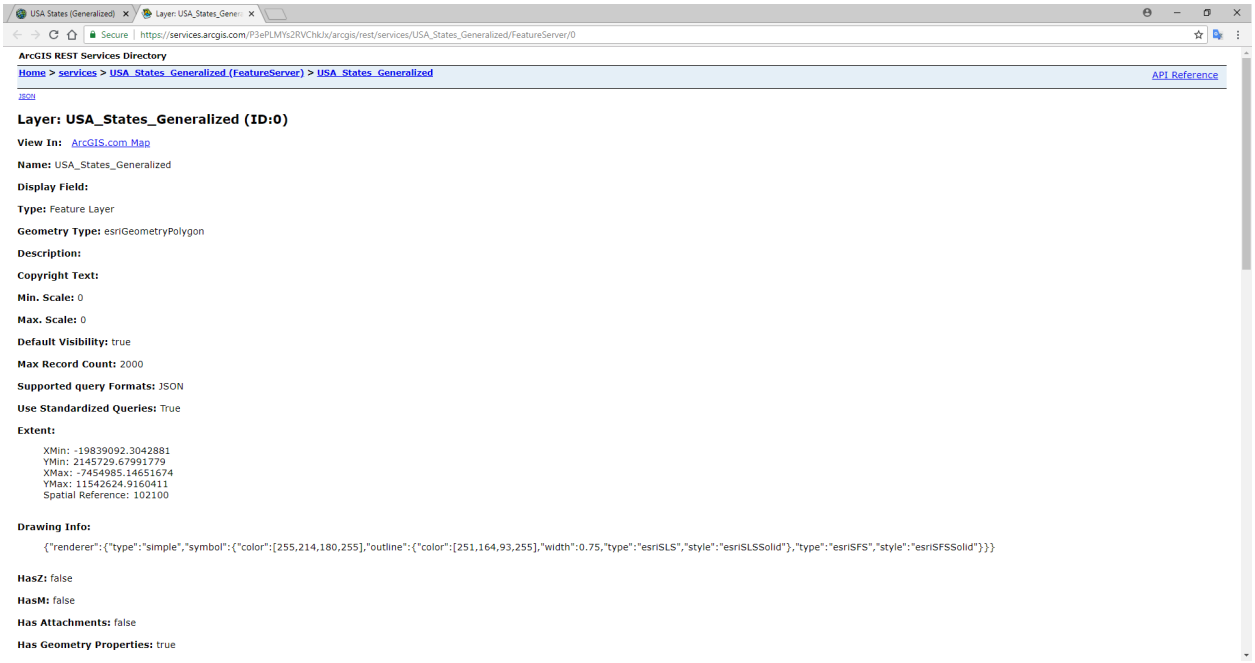
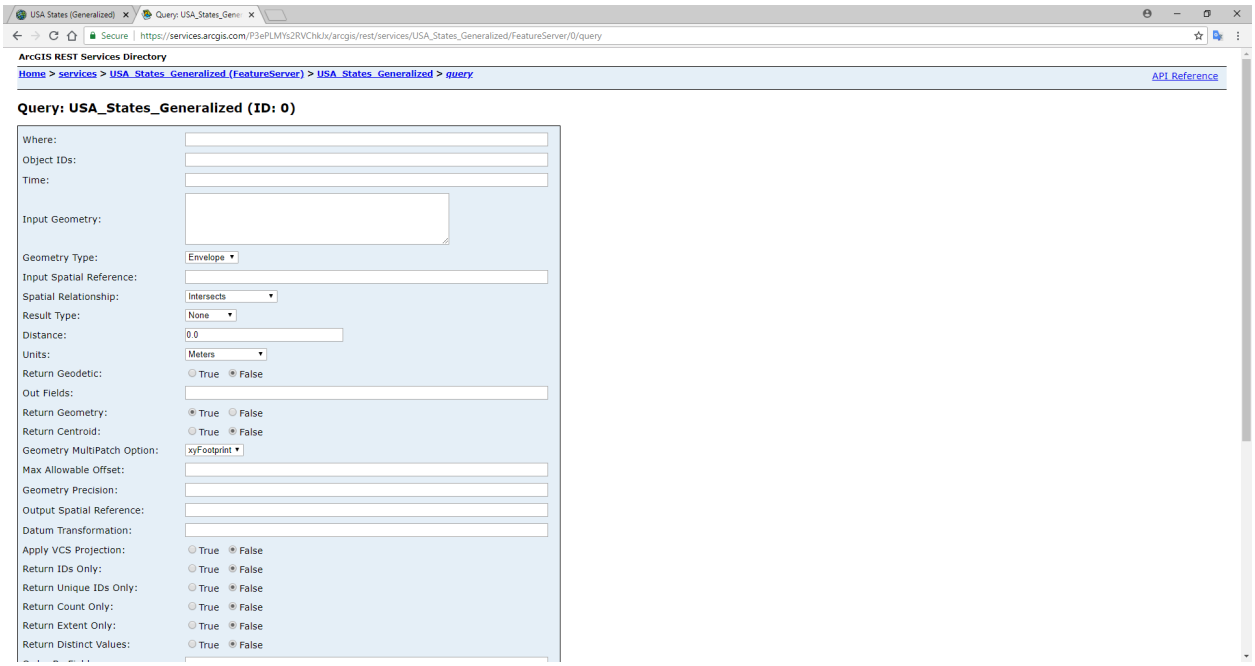


Figure 10: On this page you can enter parameters to query the **Feature Layer** *USA_States_Generalized*



This is the script to populate the full request:

```
# USA: An example with American data
library(sf)
library(tmap)
library(httr)
library(data.table)

url <- list(hostname = "services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services",
            scheme = "https",
            path = "USA_States_Generalized/FeatureServer/0/query",
            query = list(where = "1=1",
                         outFields = "*",
                         returnGeometry = "true",
                         f = "geojson")) %>%
       setattr("class","url")
request <- build_url(url)
```

The variable `request` will now contain this link. (When you click this link the server response will be shown in your browser in GeoJSON format.)

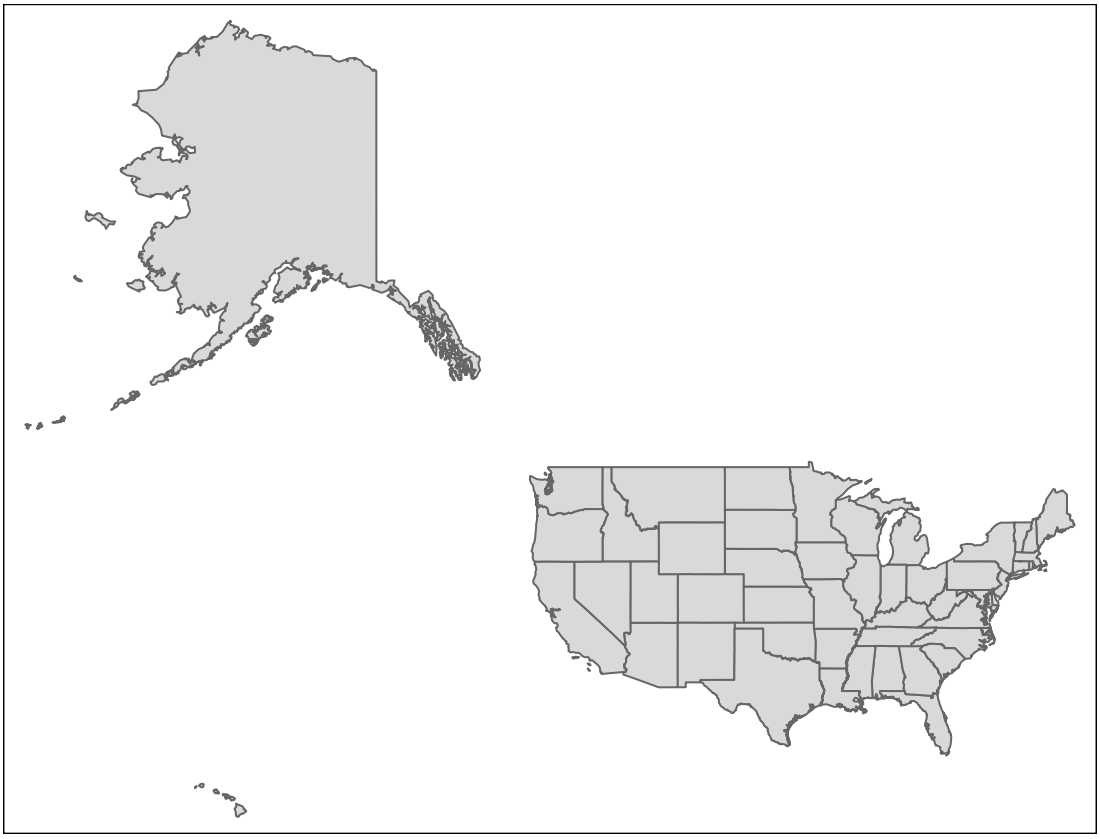
Now we want to feed this response directly into R, like this:

```
USA_States_2017 <- st_read(request)
```

```
## Reading layer `OGRGeoJSON' from data source `https://services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/res
## Simple feature collection with 51 features and 55 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -19839090 ymin: 2145730 xmax: -7454985 ymax: 11542620
## epsg (SRID):    3857
## proj4string:    +proj=merc +a=6378137 +b=6378137 +lat_ts=0.0 +lon_0=0.0 +x_0=0.0 +y_0=0 +k=1.0 +units
```

Have a look at the result:

```
qtm(USA_States_2017)
```



Anchorage

”Hey Chel you know it’s kinda funny
Texas always seems so big
But you know you’re in the largest State in the Union
When you’re anchored down in Anchorage”

Michelle Shocked - Short Sharp Shocked - 1988

5.4 Additional parameters to the `query` request

5.4.1 Filter the records returned: specify the `where`-clause

To filter records you can specify a `where`-clause.

The `query` request below only returns those States which have more than 10 million inhabitants:

```
# USA: An example with American data
library(httr)
library(data.table)

url <- list(hostname = "services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services",
            scheme = "https",
            path = "USA_States_Generalized/FeatureServer/0/query",
            query = list(where = "POPULATION>10000000",
                         outFields = "STATE_NAME,POPULATION",
                         returnGeometry = "true",
                         f = "geojson")) %>%
       setattr("class","url")
request <- build_url(url)
```

The variable `request` will now contain this link. (When you click this link the server response will be shown in your browser in GeoJSON format.)

5.4.2 Limit the number of columns returned: the `outFields` parameter

If you do not need all attribute columns of a dataset in your analysis, you can limit the number of columns returned by specifying the `outFields` parameter. You can specify a single attribute, or multiple attributes separated by commas.

The `query` request below only returns the geometry and the columns STATE_NAME and POPULATION for the United States:

```
# USA: An example with American data
library(httr)
library(data.table)

url <- list(hostname = "services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services",
            scheme = "https",
            path = "USA_States_Generalized/FeatureServer/0/query",
            query = list(where = "1=1",
                         outFields = "STATE_NAME,POPULATION",
                         returnGeometry = "true",
                         f = "geojson")) %>%
       setattr("class","url")
request <- build_url(url)
```

The variable `request` will now contain this link. (When you click this link the server response will be shown in your browser in GeoJSON format.)

5.5 Advanced topics

5.5.1 Capabilities of the hosted feature service

So what exactly are the capabilities of a hosted feature service? To answer this question we can take a look at the information on the service in the ArcGIS REST Services Directory.

Let's give it a try. We'll query the USA Rail Road dataset in ESRI's Living Atlas of the World. This is data from the Federal Railroad Administration (FRA) which is passed on by Esri.

The service URL is https://services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services/USA_Railroads_1/FeatureServer

A request to this URL will return the capabilities of the service:

```
library(httr)
library(data.table)
library(dplyr)
library(jsonlite)
library(sf)
```

```
url <- list(hostname = "services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services",
            scheme = "https",
            path = "USA_Railroads_1/FeatureServer",
            query = list(f = "json")) %>%
       setattr("class","url")
response <- build_url(url) %>% fromJSON()
```

Notice that the format of the response is JSON instead of GeoJSON (query parameter `f = "json"`).

The `response` is a list with all the capabilities. Take some time to have a look at it.

For example:

```
response$description
```

```
## [1] "USA Railroads is a comprehensive database of the nation's railway system at 1:24,000 to 1:100,0
```

```
response$copyrightText
```

```
## [1] "Federal Railroad Administration (FRA), Esri"
```

5.5.2 Layers

A hosted feature service can have multiple layers. Every layer has a name and an id. The id is necessary for constructing the appropriate URL to query the features.

The response contains the layer information needed:

```
response$layers$name
```

```
## [1] "USA Railroads"
```

The USA Rail Road hosted feature service only has one layer called *USA Railroads*.

```
layer_id <- response$layers %>% filter(name == 'USA Railroads') %>% select(id)
layer_id
```

```
##   id
## 1  0
```

The id of the USA Railroads layer is 0. We use this information to construct the correct value for the `path` variable.

```
layer_path <- paste("USA_Railroads_1/FeatureServer", layer_id, sep = "/")
layer_path
```

```
## [1] "USA_Railroads_1/FeatureServer/0"
```

5.5.3 maxRecordCount

The response also contains other valuable information like the maximum number of records returned per request:

```
response$maxRecordCount
```

```
## [1] 1000
```

5.5.4 Capabilities of an individual layer

We can get even more information on the capabilities of the railroad layer:

```
url <- list(hostname = "services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services",
            scheme = "https",
            path = layer_path,
            query = list(f = "json")) %>%
       setattr("class","url")
response <- build_url(url) %>% fromJSON()
```

For example information on the fields in the dataset:

```
response$fields %>% select(name, type)
```

```
##          name                       type
## 1    OBJECTID          esriFieldTypeOID
## 2      FRA_ID      esriFieldTypeInteger
## 3  FRA_REGION esriFieldTypeSmallInteger
## 4       STATE       esriFieldTypeString
## 5  SUBDIVISIO       esriFieldTypeString
## 6    RR_OWNER       esriFieldTypeString
## 7      TRACKS esriFieldTypeSmallInteger
## 8    NET_CODE       esriFieldTypeString
## 9    NET_DESC       esriFieldTypeString
## 10  PASSENGER       esriFieldTypeString
## 11      MILES       esriFieldTypeDouble
```

5.5.5 returnCountOnly

So now we know the query URL, the maximum number of records returned per request and the fieldnames in the dataset. We are ready to request the records!

But wait, all the railroads in the United States of America, wouldn't that be *a lot* of records? Let's find out, before we fetch them all. We can do this by specifying the query parameter `returnCountOnly = "true"`:

```
url <- list(hostname = "services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services",
            scheme = "https",
            path = paste(layer_path, "query", sep = "/"),
            query = list(where = "1=1",
                         returnCountOnly = "true",
                         f = "geojson")) %>%
       setattr("class","url")
request <- build_url(url)
response <- build_url(url) %>% fromJSON()
hits <- response$properties$count
hits
```

```
## [1] 173114
```

173,114 is way more that the maximum record count per request of 1,000!

5.5.6 Automatic paging (by GDAL)

Let's do a request to fetch the records.

```
url <- list(hostname = "services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services",
            scheme = "https",
            path = paste(layer_path, "query", sep = "/"),
            query = list(where = "1=1",
                         outFields = "OBJECTID, NET_DESC",
                         returnGeometry = "true",
                         f = "geojson")) %>%
       setattr("class","url")
request <- build_url(url)
railroads <- st_read(request)
```

```
## Reading layer `OGRGeoJSON' from data source `https://services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/res
## Simple feature collection with 173114 features and 2 fields
## geometry type:  LINESTRING
## dimension:      XY
## bbox:           xmin: -158.1522 ymin: 20.88018 xmax: -66.98391 ymax: 64.92615
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
```

What is the number of records in the railroads dataframe? One would expect, not more than 1,000.

```
nrow(railroads)
```

```
## [1] 173114
```

Isn't that strange? The dataframe contains all 173,114 records. How is this possible?
Well the GDAL library makes it very easy for us. It detects that multiple requests are needed to fetch all the records and does this automagically for us. So no need to think of response paging, GDAL already has the solution for that!

A lot of the railroads are labelled 'abandonned'. They need to be filtered out. Of course you can do this *after* you fetched the records, but this isn't very efficient. Let's change the parameter value of `where` in our request, so only railroads in use are included in the response.

```
url <- list(hostname = "services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services",
            scheme = "https",
            path = paste(layer_path, "query", sep = "/"),
            query = list(where = "NET_DESC <> 'Abandoned'",
                         outFields = "OBJECTID, NET_DESC",
                         returnGeometry = "true",
                         f = "geojson")) %>%
       setattr("class","url")
request <- build_url(url)
```

```
railroads_in_use <- st_read(request)
```

```
## Reading layer `OGRGeoJSON' from data source `https://services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/res
## Simple feature collection with 134436 features and 2 fields
## geometry type:  LINESTRING
## dimension:      XY
## bbox:           xmin: -158.1522 ymin: 20.88018 xmax: -67.26739 ymax: 64.92615
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
```

```
nrow(railroads_in_use)
```

```
## [1] 134436
```

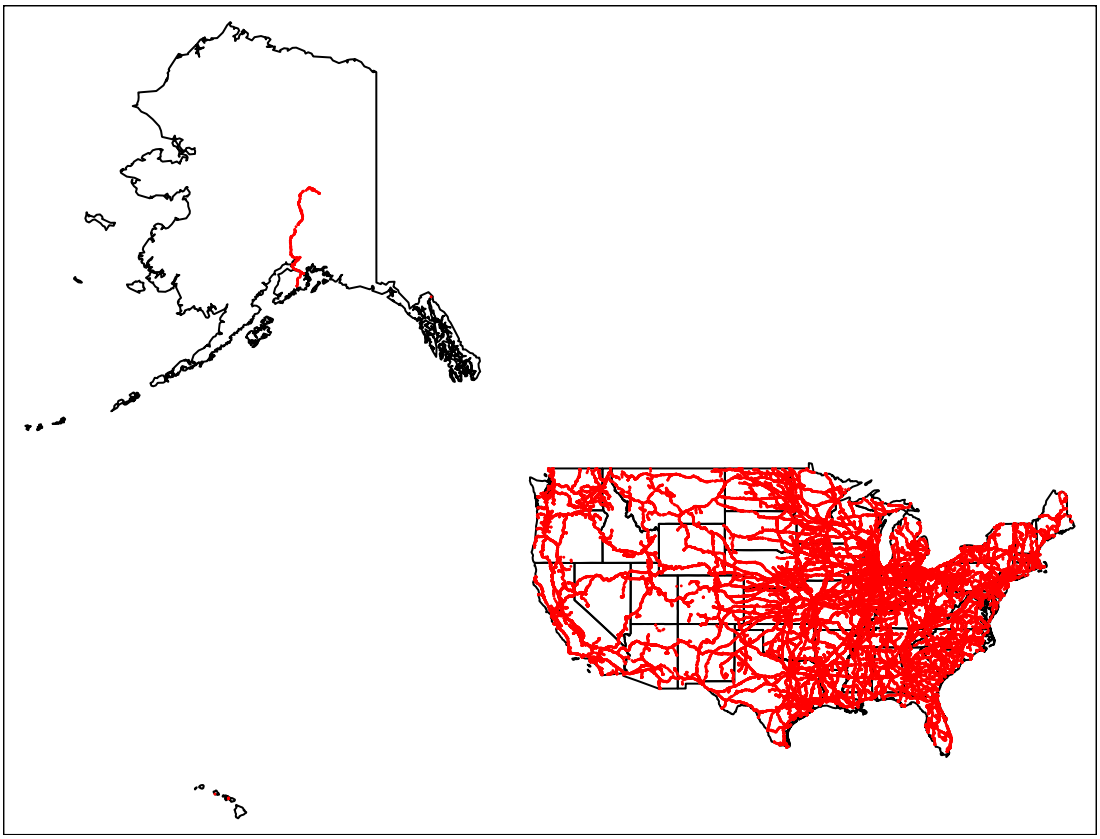
134,436 is still a lot of records, but let's have a look at the result and add the railroads on top of a map of the USA:

```
url <- list(hostname = "services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services",
            scheme = "https",
            path = "USA_States_Generalized/FeatureServer/0/query",
            query = list(where = "1=1",
                         returnGeometry = "true",
                         f = "geojson")) %>%
       setattr("class","url")
request <- build_url(url)
USA_States_2017 <- st_read(request)
```

```
## Reading layer `OGRGeoJSON' from data source `https://services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/res
## Simple feature collection with 51 features and 1 field
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -19839090 ymin: 2145730 xmax: -7454985 ymax: 11542620
## epsg (SRID):    3857
## proj4string:    +proj=merc +a=6378137 +b=6378137 +lat_ts=0.0 +lon_0=0.0 +x_0=0.0 +y_0=0 +k=1.0 +units
```

When we plot the rail network we do see a clear concentration in the Eastern part of the continent:

```
library(tmap)

tm_shape(USA_States_2017) +
  tm_borders("black") +
  tm_shape(railroads_in_use) +
  tm_symbols(col = "red", scale = 0.1, border.lwd = NA) +
  tm_legend(show = FALSE)
```



5.6 Further reading

- More on the **ArcGIS REST API** can be found at the ArcGIS for Developers pages: https://developers.arcgis.com/rest/

6 Statistics Netherlands (CBS) StatLine databank as open data

All tables in the Statistics Netherlands (CBS) StatLine databank are available as open data. Since 2014 this databank has an open data web API based on the OData protocol (https://www.odata.org/).

Using web services, data can be retrieved, filtered and combined. In this way, Statistics Netherlands aims to promote the widespread use of its statistical data.

In this chapter we will load Dutch statistical data directly into R, using the package 'cbsodataR', the Statistics Netherlands (CBS) Open Data API Client for R.

6.1 the package `cbsodataR`

The package `cbsodataR` can be found on CRAN: https://cran.r-project.org/package=cbsodataR

The table of contents can be retrieved with the function `cbs_get_toc()`. As we browse through the table of contents we do see a total of 4433 entries. For most datasets all the metadata is in Dutch, but for some of them this information is also available in English.

When we split the TOC by language we do see that currently 703 datasets are available with metadata in English. These 'English' datasets are actually copies of their Dutch equivalents. (Maybe overtime all tables will be available with a description in English?)

This is a wealth of information, and all these datasets can be directly loaded into R. In the next paragraph we will do some exercises with one particular table.

```
library(cbsodataR)

toc <- cbs_get_toc()
nrow(toc)
```

```
## [1] 4433
```

```
toc_nl <- cbs_get_toc(Language = "nl")
nrow(toc_nl)
```

```
## [1] 3730
```

```
toc_en <- cbs_get_toc(Language = "en")
nrow(toc_en)
```

```
## [1] 703
```

6.2 Exercises with regional statistical data

6.2.1 Preparing the data

The table we have chosen for the exercises in this paragraph is called 'Regionale kerncijfers Nederland', i.e. regional statistical data about the Netherlands. The identifier of this table is **70072ned**. It is a huge table containing data at 5 regional levels - from the country as a whole down to the municipal level - from 1995 up to the current year.

The table can be retrieved with the function `cbs_get_data()`:

```
NL_Regional_Statistics2017 <- cbs_get_data('70072ned')
```

As we do not need the full table, we will limit the amount of data loaded.

Firstly, we are only interested in the year 2017. We want to analyse the total population by region for that particular year.

So, we come up with the following statement (we will explain the selection of columns below):

```
library(dplyr)
library(data.table)
library(stringr)
```

```
NL_Regional_Statistics2017 <- cbs_get_data('70072ned', Perioden = "2017JJ00") %>%
  select(RegioS, Perioden, TotaleBevolking_1, Code_291, Naam_292, Code_293, Naam_294)
```

The first two columns - `RegioS` and `Perioden` - are categorical columns, i.e. they contain codes. The labels for these columns can be added with the function `cbs_add_label_columns()`:

```
NL_Regional_Statistics2017 <- cbs_add_label_columns(NL_Regional_Statistics2017)
```

We will now rename the columns of our table for further use in the exercise:

```
NL_Regional_Statistics2017 <- rename(NL_Regional_Statistics2017,
                                     Code = RegioS,
                                     Name = RegioS_label,
                                     Year = Perioden,
                                     Year_label = Perioden_label,
                                     Total_population = TotaleBevolking_1,
                                     Region_code = Code_291,
                                     Region_name = Naam_292,
                                     Province_code = Code_293,
                                     Province_name = Naam_294)
```

Each column has a label. To avoid issues later on in this exercise, we will remove these labels now:

```
cols <- colnames(NL_Regional_Statistics2017)
for (c in cols) {
  attr(NL_Regional_Statistics2017[[c]], "label") <- NULL
}
```

Let's have a first look at the table:

```
head(NL_Regional_Statistics2017)
```

```
## # A tibble: 6 x 9
##   Code  Name  Year  Year_label Total_population Region_code Region_name
##   <fct> <fct> <fct> <fct>                 <int> <fct>       <fct>
## 1 "NL0~ Nede~ 2017~ 2017               17081507 "         ~ "         ~
## 2 "LD0~ Noor~ 2017~ 2017                1722247 "         ~ "         ~
## 3 "LD0~ Oost~ 2017~ 2017                3603406 "         ~ "         ~
## 4 "LD0~ West~ 2017~ 2017                8125777 "         ~ "         ~
## 5 "LD0~ Zuid~ 2017~ 2017                3630077 "         ~ "         ~
## 6 "PV2~ Gron~ 2017~ 2017                 583581 "LD01     ~ "Noord-Ned~
## # ... with 2 more variables: Province_code <fct>, Province_name <fct>
```

In the first row we can see the total number of inhabitants in the Netherlands on January 1st, 2017, which is: **17,081,507**.

And wait, we notice something else: some columns seem to have a fixed width. For example, the columns `Code` and `Region_name`:

```
paste(NL_Regional_Statistics2017$Code[1])
```

```
## [1] "NL01  "
```

```
paste(NL_Regional_Statistics2017$Region_name[6])
```

```
## [1] "Noord-Nederland                                "
```

```
my_string <- paste(NL_Regional_Statistics2017$Region_name[6])
nchar(my_string)
```

```
## [1] 50
```

This is weird, isn't it? The string 'Noord-Nederland' - which is only 15 charaters in length - is filled out with trailing spaces to have a length of 50... This can't be good, can it? We want our strings to behave like strings with their proper length, not carrying around a trail of whitespaces.

Let's fix this issue with the function `trimws()` for the columns concerned:

```
cols <- c('Code', 'Region_code', 'Region_name', 'Province_code', 'Province_name')
NL_Regional_Statistics2017[cols] <-
  lapply(NL_Regional_Statistics2017[cols], function(x){as.factor(trimws(x))})
```

```
head(NL_Regional_Statistics2017)
```

```
## # A tibble: 6 x 9
##   Code  Name  Year  Year_label Total_population Region_code Region_name
##   <fct> <fct> <fct> <fct>                 <int> <fct>       <fct>
## 1 NL01  Nede~ 2017~ 2017               17081507 ""          ""
## 2 LD01  Noor~ 2017~ 2017                1722247 ""          ""
## 3 LD02  Oost~ 2017~ 2017                3603406 ""          ""
## 4 LD03  West~ 2017~ 2017                8125777 ""          ""
## 5 LD04  Zuid~ 2017~ 2017                3630077 ""          ""
## 6 PV20  Gron~ 2017~ 2017                 583581 LD01        Noord-Nede~
## # ... with 2 more variables: Province_code <fct>, Province_name <fct>
```

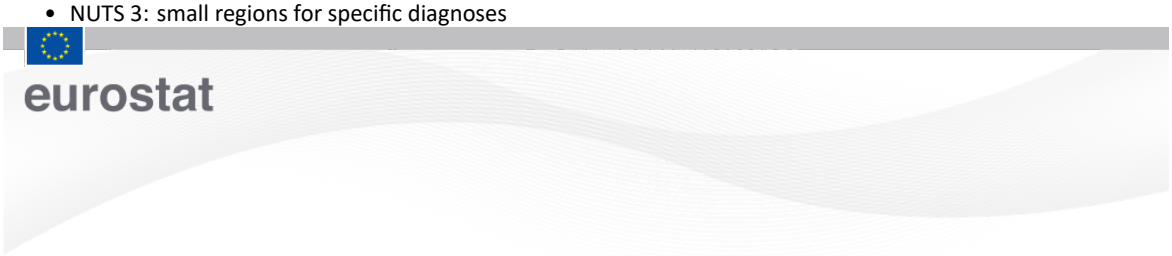
This looks much better!

NUTS - Nomenclature des Unités Territoriales Statistiques

Eurostat uses a set of geospatial classifications, but the heart of these is NUTS: the **Nomenclature of Territorial Units for Statistics**.
This classification was set up by Eurostat at the beginning of the 1970s. It serves as a single, coherent system for dividing up the EU's territory in order to produce regional statistics for the European Union.

Ther are three levels of NUTS regions:

- NUTS 1: major socio-economic regions
- NUTS 2: basic regions for the application of regional policies
- NUTS 3: small regions for specific diagnoses



More information on the NUTS classification can be found here: http://ec.europa.eu/eurostat/web/nuts/background

Eurostat even created a nice video on the topic: https://youtu.be/a4Y-hCQ-Klo

6.2.2 Extracting NUTS 1 and NUTS 2 regions

Now that we have prepared our data, we can extract the data about NUTS 1 and NUTs 2 respectively. At NUTS 1 level the Netherlands is divided into 4 regions ('Landsdelen') and at NUTS 2 level into 12 provinces:

```
NL_Regions2017_data <- filter(NL_Regional_Statistics2017, Code %like% "LD")
NL_Provinces2017_data <- filter(NL_Regional_Statistics2017, Code %like% "PV")
```

Please note: these new data frames inherit factor levels from the original data frame from which they are created. Use the **RStudio Environment pane**, to confirm that - for example - the `Name` column of the `NL_Provinces2017_data` table is now a Factor with 770 levels, whereas there are only 12 provinces. So you have to drop unused levels before you can use the new subsets for further analysis:

```
# Make sure to drop unused levels from the factors in those new data.frames
NL_Regions2017_data <- droplevels(NL_Regions2017_data)
NL_Provinces2017_data <- droplevels(NL_Provinces2017_data)
```

```
NL_Provinces2017_data %>%
  select(Code, Name, Total_population, Region_code, Region_name)
```

```
## # A tibble: 12 x 5
##    Code  Name                Total_population Region_code Region_name
##    <fct> <fct>                          <int> <fct>       <fct>
##  1 PV20  Groningen (PV)                583581 LD01        Noord-Nederland
##  2 PV21  Friesland (PV)                646874 LD01        Noord-Nederland
##  3 PV22  Drenthe (PV)                  491792 LD01        Noord-Nederland
##  4 PV23  Overijssel (PV)              1147687 LD02        Oost-Nederland
##  5 PV24  Flevoland (PV)                407818 LD02        Oost-Nederland
##  6 PV25  Gelderland (PV)              2047901 LD02        Oost-Nederland
##  7 PV26  Utrecht (PV)                 1284504 LD03        West-Nederland
##  8 PV27  Noord-Holland (PV)           2809483 LD03        West-Nederland
##  9 PV28  Zuid-Holland (PV)            3650222 LD03        West-Nederland
## 10 PV29  Zeeland (PV)                  381568 LD03        West-Nederland
## 11 PV30  Noord-Brabant (PV)           2512531 LD04        Zuid-Nederland
## 12 PV31  Limburg (PV)                 1117546 LD04        Zuid-Nederland
```

We have noticed that the province names in the column `Name` all have the suffix *(PV)*. Let's get rid of this suffix, while making sure that the column remains a factor:

```
NL_Provinces2017_data$Name <-
  as.factor(str_replace(NL_Provinces2017_data$Name," \\(PV\\)", ""))
```

Likewise, the region names in the column `Name` all have the suffix *(LD)*. We will also remove that:

```
NL_Regions2017_data %>% select(Code, Name, Total_population)
```

```
## # A tibble: 4 x 3
##   Code  Name                  Total_population
##   <fct> <fct>                            <int>
## 1 LD01  Noord-Nederland (LD)           1722247
## 2 LD02  Oost-Nederland (LD)            3603406
## 3 LD03  West-Nederland (LD)            8125777
## 4 LD04  Zuid-Nederland (LD)            3630077
```

```
NL_Regions2017_data$Name <-
  as.factor(str_replace(NL_Regions2017_data$Name," \\(LD\\)", ""))
```

6.2.3 barplot()

Let's investigate the spread of the Dutch population over the different regions by creating a barplot.

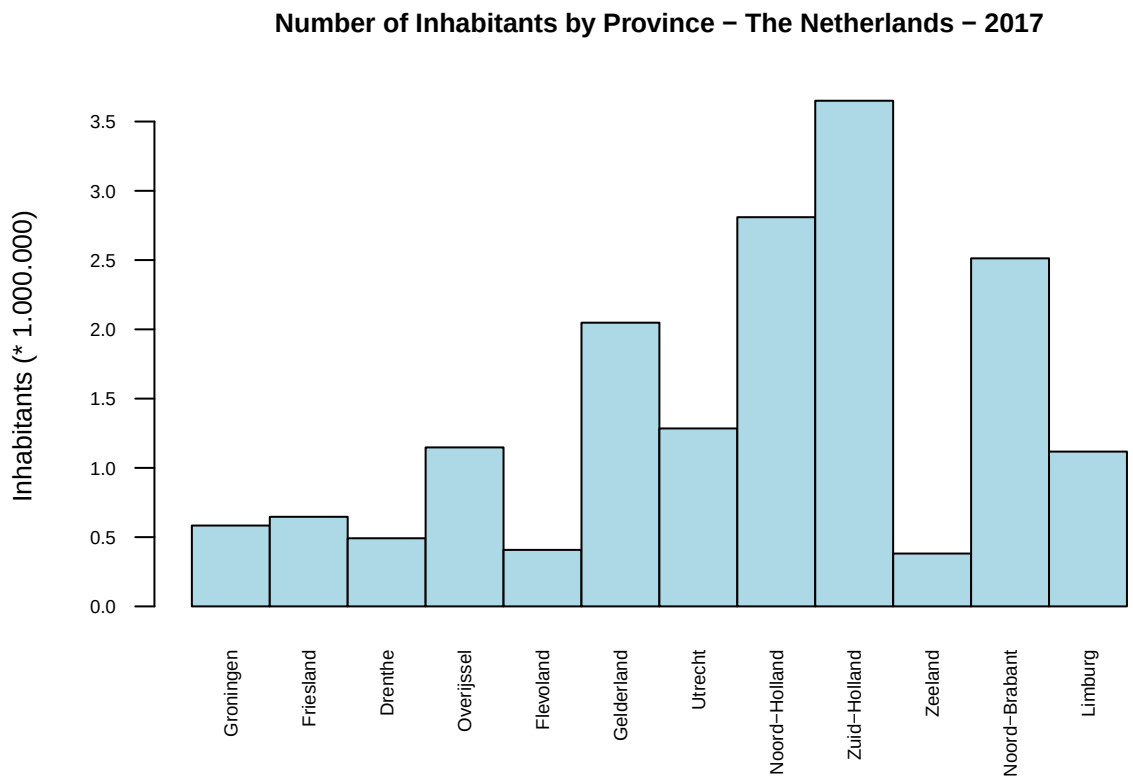
The most basic call to the `barplot()` function, is just to provide a column from a dataset and leave all the defaults:

```
barplot(NL_Provinces2017_data$Total_population)
# Try this yourself - result not printed in this manual
```

This returns a very basic plot, with no labels and no title, so not very useful.

Use `?barplot()` to see what arguments we can use. After some trial and error we might come to a statement like this:

```
barplot(NL_Provinces2017_data$Total_population / 1000000,
        names = NL_Provinces2017_data$Name,
        las = 2, cex.axis = .6, cex.names = .6,
        cex.main = .8, cex.lab = .8, space = 0,
        col = "lightblue",
        ylab = "Inhabitants (* 1.000.000)",
        main = "Number of Inhabitants by Province - The Netherlands - 2017")
```



Not bad at all, for a first attempt. Please note: the provinces are not presented in alphabetical order, but in their *natural* order, from Groningen (PV20) in the North to Limburg (PV31) in the South.

The provinces at the NUTS 2 level are grouped into regions ('Landsdelen') at the NUTS 1 level. We do know for each province to which region it belongs. Wouldn't it be nice to reflect this division in the plot?

So, In the next plot we will color the bars by region - North, East, West, South - using the `col` argument.

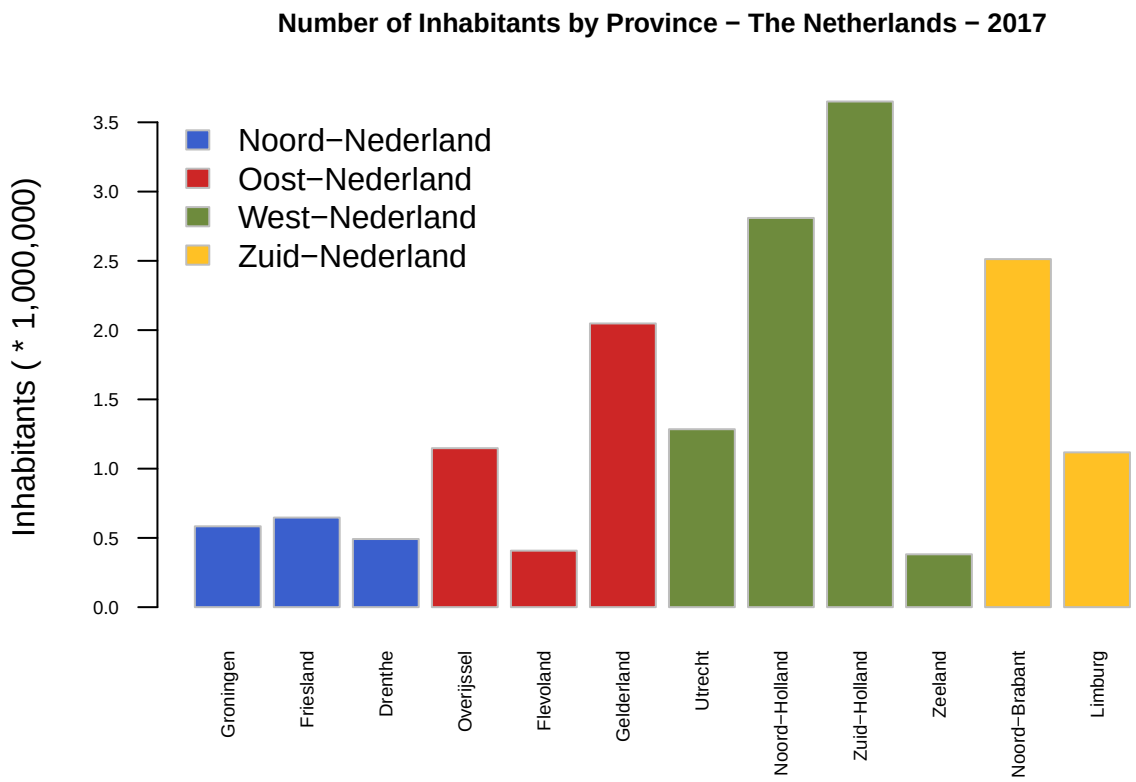
We will need 4 different colors for this plot. In the statement below we use the `palette()` function to manipulate the color

palette which is used when a `col=` has a numeric index. Using the `c()` function you create a vector with 4 distinct colors, which is passed as an argument into `palette()`:

```
palette(c("royalblue3", "firebrick3", "darkolivegreen4", "goldenrod1"))
```

And now we can create the barplot like this - with the arguments for the legend provided as a `list`:

```
barplot(NL_Provinces2017_data$Total_population / 1000000,
        names = NL_Provinces2017_data$Name,
        las = 2, cex.axis = .6, cex.names = .6,
        cex.main = .8, border = "grey",
        col = NL_Provinces2017_data$Region_code,
        ylab = "Inhabitants ( * 1,000,000)",
        main = "Number of Inhabitants by Province - The Netherlands - 2017",
        legend.text = unique(NL_Provinces2017_data$Region_name),
        args.legend = list(x = 'topleft',
                           bty = 'n',
                           fill = unique(NL_Provinces2017_data$Region_name),
                           border = 'grey'))
```



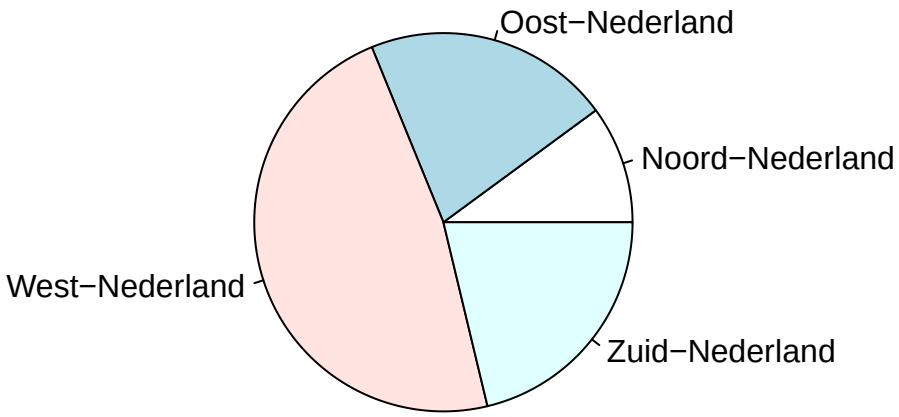
6.2.4 pie()

Based on the plots in the previous sections we might think that almost half the Dutch population lives in the westernmost region.

To confirm this hypothesis we will create a pie chart with the population by region.

This basic statement will create a simple pie chart:

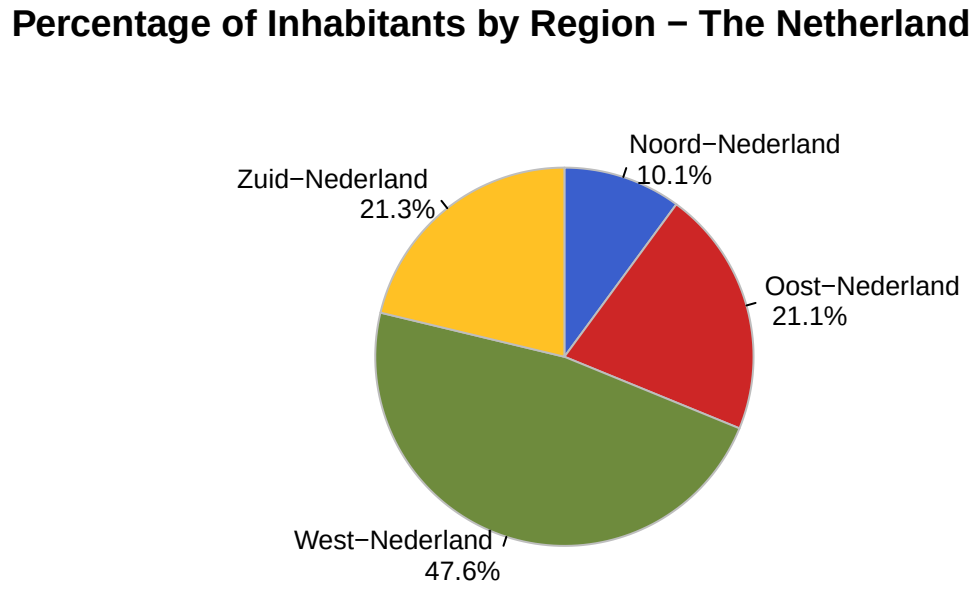
```
pie(NL_Regions2017_data$Total_population, labels = NL_Regions2017_data$Name)
```



And yes, almost half of the Dutch population lives in West-Nederland.

The pie chart on the previous page already confirmed what we wanted to know. But of course - with a little extra effort - we can improve the quality of the graph by adding a title, calculating the actual percentages and using the same colors for the regions as we did in the barplot:

```
pct <- paste0(round(NL_Regions2017_data$Total_population /
                    sum(NL_Regions2017_data$Total_population) * 100, 1), "%")
lbls <- paste(NL_Regions2017_data$Name, "\n", pct)
palette(c("royalblue3", "firebrick3", "darkolivegreen4", "goldenrod1"))
pie(NL_Regions2017_data$Total_population, labels = lbls, clockwise = TRUE,
    cex = .8, col = NL_Regions2017_data$Name, border = "grey",
    main = "Percentage of Inhabitants by Region - The Netherlands - 2017")
```



6.2.5 Extracting municipal data

Now we will extract the municipalities from `NL_Regional_Statistics2017`.

```
NL_Municipalities2017_data <- filter(NL_Regional_Statistics2017, Code %like% "GM")
nrow(NL_Municipalities2017_data)
```

```
## [1] 713
```

Are there 713 municipalities in the Netherlands? No, not anymore! There used to be much more: in 1850 there were more or less 1200 municipalities. But today there are much less: the current (2018) amount is 380. The Netherlands has a long standing tradition of grouping smaller municipalities into larger ones. This is a slow process, but each year on the 1^st of January the number of municipalities diminishes again.

We are looking at 2017, and in that year there were still 388 municipalities.

The table we have accessed contains data from 1995 up to today. And apparently in that year there were still 713 municipalities.

We can filter out the municipalities that do no longer exist by removing the ones without data. Here we use the column `Total_population` to check for that:

```
NL_Municipalities2017_data <- filter(NL_Municipalities2017_data, Total_population != "")
nrow(NL_Municipalities2017_data)
```

```
## [1] 388
```

And of course we should also drop unused factor levels:

```
NL_Municipalities2017_data <- droplevels(NL_Municipalities2017_data)
```

6.2.6 Merging sf and data.frame objects

Now we want to plot a map with the regional subdivision. To be able to do so we have to merge our data frame with a dataset containing geometry.

We will access a WFS server to download the municipal geometry - for the year 2017! - as we have done in paragraph 4.4.1:

```
# NL: Example with Dutch data
library(sf)
library(tmap)
library(httr)
library(data.table)
library(dplyr)

url <- list(hostname = "geodata.nationaalgeoregister.nl/cbsgebiedsindelingen/wfs",
            scheme = "https",
            query = list(service = "WFS",
                         version = "2.0.0",
                         request = "GetFeature",
                         typename =
                           "cbsgebiedsindelingen:cbs_gemeente_2017_gegeneraliseerd",
                         outputFormat = "application/json")) %>%
       setattr("class","url")
request <- build_url(url)

NL_Municipalities2017 <- st_read(request)
```

```
## Reading layer `OGRGeoJSON' from data source `https://geodata.nationaalgeoregister.nl/cbsgebiedsindeli
## Simple feature collection with 388 features and 5 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 13565.4 ymin: 306846.9 xmax: 277992.8 ymax: 619291
## epsg (SRID):    28992
## proj4string:    +proj=sterea +lat_0=52.15616055555555 +lon_0=5.38763888888889 +k=0.9999079 +x_0=1550
0.406857,0.350733,-1.87035,4.0812 +units=m +no_defs
```

As we do not need all attribute columns, we will select only the relevant ones:

```
NL_Municipalities2017 <- select(NL_Municipalities2017, statcode, statnaam)
```

Please note: in the statement above we only select 2 columns, but the resulting object does also contain the `geometry`column. This is due to the *sticky* character of this `geometry` column. When you select a subset of columns, the geometry will always be included, unless you explicitly drop it. In other words: a subset of an `sf` object will in itself also be an `sf` object.

Now we will rename the columns:

```
NL_Municipalities2017 <- rename(NL_Municipalities2017, Code = statcode, Name = statnaam)
```

Both datasets do contain a column with municipal codes, so now you can merge them, like you would do with any two data frames:

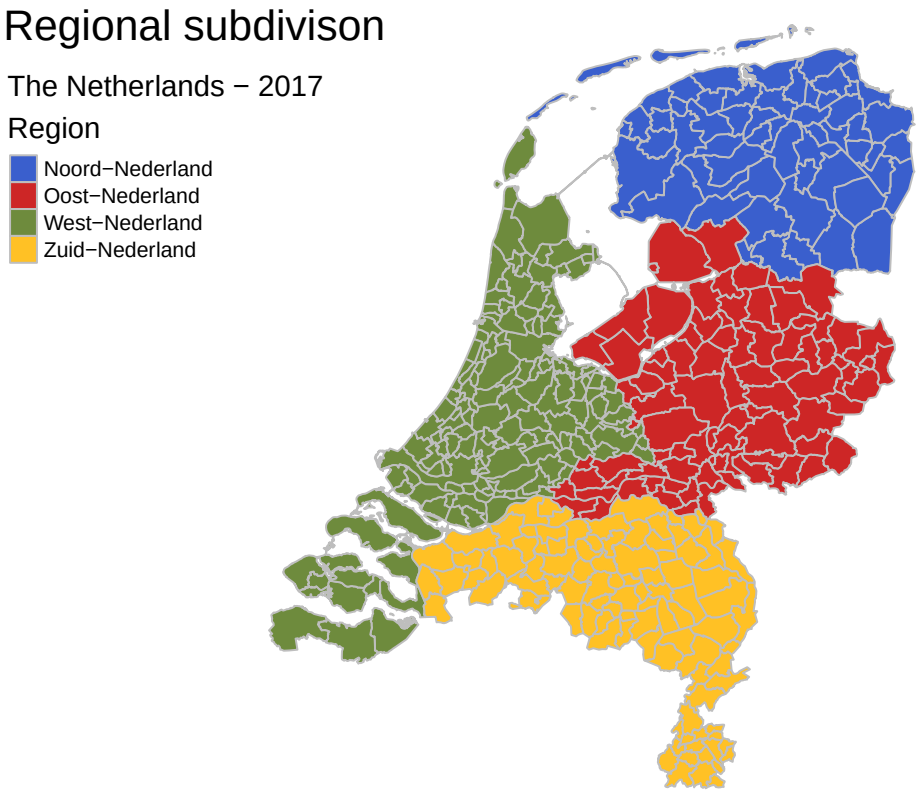
```
NL_Municipalities2017 <-
  merge(NL_Municipalities2017, NL_Municipalities2017_data, by = "Code")

class(NL_Municipalities2017)
```

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

6.2.7 qtm()

And now we can plot our map with NUTS 1 regions, again using the same colors as before:

```
qtm(shp = NL_Municipalities2017,
    title = "Regional subdivison",
    fill = "Region_name",
    fill.title = "The Netherlands - 2017\nRegion",
    fill.palette =
      palette(c("royalblue3", "firebrick3", "darkolivegreen4", "goldenrod1")),
    borders = "grey",
    format = "NLD_wide")
```



Factors in R

Conceptually, factors are variables in R which can only contain a pre-defined set of values, known as levels. Factors are often referred to as *categorical variables*, which can either be **ordered** (e.g. 'low', 'medium', 'high') or **unordered** (e.g. 'male', 'female').

One of the most important uses of factors is in statistical analysis and plotting. Storing categorical variables as factors insures that the statistical modeling functions will treat such data correctly. Factors are stored in R as a vector of integers, with a corresponding set of labels. These labels - one for each level - are used when the factor is displayed.

When data are being loaded into R, for example with the function **read.csv()**, all columns containing text (character data) will be automatically converted to factors. While this is very useful in many cases, there might also be situations where you will want to treat your data as continuous variables.

6.3 Factor: group municipalities into categories

In a previous exercise we have learned that there is an ongoing process of municipal regrouping in the Netherlands. Let's investigate opportunities for future regroupings by maping municipalities by size.

We will start this exercise by dividing our municipalities in 3 groups: small, medium and large.

A small municipality has less than 50,000 inhabitants, a medium sized one has between 50,000 and 200,000, whereas a municipality with more than 200,000 inhabitants is considered to be large.

```
NL_Municipalities2017 <- mutate(NL_Municipalities2017,
  Category = case_when(Total_population < 50000 ~ "Small population",
                       Total_population >= 50000 &
                       Total_population < 200000 ~ "Medium population",
                       Total_population >= 200000 ~ "Large population"))
```

```
class(NL_Municipalities2017$Category)
```

```
## [1] "character"
```

Now we will create an ordered factor with 3 levels:

```
population_levels <- c("Small population", "Medium population", "Large population")
NL_Municipalities2017 <- mutate(NL_Municipalities2017,
  Category = factor(Category, levels = population_levels, ordered = TRUE))

class(NL_Municipalities2017$Category)
```

```
## [1] "ordered" "factor"
```

And now we can plot our map.

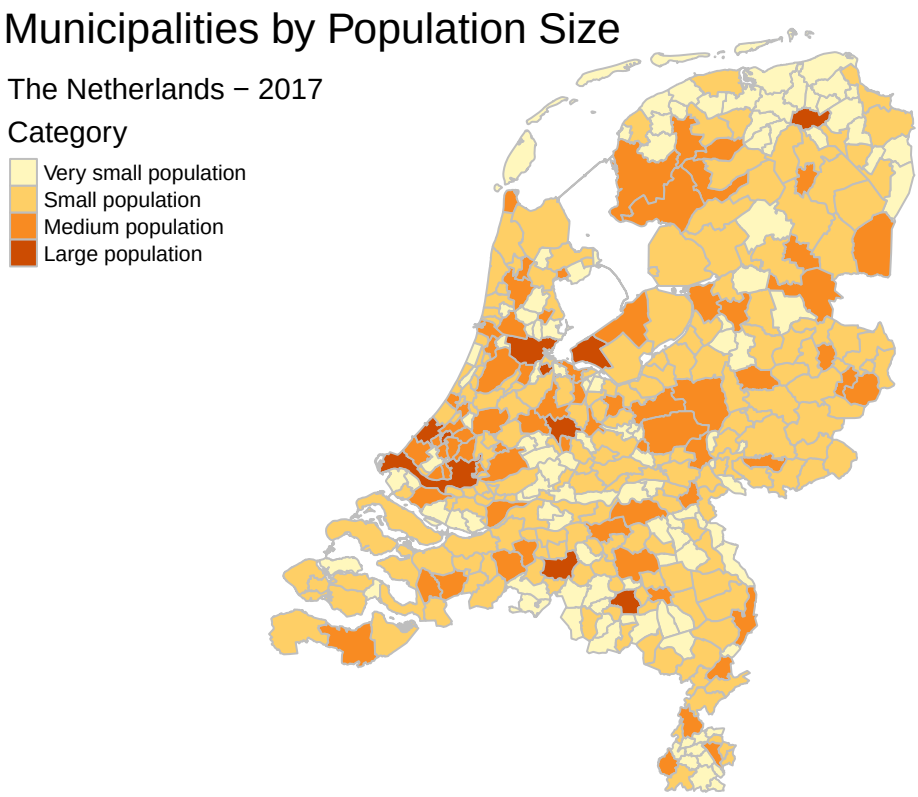
Please note: in the map printed below there are not 3 categories, but 4. We have added a category 'very small' for municipalities with less than 20,000 inhabitants.

Question:

What steps do you need to take to add a new factor level after factors have been defined?

As you can see there is still some potential to cluster very small municipalities together. But of course, this will be more a political than a scientific issue.

```
qtm(shp = NL_Municipalities2017, title = "Municipalities by Population Size",
    fill = "Category", fill.title = "The Netherlands - 2017\nCategory",
    borders = "grey", format = "NLD_wide")
```



7 Spatial Reference Systems

In this chapter we will have a quick look at an important topic when handling geospatial data, i.e the Coordinate Reference System (or Spatial Reference System), which defines how spatial elements relate to the surface of the Earth and which is used to project our geodata on a flat screen. We have ignored this subject more or less in the previous exercises, mainly because of the fact that the whole matter of projecting the data is handled very well in the 'sf' package, or rather the 'proj.4' software behind it. We just didn't have to think about it. Or wait, we did set a 'crs' explicitly - using an 'epsg' code - in the very first exercise this morning. Remember?

In general when you read in spatial data it comes with a proper CRS and if all your data is in this same projection, you really don't have to worry about all the geodesy behind it. But what if you want to combine data from different sources which appear to have different spatial reference systems?

Therefor we will shortly touch upon:

- Geographic coordinate systems vs. Projected coordinate systems
- the two main ways to describe a CRS in R: the 'epsg' code and the 'proj4string' definition
- how to reproject your data - temporarily or permanently

7.1 Coordinate systems: Geographic vs. Projected

Geographic coordinate systems identify locations on the Earth's surface using longitude and latitude. Longitude is the distance East or West in angular distance, i.e. measured in degrees, from the Prime Meridian plane. Latitude is this distance North or South of the equatorial plane. And no, there is not just one 'Lat\Lon coordinate system'. All geographic systems have to use the same reference, i.e. the Prime Meridian and the Equator, but the exact location on the Earth depends on several parameters like the ellipsoid and the datum used. So even if someone gives you data 'in degrees' you still have to find out which CRS to use. With a wrong CRS your data might end up centimeters or even several meters from the correct location. An important advantage of these geographic systems is their global reach - they can be used everywhere and are useful when dealing with international datasets. An important disadvantage is: distances cannot be easily measured in for example meters or kilometers in geographic CRSs.

Projected coordinate system have a much smaller reach, as these are based on Cartesian coordinates on an implicitly flat surface. They are only valid for a specific area - a country or even just a part of a country - where they have an origin, x and y axes, and a linear unit of measurement such as meters. We have seen several examples of data in such local, projected systems, e.g. EPSG:31287 for Austria, EPSG:3067 for Finland and EPSG:28992 for the Netherlands (but only onshore). These systems are ideal for dealing with data at a national or subnational level. In such a case you shuld really stick with the local CRS.

7.2 `epsg` codes and. `proj4string` definitions

As there are so many coordinate systems worldwide it is good to be clear about what system you use in your analysis. There are two main ways to describe a CRS in R: the `epsg` code and the `proj4string` definition. If you get your spatial data from a proper source, the `st_read()` function will return both, as we have seen in the exercises before.

The `epsg` code gets it name from the - no longer existing - *European Petroleum Survey Group*. The set of EPSG definitions is currently maintained by the International Association of Oil & Gas Producers (IOGP). Apparently location is very important in the fossil fuel industry.

An `epsg` code refers to only one, well-defined coordinate reference system. This coding system is widely adopted by the geospatial community, so if you get XY data with an existing `epsg` code you can easily plot your data on the map, in R or in any other GIS system for that matter.

A `proj4string` definition contains different parameters such as the projection type, the datum and the ellipsoid. It allows you to specify your own projection, or to modfy an existing one. But in general it would be advisable to refer to the data provider if there is something wrong with the projection of your data.

7.3 Reprojecting data

7.4 Further reading

A good introduction into Spatial Refernce Systems in R (and many other topics R spatial):

- **Lovelace, Robin, Jakub Nowosad & Jannes Muenchow** (Forthcoming). *Geocomputation with R*. CRC Press. Online version: https://geocompr.robinlovelace.net/

8 Manipulating Spatial Data

8.1 Reading the datasets

The municipality of The Hague publishes some nice datasets on http://denhaag.dataplatform.nl. We'll use some of them to learn more about manipulating spatial data:

- Neighborhoods (polygons)
- Trees (points)
- Bat flight paths (lines)

Den Haag

```
library(sf)
library(dplyr)

# Neighborhoods in The Hague
url <- "https://ckan.dataplatform.nl/dataset/c1059cef-be66-4a7a-9657-
↪  2f38f55794ed/resource/a175afe5-67e2-4e45-8b71-62f30377bf7d/download/wijken.json"
neighborhoods <- st_read(url)
```

```
## Reading layer `wijken' from data source `https://ckan.dataplatform.nl/dataset/c1059cef-
be66-4a7a-9657-2f38f55794ed/resource/a175afe5-67e2-4e45-8b71-62f30377bf7d/download/wijken.json' using di
## Simple feature collection with 44 features and 10 fields
## geometry type:  POLYGON
## dimension:      XY
## bbox:           xmin: 4.196253 ymin: 52.01485 xmax: 4.42249 ymax: 52.12791
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
```

Please note: the Spatial Reference System of this dataset is WGS84, i.e. a geographic coordinate system. In chapter 7 we have learned that if you want to make measurements, you should use a projected coordinate system.

Therefor we reproject the dataset to RD_New (RD_New or EPSG:28992 is the projected coordinate system for The Netherlands):

```
neighborhoods <- st_transform(neighborhoods, 28992)
```

```
# Trees in The Hague
url <- "https://ckan.dataplatform.nl/dataset/d604d9bb-8c2f-4e7d-a69c-
↪  ee6102890baf/resource/85327fde-9e76-40f3-a8d4-25970896fd8f/download/bomen-json.zip"
zip_file <- tempfile(fileext = ".zip")
download.file(url, destfile = zip_file, mode = "wb")
dir.create("./Data", showWarnings = FALSE)
unzip(zip_file, exdir = "./Data")
unlink(zip_file)
rm(url, zip_file)
trees <- st_read("Data/bomen-json.json") %>%
         select(id = ID, species = BOOMSOORT_WETENSCHAPPELIJ, age = LEEFTIJD)
```

```
## Reading layer `bomen' from data source `C:\GitHub\uRos2018\Spatial_Analysis_in_R_with_Open_Geodata\Da
json.json' using driver `GeoJSON'
## Simple feature collection with 137376 features and 23 fields
## geometry type:  POINT
## dimension:      XY
## bbox:           xmin: 73918.33 ymin: 447938.8 xmax: 88800.66 ymax: 459300.2
## epsg (SRID):    28992
## proj4string:    +proj=sterea +lat_0=52.15616055555555 +lon_0=5.38763888888889 +k=0.9999079 +x_0=1550
0.406857,0.350733,-1.87035,4.0812 +units=m +no_defs
```

```
# Bat flight routes in The Hague
url <- "https://ckan.dataplatform.nl/dataset/c7e9cb41-3b2d-47a4-9f1e-
↪  60ee7708f561/resource/ed7d5778-c890-4f4e-bfc3-
↪  048923761ace/download/vleermuisroutes.json"
bat_flight_paths <- st_read(url) %>% select(func = FUNCTIE, id = COUNTER)
```

```
## Reading layer `vleermuisroutes' from data source `https://ckan.dataplatform.nl/dataset/c7e9cb41-
3b2d-47a4-9f1e-60ee7708f561/resource/ed7d5778-c890-4f4e-bfc3-048923761ace/download/vleermuisroutes.json
## Simple feature collection with 92 features and 2 fields
## geometry type:  LINESTRING
## dimension:      XY
## bbox:           xmin: 4.194388 ymin: 52.0152 xmax: 4.401226 ymax: 52.12097
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
```

```
levels_dutch <- c("Migratieroute water- en meervleermu",
                  "Vliegroute gewone dwergvleermuis",
                  "Vliegroute laatvlieger",
                  "Vliegroute rosse vleermuis",
                  "Vliegroute watervleermuis")
levels_eng <- c("Migration route Myotis daubentonii and Myotis dasycneme",
                "Flight path Pipistrelllus pipistrellus",
                "Flight path Eptesicus serotinus",
                "Flight path Nyctalus noctula",
                "Flight path Myotis daubentoni")
bat_flight_paths$func <-
  plyr::mapvalues(bat_flight_paths$func, from = levels_dutch, to = levels_eng)
```

We will also reproject this dataset:

```
bat_flight_paths <- st_transform(bat_flight_paths, 28992)
```

8.2 st_area()

When we have a look at the dataset `neighborhoods` in the RStudio Data Viewer (`View(neighborhoods)`) we will notice a column called OPPERVLAKTE, meaning 'AREA'. This is a numeric field:

```
class(neighborhoods$OPPERVLAKTE)
```

```
## [1] "numeric"
```

Please note: it is bad habbit to store the AREA of a polygon in a permanent attribute field. Why? When a polygon is edited, the actual area of the object will change, but this static attribute value will remain the same.

So, it is best to only retrieve the area of our polygons when we actually need them in our analysis. To do so we can use the function `st_area()`:

```
neighborhoods$area <- st_area(neighborhoods)
```

This is a 'units' field, with the units being square meters (the units are set according to the CRS)

```
class(neighborhoods$area)
```

```
## [1] "units"
```

In this case, when we compare the content of the column OPPERVLAKTE with that of our newly created column `area` the values are still the same:

```
neighborhoods[,c("OPPERVLAKTE","area")]
```

```
## Simple feature collection with 44 features and 2 fields
## geometry type:  POLYGON
## dimension:      XY
## bbox:           xmin: 73319.26 ymin: 447950.8 xmax: 88837.08 ymax: 460520.5
## epsg (SRID):    28992
## proj4string:    +proj=sterea +lat_0=52.15616055555555 +lon_0=5.38763888888889 +k=0.9999079 +x_0=1550
0.406857,0.350733,-1.87035,4.0812 +units=m +no_defs
## First 10 features:
##    OPPERVLAKTE          area                       geometry
## 1    3085059.9 3085059.8 m^2 POLYGON ((79771.83 459540, ...
## 2     972968.5  972968.5 m^2 POLYGON ((79275.05 459063.7...
## 3    1791381.8 1791381.8 m^2 POLYGON ((79533.69 457903.1...
## 4    2744298.9 2744298.9 m^2 POLYGON ((80931.01 457037.9...
## 5     871291.1  871291.1 m^2 POLYGON ((80075.16 456289.9...
## 6    1114378.7 1114378.7 m^2 POLYGON ((79210.1 457434.2,...
## 7    2469866.8 2469866.7 m^2 POLYGON ((77191 457958.8, 7...
## 8     847246.6  847246.6 m^2 POLYGON ((76660.75 456164.3...
## 9    1351610.5 1351610.5 m^2 POLYGON ((78080.86 456527.1...
## 10    875515.8  875515.8 m^2 POLYGON ((79003.62 456409, ...
```

Now, let's rename our columns to English equivalents for further analysis (dropping this silly OPPERVLAKTE and some other irrelevant fields on the fly):

```
neighborhoods <- neighborhoods %>% select(nh_code = WIJKCODE, nh_name = WIJKNAAM,
↪  district_code = STADSDEELCODE)
```

8.3 The package `units` with the function `set_units()`

Here we would like to introduce the package `units` - also created by Edzer Pebesma - which provides support for measurement units in R.

The package `units` on CRAN:
https://cran.r-project.org/package=units

As we have seen in the previous paragraph - when we retreive the area of a polygon with the function `st_area()` the units will be taken from the CRS:

```
neighborhoods$area <- st_area(neighborhoods)
```

But we think it a bit strange to talk about neighborhood size in terms of square meters. Don't you agree? We would rather know these values in square kilometers. Let's use the function `set_units()` to convert the values:

```
library(units)
```

```
## udunits system database from C:/Users/Egge-JanPolleTensing/Documents/R/win-library/3.5/units/share/u
```

```
neighborhoods$area2 <- set_units(neighborhoods$area, km^2)
```

And we may also want to round these values to 2 digits:

```
neighborhoods$area3 <- round(neighborhoods$area2, digits = 2)
```

And now it is simple to answer the following question: What is the largest neighborhood in The Hague?

```
neighborhoods %>% filter(area == max(area)) %>% st_set_geometry(NULL)
```

```
##   nh_code  nh_name district_code        area         area2     area3
## 1      42 Ypenburg             8 5056216 m^2 5.056216 km^2 5.06 km^2
```

8.4 `st_length()`

In a similar way we can retrieve the length of line objects:

```
bat_flight_paths$length <- st_length(bat_flight_paths)
```

Next question: What are the five shortest bat flight routes?

```
bat_flight_paths %>% arrange(length) %>%
                     st_set_geometry(NULL) %>%
                     head(n = 5)
```

```
##                                  func   id     length
## 1          Flight path Myotis daubentoni 1509 66.71602 m
## 2 Flight path Pipistrelllus pipistrellus 1521 67.04012 m
## 3 Flight path Pipistrelllus pipistrellus 1507 76.93388 m
## 4 Flight path Pipistrelllus pipistrellus 1522 81.66870 m
## 5          Flight path Myotis daubentoni 1575 98.18318 m
```

8.5 `st_distance()`

How many meters are tree 1351893 and 1398051 apart from each other?

```
tree_1351893 <- filter(trees, id == 1351893)
tree_1398051 <- filter(trees, id == 1398051)

st_distance(tree_1351893, tree_1398051) %>% round()
```

```
## Units: m
##      [,1]
## [1,]   36
```

How far is the nearest bat flight path from tree 1351893?

```
st_distance(tree_1351893, bat_flight_paths) %>% min() %>% round()
```

```
## 10 m
```

And how many trees are there within 20 meters from the bat flight path 1509?

```
flight_path_1509 <- filter(bat_flight_paths, id == 1509)
lengths(st_is_within_distance(flight_path_1509, trees, dist = 20))
```

```
## [1] 9
```

8.6 Aggregating and filtering data

One of the nice things of sf objects is that you can take advantage of the dplyr functions like `select()`, `filter()`, `mutate()`, `group_by()`, `summarize()` and `arrange()`. We already saw some examples in earlier chapters.
To illustrate the power of dplyr, let's calculate the count and mean age for each species in the trees dataset? List the species in alphabetical order. Only print the first 3 species.

```
trees %>% select(species, age) %>%
          group_by(species) %>%
          summarize(mean_age = round(mean(age, na.rm =  TRUE), 2), count = n()) %>%
          arrange(species) %>%
          st_set_geometry(NULL) %>%
          head(n = 3)
```

```
## # A tibble: 3 x 3
##   species        mean_age count
##   <fct>             <dbl> <int>
## 1 Abies              19.5    78
## 2 Abies concolor     31.5     4
## 3 Abies firma        47       3
```

8.7 Spatial aggregation

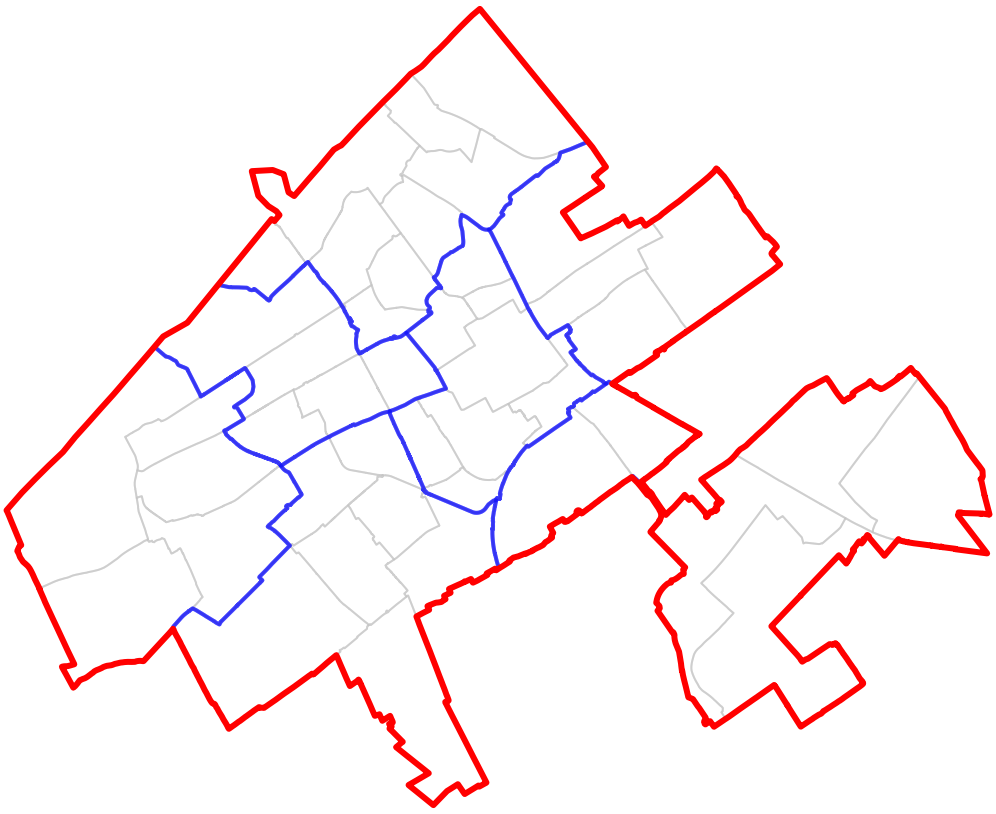
You can also aggregate spatial data, for instance calculate the city border using the neighborhoods dataset:

```
the_hague <- st_union(neighborhoods)
```

Or you can group the neighborhoods by district:

```
the_hague_districts <- neighborhoods %>% group_by(district_code) %>% summarise()
```

```
tm_shape(neighborhoods) +
  tm_borders(col = "grey", alpha = 0.5) +
tm_shape(the_hague_districts) +
  tm_borders(col = "blue", lwd = 2, alpha = 0.5) +
tm_shape(the_hague) +
  tm_borders("red", lwd = 3) +
tm_layout(frame = FALSE)
```



Or calculate the total number of trees for each neighborhood:

```
neighborhoods$total_trees <- lengths(st_covers(neighborhoods, trees))
```

It's always a good idea the verify your outcomes.

```
nrow(trees)
```

```
## [1] 137376
```

```
sum(neighborhoods$total_trees)
```

```
## [1] 133504
```

We're short 3872 trees in the neighborhood dataset! Where did those trees go?

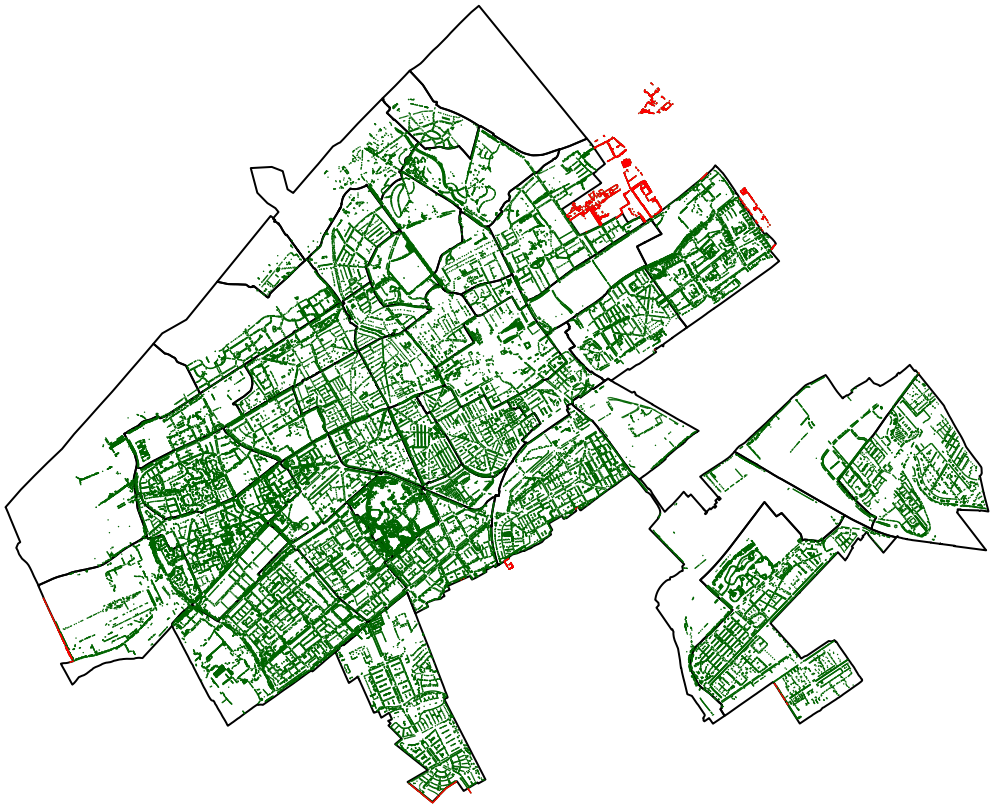
```
trees_not_in_neighborhood <- trees[the_hague, op = st_disjoint]
nrow(trees_not_in_neighborhood)
```

```
## [1] 3872
```

3872 trees are not in a neighborhood of The Hague. Let's make a plot:

```
tm_shape(neighborhoods) +
  tm_borders("black") +
tm_shape(trees) +
  tm_symbols(col = "darkgreen", scale = 0.05, border.lwd = NA) +
tm_shape(trees_not_in_neighborhood) +
  tm_symbols(col = "red", scale = 0.05, border.lwd = NA) +
```

```
tm_layout(frame = FALSE) +
tm_legend(show = FALSE)
```



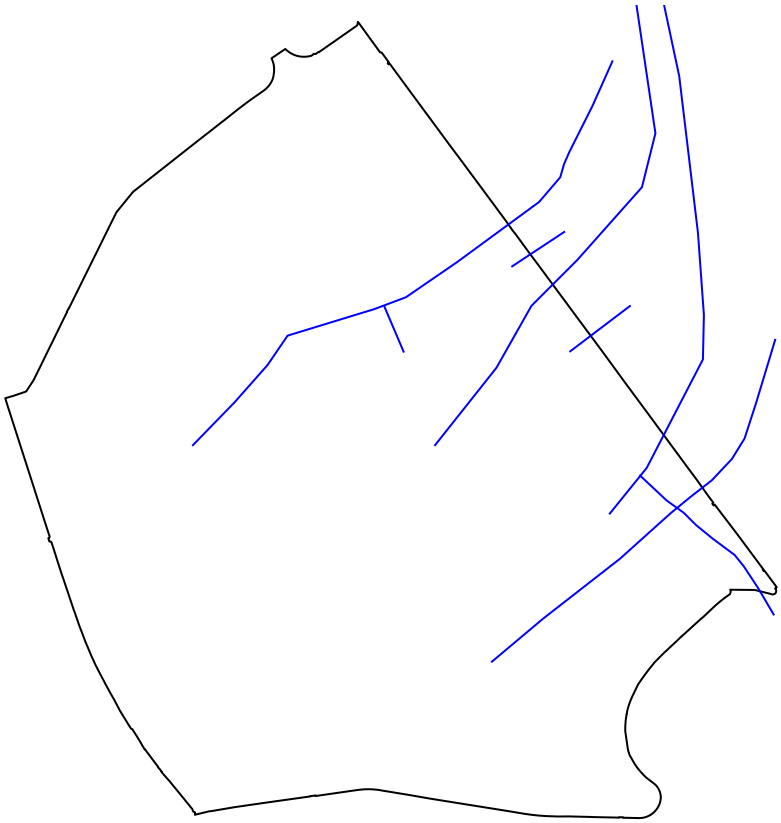
To be on the safe side, we'll remove all the trees from our dataset that are outside the neighborhoods. These are the red trees in the plot.

```
trees <- trees[the_hague, op = st_intersects]
nrow(trees)
```

```
## [1] 133504
```

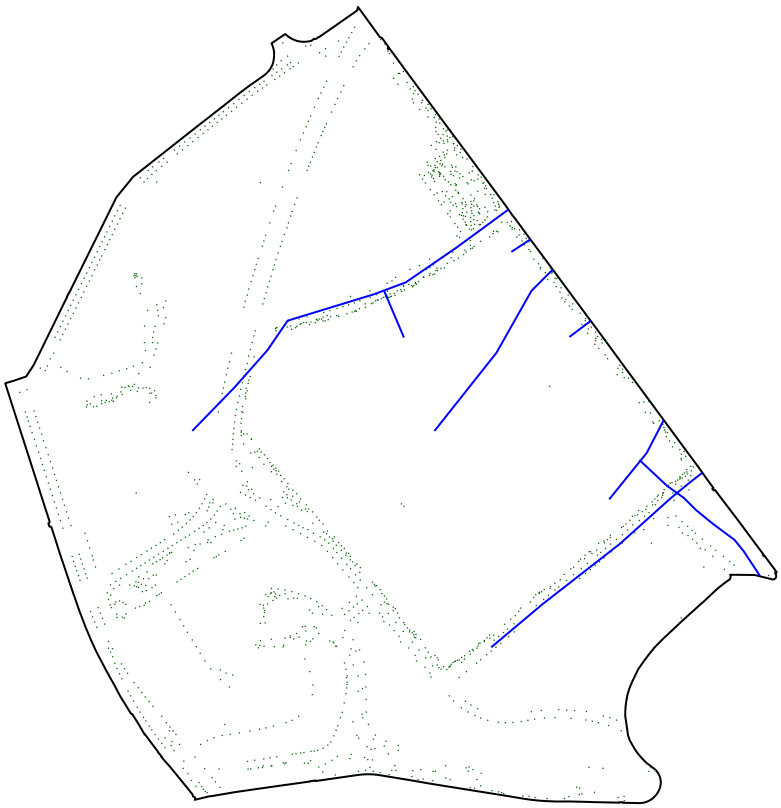
Another example of applying a spatial filter: Where are the bat flight paths in the Zorgvliet neighborhood?

```
zorgvliet <- filter(neighborhoods, nh_name == "Zorgvliet")
bat_flight_paths_zorgvliet <- bat_flight_paths[zorgvliet, op = st_intersects]

tm_shape(zorgvliet) +
  tm_borders("black") +
tm_shape(bat_flight_paths_zorgvliet) +
  tm_lines(col = "blue") +
tm_layout(frame = FALSE) +
tm_legend(show = FALSE)
```



That's kind of ugly, those flight paths running outside the neighborhood. Let's cut them on the neighborhood border. Also add the trees to the plot.

```
bat_flight_paths_zorgvliet <- st_intersection(bat_flight_paths, zorgvliet)
trees_zorgvliet <- st_intersection(trees, zorgvliet)

tm_shape(zorgvliet) +
  tm_borders("black") +
tm_shape(trees_zorgvliet) +
  tm_symbols(col = "darkgreen", scale = 0.05, border.lwd = NA) +
tm_shape(bat_flight_paths_zorgvliet) +
  tm_lines(col = "blue") +
tm_layout(frame = FALSE) +
tm_legend(show = FALSE)
```



8.8 Calculating and visualizing density

Note that the dataset only contains trees in public space. But still, the tree density may give us an idea of how 'green' a neighborhood is.

What are the neighborhoods with the least and the most trees?

```
neighborhoods$nh_name[which.min(neighborhoods$total_trees)] %>% as.character()
```

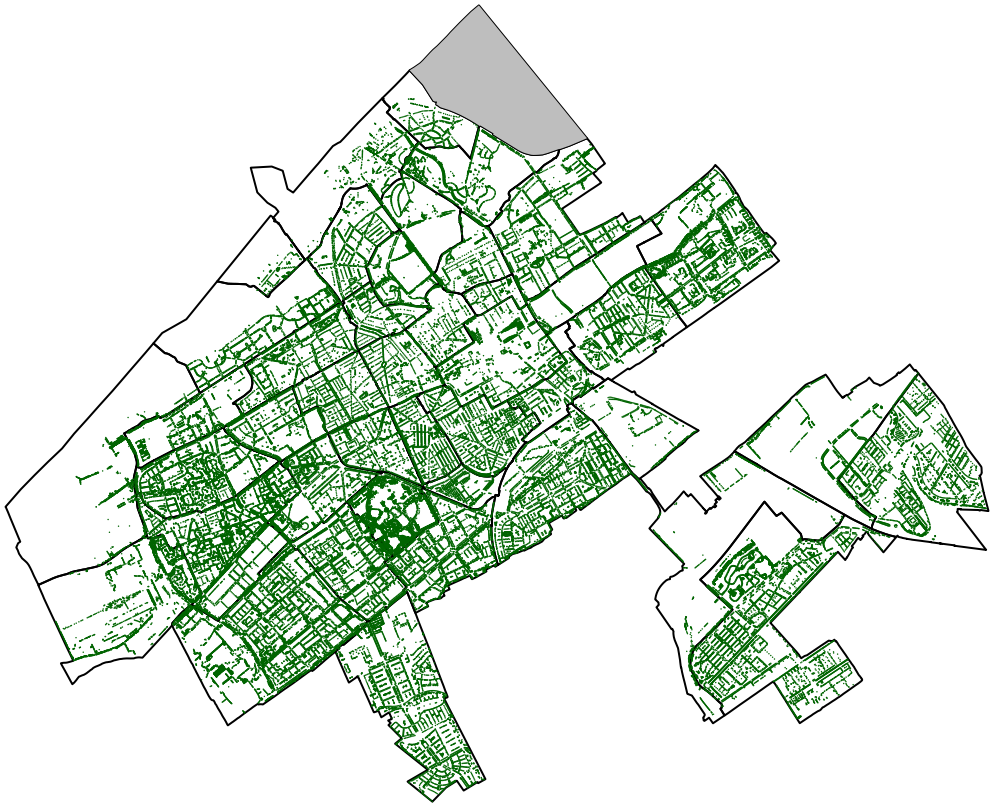
```
## [1] "Oostduinen"
```

```
neighborhoods$nh_name[which.max(neighborhoods$total_trees)] %>% as.character()
```

```
## [1] "Bouwlust"
```

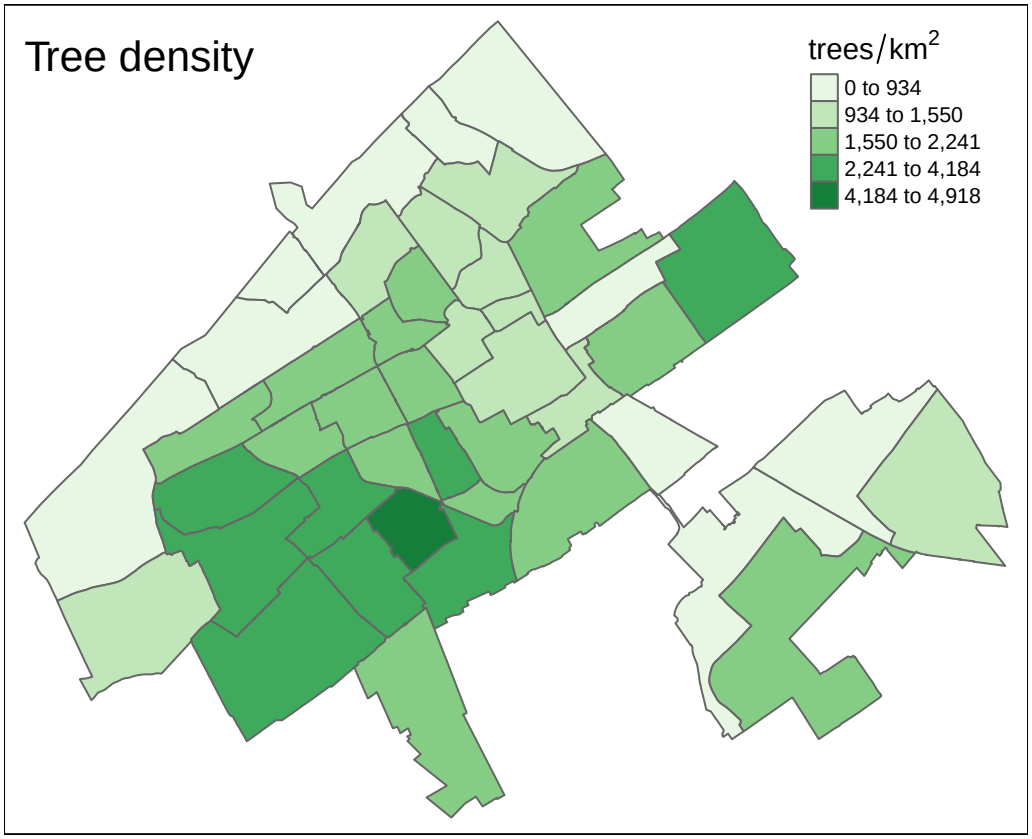
is a neighborhood with no trees at all in public space. Let's find out where it is.

```
no_trees <- filter(neighborhoods, total_trees == 0)

tm_shape(neighborhoods) +
  tm_borders("black") +
tm_shape(no_trees) +
  tm_fill("grey") +
tm_shape(trees) +
  tm_symbols(col = "darkgreen", scale = 0.05, border.lwd = NA) +
tm_layout(frame = FALSE) +
tm_legend(show = FALSE)
```



Now visualize the tree density for each neighborhood in The Hague.

```
neighborhoods$tree_density <-
  round(neighborhoods$total_trees / units::set_units(neighborhoods$area, km^2))
qtm(shp = neighborhoods, fill = "tree_density",
fill.palette = "Greens",
fill.style = "kmeans", title = "Tree density",
fill.title = parse(text = "trees/km^2"))
```



Again notice that this plot gives a distorted view of reality, because the dataset only contains trees in public space.But as an example of the power of spatial data manipulation and visualisation, it suffices.

8.9 Spatial joins

Appendix A: List of abbreviations used

Abbreviation	Meaning
API	application programming interface
CRAN	Comprehensive R Archive Network
GIS	Geographic Information System
GML	Geography Markup Language
IOGP	International Association of Oil & Gas Producers
JSON	JavaScript Object Notation
OGC	Open Geospatial Consortium
REST	Representational State Transfer
SaaS	software as a service
WFS	Web Feature Service
XML	eXtensible Markup Language

Appendix B: Additional exercises

OGC Web Feature Service (WFS)

For background information on the steps taken in the exercise below, please refer to chapter 4

`request=GetFeature` - another example from the Netherlands

In this additional exercise we will access a WFS service offering information about *bevolkingskernen* (human settlements) in the Netherlands in 2011. More information on this dataset can be found here: Dataset: CBS Bevolkingskernen.

The URL with `GetCapabilities` request is:
https://geodata.nationaalgeoregister.nl/bevolkingskernen2011/wfs?request=GetCapabilities

In the `<FeatureTypeList>` section of this XML response we can see that this service only offers one `<FeatureType>`, i.e. one dataset. This dataset has the `<Name>` **bevolkingskernen2011:cbsbevolkingskernen2011**. That's the value we are giving to the `typename` parameter in our `GetFeature` request.

The URL with `DescribeFeatureType` request is:
https://geodata.nationaalgeoregister.nl/bevolkingskernen2011/wfs?service=WFS&version=2.0.0&request=DescribeFeatureType&typename=bevolkingskernen2011:cbsbevolkingskernen2011

This dataset does contain quite some attribute columns!

This is the script to build the full `GetFeature` request:

```
# NL: Human Settlement Analysis
library(sf)
library(tmap)
library(httr)
library(data.table)

url <- list(hostname = "geodata.nationaalgeoregister.nl/bevolkingskernen2011/wfs",
            scheme = "https",
            query = list(service = "WFS",
                         version = "2.0.0",
                         request = "GetFeature",
                         typename = "bevolkingskernen2011:cbsbevolkingskernen2011",
                         outputFormat = "application/json")) %>%
       setattr("class","url")
request <- build_url(url)
```

The variable `request` will now contain this link. (When you click this link the server response will be shown in your browser in GeoJSON format.)

Now we want to feed this response directly into R, like this:

```
NL_Human_Settlements2011 <- st_read(request)
```

```
## Reading layer `OGRGeoJSON' from data source `https://geodata.nationaalgeoregister.nl/bevolkingskernen
## Simple feature collection with 2168 features and 112 fields
## geometry type:  POLYGON
## dimension:      XY
## bbox:           xmin: 15153.79 ymin: 308078.3 xmax: 277312.9 ymax: 610955.6
## epsg (SRID):    28992
## proj4string:    +proj=sterea +lat_0=52.15616055555555 +lon_0=5.38763888888889 +k=0.9999079 +x_0=155000
0.406857,0.350733,-1.87035,4.0812 +units=m +no_defs
```

```
plot(st_geometry(NL_Human_Settlements2011), col = "orange", border = "red")
```