Creating a plot based on OData tables by Statistics Netherlands (CBS)

Han Oostdijk (www.hanoostdijk.nl)

6 april 2016

R libraries used

```r
library(curl)
library(magrittr)
library(XML)
library(dplyr)
library(ggplot2)
library(rgdal)
library(rgeos)
library(maptools)
```

Introduction

This document gives an example of accessing data of Statistics Netherlands (CBS). Because I was not aware of the R package **cbsodataR** by Edwin de Jonge I created some functions to do this. After that I show how this data can be merged with CBS map information to produce a map of the Netherlands coloured according to one of these statistics.

References about the OData environment of Statistics Netherlands.

The file 2014handleidingcbsopendataservices.pdf (in Dutch) describes the OData environment at CBS. The catalog contains information about the available data.
A introduction to **OData** can be found in Introducing OData and full details in OData - the best way to REST.
When writing queries a useful reference is paragraph *11.2.5 Querying Collections* in the protocol document.

Main code

Here we show the code for a specific map that we want to create. We include (source) the program files *odata.r* and *plot_NL.r* that will be described later. I selected from the catalog a table with a regional component: **82935NED**. This has information (in Dutch only) about investment in various fixed assets for (the regions of) the Netherlands. Let's say that I want to see how total total investment in 2013 (the last year available) is distributed over the provinces in the form of a map.

Access describing information (meta data) of table

The information (in Dutch only) in the catalog gives some information about the table. This information is also available in table form and we will read this into the variables:

- table_list : a named character vector that contains the full url of the sub tables. We are interested in the sub table *TypedData* that contains the topic data according to some coded dimensions. So for each dimension there is also a sub table with the mapping coded <-> decoded value.

- props : a data.frame containing the dimensions and topics
- tabinfo : information about the table

We can use this information to see what is available and to build the code that selects precisely the information that we need.

```r
source('odata.r')
source('plot_NL.r')

myroot   = "http://opendata.cbs.nl/ODataFeed/OData"
mytable  = "/82935NED"

table_list = get_cbs_table_info(get_cbs_data(myroot,mytable))

props      = copy_table(table_list['DataProperties'],mt=prop_table_fun)
tabinfo    = copy_table(table_list['TableInfos'])
```

Select the information we need

We are interested in a small table, so we could read the entire table in R and do the selections there. Statistics Netherlands also handles very large tables and in that case it is better to do some preprocessing on its server by using a query statement. (See paragraph *11.2.5 Querying Collections* in the protocol document for query functions.) In any case the server will not transfer more than 10000 records per request. Use *$top=* and *$skip=* clauses in such cases.
For demonstration purposes we will show both selection methods here: *query* in the request and *filter* in the R code. So we request the *TypedDataSet* table and use R code to select the information we need:

- *couple_data* function ensures that all coded dimensions are replaced by their decoded counterparty and that the topic fields are made numeric. Only the region dimension is also kept in coded form
- *filter* selects only province data from 2013 (repeats the selection in the query)
- *mutate* keeps the first 4 characters of the region code (removes trailing blanks)
- *mutate* creates the variable *totFixedAssetsr* in billions of euro
- *mutate* creates the variable *label_var* with the formatted value of total fixed assets
- *select* only keeps the variables that we need for the plot

```r
query      = paste0( "?$format=atom&",
                     "$filter=substring(Perioden,0,4) eq '2013' and ",
                     "substring(RegioS,0,2) eq 'PV'")
TypedData  = copy_table(table_list['TypedDataSet'],mt=data_table_fun,q=query)

data_df    =
  couple_data (TypedData,dim_vars(props),topic_vars(props),table_list) %>%
  filter(grepl('^PV',.$RegioS_coded)&Perioden=='2013') %>%
  mutate(RegioS_coded=substring(RegioS_coded,1,4)) %>%
  mutate(totFixedAssets=TotaleInvesteringen_1/1000) %>%
  mutate(label_var=sprintf('%.1f',totFixedAssets)) %>%
  select(RegioS_coded,totFixedAssets,label_var)
```

Combine the data with map information and plot

With the *plot_NL* function we combine the selected information (on province level) with the CBS map information of the same level. We label each province with the total amount of fixed assets that we formatted in variable *label_var* and colour it based on the same amount. The result can be seen in Figure 1 on page 4.

```
p = plot_NL(dsn_map('P'),         # map to use (provincial)
    data_df,                      # data to use
    'totFixedAssets',             # name variable to plot
    'total\ninvestments\nbln euro',    # caption variable to plot
    'RegioS_coded',               # name variable to link to map
    labels=T,                     # labels to plot?
    label_var='label_var',        # name variable that contains labels
    co=T)                         # check overlap labels ?
```

```
## OGR data source with driver: GML
## Source: "D:/data/maps/cbs_provincies.gml", layer: "cbs_provincie_2015_gegeneraliseerd_voorlopig"
## with 12 features
## It has 5 fields

## Regions defined for each Polygons
```

```
print(p)
```

Session info

```
sessionInfo()
```

```
## R version 3.2.4 (2016-03-10)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 10586)
##
## locale:
## [1] LC_COLLATE=English_United States.1252  LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252 LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] maptools_0.8-39 rgeos_0.3-19    rgdal_1.1-8     sp_1.2-2        ggplot2_2.1.0
##  [6] dplyr_0.4.3     XML_3.98-1.4    magrittr_1.5    curl_0.9.6      knitr_1.12.3
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.3        munsell_0.4.3      lattice_0.20-33    colorspace_1.2-6
##  [5] R6_2.1.2           stringr_1.0.0      plyr_1.8.3         tools_3.2.4
##  [9] parallel_3.2.4     grid_3.2.4         gtable_0.2.0       DBI_0.3.1
## [13] htmltools_0.3      lazyeval_0.1.10    yaml_2.1.13        assertthat_0.1
## [17] digest_0.6.9       RColorBrewer_1.1-2 formatR_1.3        evaluate_0.8.3
## [21] rmarkdown_0.9.5    labeling_0.3       stringi_1.0-1      scales_0.4.0
## [25] foreign_0.8-66
```

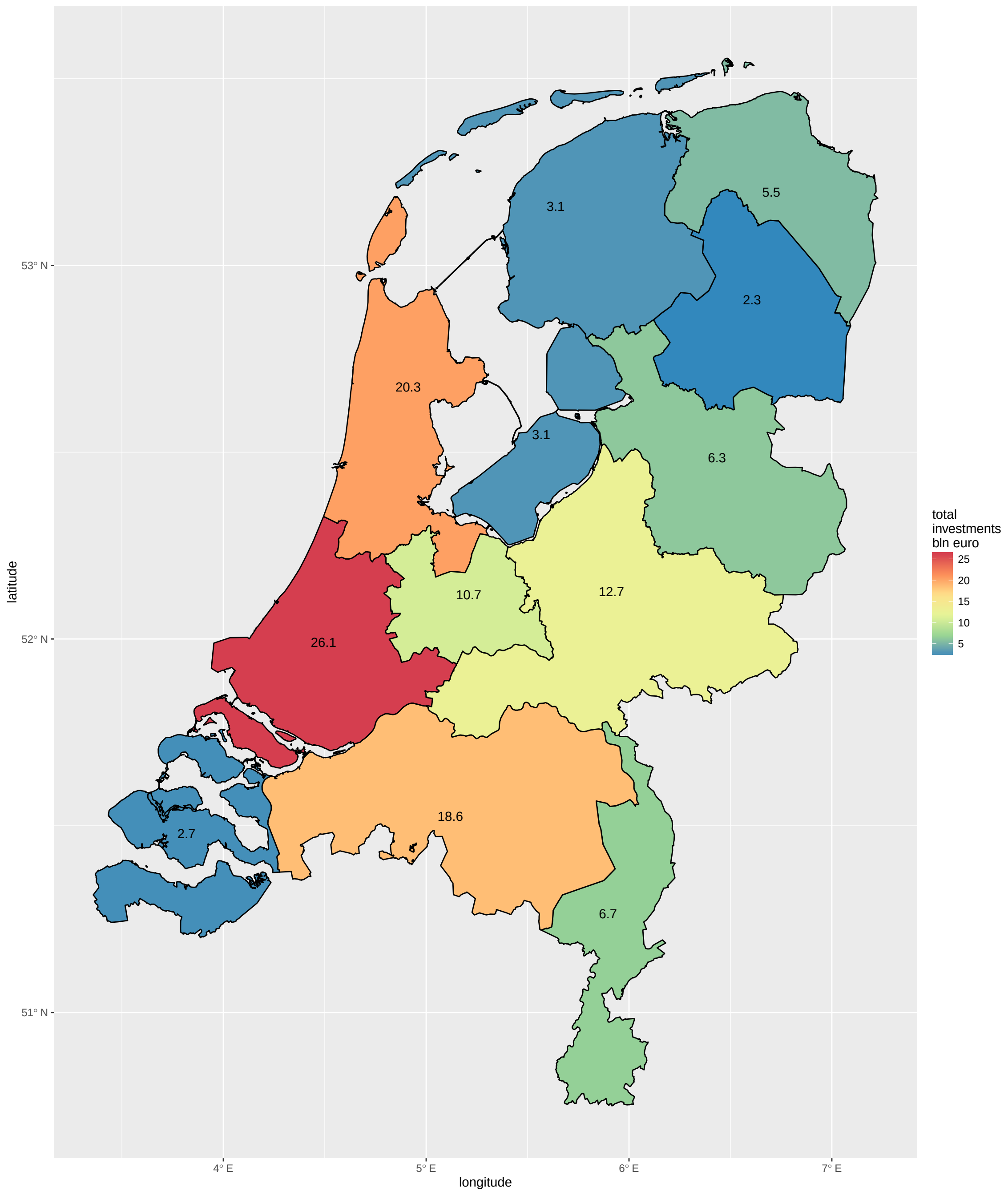

Figure 1: Total fixed assets (in bln. euro) per province for 2013 (source CBS)

Appendix

Functions related to OData

Libraries and constants

```r
library(magrittr)
library(dplyr)
library(curl)
library(XML)

std_namespaces = c(ns="http://www.w3.org/2005/Atom",
  m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata",
  d="http://schemas.microsoft.com/ado/2007/08/dataservices")
```

get_cbs_data

This function passes a request for data to the CBS OData server

```r
get_cbs_data <- function (url, query=NULL, save_file_name = NULL) {
  # query  = "?$format=atom&$filter=GeneesmiddelengroepATC eq '100000'"
  if (!is.null(url)) {
    f  = paste0(url,query)
  } else{
    f  = url
  }
  f  = URLencode(f)
  r  = curl_fetch_memory(f)
  x  = rawToChar(r$content)
  doc = xmlParse(x,asText =T)
  if (!is.null(save_file_name)) {
    saveXML(doc, save_file_name)
  }
  return(doc)
}
```

get_cbs_table_info

This function returns a named character vector with the urls of the sub tables. When it is e.g used as *table_list = get_cbs_table_info(get_cbs_data("http://opendata.cbs.nl/ODataFeed/OData/82935NED"))* one can use *table_list['DataProperties']* and *table_list['TableInfos']* as references to two of its sub tables.

```r
get_cbs_table_info <- function(doc) {
  m1    = xpathSApply(doc,"//@href/..",
    function(x) c(xmlValue(x), xmlAttrs(x)[["href"]]))
  hrefs = m1[2,]
  names(hrefs) =m1[1,]
  return(hrefs)
}
```

Get information from the XML structure

The function **copy_table** calls the **get_cbs_data** function to get the data from the CBS server and extracts the information from the XML structure. How the information is to be extracted is specified by the *mt*

parameter. It can point to the **prop_table_fun** function (for the *DataProperties* sub table) or to the **data_table_fun** function for the other (rectangular) sub tables. In the default case the full XML structure is returned.
When the data needs to be read in more than one call of **get_cbs_data** later parts are not written to an xml file and when no value was given for *mt* the later parts are skipped because there is no obvious way to concatenate xml structures.

```r
copy_table <- function (dsn,  mt = NULL, query= NULL, save_XML = NULL) {
  n1 = paste0('temp_', names(dsn))
  if (is.null(save_XML)) {
    save_file_name = NULL
  } else if (nchar(save_XML) == 0) {
    save_file_name = paste0(n1, '.xml')
  } else {
    save_file_name = save_XML
  }
  t1    = get_cbs_data(dsn, query, save_file_name = save_file_name)
  if (is.null(mt))
    return(t1)
  t1d = mt(t1)
  next1 = xpathSApply(t1,"//ns:link[@rel='next']",
    function(x) xmlAttrs(x)[["href"]],
    namespaces = std_namespaces)
  while (length(next1)> 0 ) {
    t1    = get_cbs_data(next1) # no save for part2 and later
    t1d   = rbind(t1d,t1d = mt(t1))
    next1 = xpathSApply(t1,"//ns:link[@rel='next']",
      function(x) xmlAttrs(x)[["href"]],
      namespaces = std_namespaces)
  }
  return(t1d)
}

data_table_fun <- function(doc) {
  t1n <- xpathApply(doc,
    '//ns:entry[1]//m:properties[1]/d:*',
    xmlName,
    namespaces = std_namespaces)
  t1d  = xpathSApply(doc, '//m:properties/d:*',xmlValue)
  t1d  = as.data.frame(matrix(t1d, ncol = length(t1n), byrow = T),
    stringsAsFactors =F)
  names(t1d) = t1n
  return(t1d)
}

prop_table_fun <- function(doc) {
  m     = xpathSApply(doc, '//m:properties/d:*',
    function(x)
      c(
        xpathSApply(xmlParent(x), './d:ID', xmlValue, namespaces = std_namespaces),
        xmlName(x),
        xmlValue(x)
      ))
  # m matrix: r1 number; r2 field ; r3 value
  uf    = unique(m[2, ])
  # "ID" "Position" "ParentID" "Type" "Key" "Title" "Description" "ReleasePolicy"
  # "Datatype" "Unit" "Decimals" "Default"
  nc    = length(uf)
  nr    = 1+max(as.numeric(m[1, ]))
  m2 = matrix(rep('', nr * nc), nrow = nr, ncol = nc)
  for (i in 1:nr) {
    m3 = m[, m[1, ] == paste(i-1)] # counting origin=0
    ix = match(m3[2, ], uf)
    m2[i, ix] = m3[3, ]
  }
  colnames(m2) = uf
  rownames(m2) = 1:nr
  as.data.frame(m2,stringsAsFactors =F)
}
```

couple_data

The **couple_data** function takes a coded data.frame, converts all topic variables to numeric and decodes all dimensions with the aid of **couple_data_dim**. The coded *RegionS* dimension is allways kept under the name *RegionS_coded*.

```
couple_data <- function(
  df,         # data.frame with coded dimensions (e.g. read by copy_table)
  dv,         # character vector with the names of the dimensions
  tv,         # character vector with the names of the topics
  table_list # named vector with urls of sub tables
             # (e.g. read by get_cbs_table_info)
  ) {
  tt = df %>%
    mutate_each_(funs(as.numeric),tv$Key)  #topics -> numeric
  for (dim in dv$Key)  {
    if (dim == c('RegioS') ) {
      tt = couple_data_dim(tt, table_list['RegioS'],keep_code=T) # link RegioS
    } else {
      tt = couple_data_dim(tt, table_list[dim],keep_code=F) # link other dimensions
    }
  }
  return(tt)
}

couple_data_dim <- function(tt, dsn, keep_code=F) {
  dim  = names(dsn)
  tab1 = copy_table(dsn, data_table_fun) %>%
    select(Key, Title) %>%
    rename_(.dots = setNames('Title', paste0(dim, '_decode')))
  by1  = c('Key') ; names(by1) = dim
  tt = tt %>%
    inner_join(tab1, by = by1) %>%
    rename_(.dots = setNames(dim, paste0(dim, '_coded'))) %>%
    rename_(.dots = setNames(paste0(dim, '_decode'), dim))
  if (keep_code == F) {
    tt = tt %>%
      select_(.dots = setdiff(names(.), paste0(dim, '_coded')))
  }
  return(tt)
}
```

Functions to produce the topics and dimensions

The **topic_vars** and **dim_vars** functions return a data.frame with resp. the names of the topics and the dimensions.

```
topic_vars <- function(props)
  props %>%
  filter(Type=='Topic') %>%
  select(Key)

dim_vars <- function(props) {
  props %>%
    filter(Type %in% c('Dimension','TimeDimension','GeoDimension')) %>%
    select(Key)
}
```

Functions related to CBS maps and preparing of plot data set

plot_NL

This function merges the information of a CBS map file with a data.frame

```
library(magrittr)
library(dplyr)
library(ggplot2)
library(rgdal)
library(rgeos)
library(maptools)

plot_NL <- function (dsn,                     # map data set to use
        data_df,                              # data to use
        plot_var,                             # name variable to plot
        plot_varc      = plot_var,            # caption variable to plot
        data_linkvar   = 'RegioS_coded',      # name variable to link to map
        labels         = F,                   # labels to plot?
        label_var      = NULL,                # name variable that contains labels
        co             = F)                   # check overlap labels?
  {
  mylayers              = ogrListLayers(dsn)
  gp                    = readOGR(dsn, mylayers[1], disambiguateFIDs = T)
  gp                    = spTransform(gp, CRS("+init=epsg:4238"))
  gp.f                  = fortify(gp)
  gp@data$id            = rownames(gp@data)
  gp@data$gp_code       = as.character(gp@data$statcode)
  data_df$plot_var      = data_df[,plot_var]
  if (!is.null(label_var)) {
    data_df$label_var = data_df[,label_var]
  }
  gp@data               =  gp@data %>%
    inner_join(data_df, by = c('gp_code' = data_linkvar))
  gp.f                  = merge(gp.f, gp@data, by.x = "id", by.y = "id") %>%
    arrange(group,piece,order)

  if (labels==T) {
    gp.n = gp.f %>%
      group_by(statnaam) %>%
      summarize(
        minlat  = min(lat),
        maxlat  = max(lat),
        minlong = min(long),
        maxlong = max(long),
        label_var  = first(label_var)
      ) %>%
      mutate(cenlat  = (minlat + maxlat) / 2,
        cenlong = (minlong + maxlong) / 2) %>%
      select(statnaam, lat = cenlat, long = cenlong,label_var)
  }

  p = ggplot(gp.f,
    aes(long, lat, color = plot_var)) +
    geom_polygon(aes(group = group, fill = plot_var), color = 'black' )  +
    labs(x = "longitude", y = "latitude") +
    scale_fill_distiller(plot_varc,palette = "Spectral") +
    scale_x_continuous(labels=format_WE) +
    scale_y_continuous(labels=format_NS)
  if  (labels==T) {
    p = p +   geom_text(data = gp.n, aes(label = label_var),
        color = 'black', check_overlap = co)
  }
  return(p)
}
```

dsn_map

This function selects one of the three region map files that I have downloaded from the nationaalgeoregister website. See Downloading a mapfile for details.

```
dsn_map <- function(gem_prov) {
  if (gem_prov == 'P') {
    dsn        = "D:/data/maps/cbs_provincies.gml"
  } else if (gem_prov == 'C') {
    dsn        = "D:/data/maps/cbs_coropplusgebied.gml"
  } else {
    dsn        = "D:/data/maps/cbs_gemeenten.gml"
  }
}
```

Formatting functions

These functions format the values of the longitude and lattitude axes of the plot.

```
format_WE <- function(x,dig=3) {
  format_WENS(x,'WE',dig)
}

format_NS <- function(x,dig=3) {
  format_WENS(x,'NS',dig)
}

format_WENS <- function(x,WENS,dig=3) {
  if (WENS=='WE') {
    Z1 = 'W' ; Z2 = 'E'
  } else {
    Z1 = 'S' ; Z2 = 'N'
  }
  f  = sprintf('%%.%.0ff',dig)
  xf = sprintf(f,abs(x))
  Z  = ifelse(x < 0, Z1, ifelse(x > 0, Z2,''))
  # e= bquote(.(xf)*degree*~.(quote(Z)))
  # e=as.expression(e) # fails in plot why ?
  f = parse(text = paste(xf, "*degree~", Z),keep.source = F)
}
```

Downloading a mapfile

To download a CBS mapfile:

- press the download button on CBS gebiedsindelingen
- in the pop-up window specify in 'Kies een kaartlaag:' which map you want to download. I did choose e.g. 'cbs_provincie_2015_gegeneraliseerd_voorlopig' and 'cbs_gemeente_2015_gegeneraliseerd_voorlopig'
- in the same window specify the output format. I did choose 'GML 2'.
- I did not change the other fields and pressed then the 'Download data' button

I got the output in my browser window and saved that with a 'gml' extension. In the browser you also see the request that was generated. You could use that directly (not interactive) with the following lines of code:

```
map_url  = "http://geodata.nationaalgeoregister.nl/cbsgebiedsindelingen/wfs"
map_qry  =  paste0("?&REQUEST=GetFeature&SERVICE=WFS&VERSION=1.1.0&",
                  "TYPENAME=cbs_coropplusgebied_2015_gegeneraliseerd_voorlopig&",
                  "SRSNAME=EPSG:28992&OUTPUTFORMAT=GML2")
r        = curl_fetch_memory(paste0(map_url, curl_escape(map_qry)))
x        = rawToChar(r$content)
map_fle = file("D:/data/maps/cbs_coropplusgebied.gml",open="wt")
writeChar(x,map_fle)
close(map_fle)
```