Reading TED Europe csv data with R

Han Oostdijk

22 april 2016

Abstract

This document shows how R code can be used to download a large file with irregular comma separated values that is transformed to a *data.frame* and loaded from there in a MongoDB database. It also gives some examples to retrieve information from that database with JSON statements. Also is shown how to download, unzip and read a zipped internet file in xls format. Packages used: **mongolite**, **jsonlite**, **dpyr**, **magrittr** and **openxlsx**.

Introduction

The Tenders Electronic Daily (TED) database covers public procurement for the European Economic Area, Switzerland, and the former Yugoslav Republic of Macedonia. This document shows how to read the csv version of the TED database. Broader coverage is also available in XML format.

This document shows how to read the csv version that is a large comma separated file with embedded separators and sometimes a missing last field. Because of this irregular csv format the conversion to a data.frame is not trivial. Therefore we only do this conversion code when it is necessary. In this case (being Dutch) I am only interested in procurements in the Netherlands and therefore I (pre-) select records containing *',"NL",'*. The reader is advised to do a similar preselection but this is not necessary (when the amount of available memory is sufficient). Because of possible memory constraints I read batches of lines from the csv file and convert them to a *data.frame*. From the data.frame I select the records and fields that I need and add them to the data that was read earlier. Here I write the information to a MongoDB collection, but it is of course possible to keep the information in data.frame format.

In the remainder of the document we first list the functions we created to select the information. At the end of the document we show how these functions are used. In the last part of the document I give some examples of how the information can be queried with MongoDB commands.

R libraries used

```
library(dplyr)
library(magrittr)
library(mongolite) # if one wants to use mongolite
library(jsonlite)  # if one wants to use mongolite
library(openxlsx)
```

Function for downloading the csv file

It is possible to skip the download part of the procedure and use the **readLines** function directly on the url of the csv files but I advise downloading the file:

- in case of reruns a local version will run much faster
- it is easier for checking results and debugging errors and small subsets can be made for testing

The location of the files can be found on the Tenders Electronic Daily (TED) (csv subset) – public procurement notices website. There is a version for each of the years 2006 up to 2015 and a combined version for the periode 2009-2015. In this document we only consider the *contract award notices*.

```
ted_download_csv <- function (year,outfile) {
  # year 2006 ... 2015 or 2009-2015
  url = paste0('http://open-data.europa.eu/repository/ec/dg-grow/mapps/TED_CAN_',
    year,'.csv')
  download.file(url, destfile = outfile)
}
```

Utility functions

Because the downloaded files are large (for the combined 2009-2015 file about 1.6 GB) they can't be easily viewed in an editor or notebook. I used the following utility functions to create subsets or view a part of the large file:

```
ted_copy_csv <- function (infile,outfile,skip=0,nrows=5) {
  g     = file(outfile, open = 'wt')
  on.exit(close(g), add = FALSE)
  d     = ted_read_skip(infile,skip=0,n=5)
  writeLines(d,g,useBytes = T)
  invisible(NULL)
}

ted_read_skip <- function (infile,skip=0,n=5) {
  f     = file(infile, open = 'rt')
  on.exit(close(f))
  if (skip > 0)   d = readLines(f,n=skip,encoding ='UTF-8')
  d   = readLines(f,n=n,encoding ='UTF-8' )
}
```

Selection function

Because we are not interested in all the information in the csv file we define the function **fun_sel** that will select the necessary data from the *data.frame* that we create in each step. When we need all the information we will use **fun_id**.

```
fun_sel <- function (df) {
  df %>%
    filter(ISO_COUNTRY_CODE=='NL') %>%
    select(ID_NOTICE_CAN,DT_DISPATCH,CAE_NAME,CAE_ADDRESS,
           CAE_TOWN,CAE_POSTAL_CODE,ISO_COUNTRY_CODE,
           CPV,ADDITIONAL_CPVS,VALUE_EURO,VALUE_EURO_FIN_1,
           VALUE_EURO_FIN_2,ID_AWARD,WIN_NAME,WIN_ADDRESS,
           WIN_TOWN,WIN_POSTAL_CODE,WIN_COUNTRY_CODE,
           AWARD_EST_VALUE_EURO,AWARD_VALUE_EURO,
           VALUE_EURO_FIN_1_1,DT_AWARD) %>%
    mutate(ID_NOTICE_CAN=as.character(ID_NOTICE_CAN),
           CPV = as.character(CPV) )
}

fun_id  <- function (df) {
  df %>%
```

```
    mutate(ID_NOTICE_CAN=as.character(ID_NOTICE_CAN),
           CPV = as.character(CPV) )
    }
```

Function for correcting the csv records

To include each csv record as a row in a *data.frame* we have correct the following issues:

- sometimes the last field is missing. When that is the case we simply add a blank
- sometimes fields contain embedded commas as in *"39160000,45261100,45261213,45421100"* where a list of CPV (Common Procurement Vocabulary) items is given. In this case (only) we replace the commas with blanks.
- therefore we need to know if the commas are inside or outside double quotes. We can do that by splitting the line at the places of the double quotes. Commas in the *even* parts of the splitted line can be replaced.
- however this goes wrong in the case of an empty string: a pair of double quotes. Therefore we temporarily replace such pairs with *$$*.
- if the last character of the line is a double quote, it would be removed by this procedure. Therefore is it added in that case. (Not applicable with this file.)

```
c2b <- function (cs) {
  if (substr(cs,nchar(cs),nchar(cs)) == ",")
      cs = paste0(cs, ' ')
  x1 = gsub('\"\"','$$',cs,fixed=T)
  x1 = strsplit(x1,'"',fixed=T)[[1]]
  i2 = seq.int(2,length(x1),2)
  x1[i2] = gsub(',',' ',x1[i2])
  x1 = paste(x1,collapse='"')
  if (substr(cs,nchar(cs),nchar(cs)) == "\"") {
    x1 = paste0(x1,"\"")
  }
  x1 = gsub('$$','\"\"',x1,fixed=T)
}
```

Function for converting the csv file to data.frame and writing to Mongodb database

The function **ted_read_csv** ensures that the necessary data is selected from the *data.frames* read by **ted_read_table** and that they are written to the MongoDb collection or a *data.frame* (depending on parameter *m*). The actual reading of the data, correction of the data lines and the creation of a data.frame for a block of input lines is done in function **ted_read_table**. Because the file is large and we need only a subset of the data, the records are read in blocks so that each time only a limited amount of data is in memory.

```
ted_read_csv <-
  function (
    infile,          # filename of csv file
    fun,             # function that does a selection on the data.frame
    m = NULL,        # MongoDB collection object or NULL
    blksize = 1e5,   # size of blocks to be read by ted_read_table
    prepsel = NULL   # preselection statement (for use with grepl)
    ) {
    use_mongo =  'mongo' %in% class(m)
    if (use_mongo == T) {
```

```
      written1 = 'written to mongodb :' ; written2 = 'documents'
    } else {
      written1 = 'added to data.frame :'; written2 = 'rows'
    }
    f     = file(infile, open = 'rt')
    on.exit(close(f))
    # read data line with header (and therefore no prepsel!)
    df = ted_read_table(f, header = T, nrows = 1, sep = ',' ,
       ncols = NULL, encoding = 'UTF-8'
    )
    df1      = df$df          # data.frame with first line
    totlines = df$numread     # lines read : 2 (header + data line)
    cn       = names(df1)     # names of columns (before selection)
    nc       = length(cn)     # number of columns (before selection)
    nr       = dim(df1)[1]    # initialize nr for while loop
    df1      = fun(df1)       # do selection on data.frame
    while (nr > 0) {          # go on if we did read data lines in last pass
      df =  ted_read_table(f, header = F, nrows = blksize, sep = ',' ,
          ncols = nc, encoding = 'UTF-8' ,
          prepsel = prepsel,  col.names = cn
        )
      df2 = df$df             # data.frame with block of data lines
      totlines = totlines + df$numread # lines read up till now
      cat(paste('lines read (including header) :', sprintf('%.0f', totlines)), '\n')
      nr  = dim(df2)[1]       # number of rows in resulting block
      if (nr > 0) df2  = fun(df2)  # do selection of data.frame
      nr  = dim(df2)[1]       # number of rows in resulting block
      if (nr > 0) {           # if non-empty data.frame
        if (use_mongo == T) {
          dummy = m$insert(df2) # write to mongodb collection m
        } else {
          df1 = rbind(df1, df2) # append to existing rows
        }
        cat(paste(written1, sprintf('%.0f', dim(df2)[1]), written2), '\n')
      }
      nr = df$numread         # number of lines read this pass
    }
    if (use_mongo == T) {
      invisible(NULL)
    } else {
      df1
    }
  }

ted_read_table <- function(
  f,                          # file connection to input csv file
  header = F,                 # first line is a header line?
  nrows = 1000,               # number of lines to read from csv file
  sep = ',',                  # separator between fields
  ncols = NULL,               # number of colums for a full line
  encoding = 'UTF-8',         # encoding to use
  prepsel = NULL,             # preselection statement (for use with grepl)
  ...                         # can be used for col.names
  ) {
  # initialize in case we are finished or get an error
  t1 = NULL
  df = data.frame()
  nrows = ifelse(header == T, nrows + 1, nrows)
```

```
  tryCatch({
    t1   = readLines(f, n = nrows, encoding = encoding)
  },
  error = function(e) {
    cat(as.character(e), '\n')
    cat('possibly data lost', '\n')
  })
  numread = length(t1)       # number of lines read
  if (is.null(t1) | length(t1) == 0) {
   return(list(df = df, numread = numread)) # nothing read: return
  }
  # do preselection
  if (!is.null(prepsel))
    t1   = t1[grepl(prepsel, t1)]
  if (is.null(t1) | length(t1) == 0) # nothing left after preselection: return
   return(list(df = df, numread = numread))
  # do correction of lines
  t1   = sapply(t1, c2b, USE.NAMES = F)
  # if we know the number of columns we need
  if (!is.null(ncols)) {
    cnc = sapply(t1, function(x)
      { length(strsplit(x, ',', fixed = T)[[1]]) == ncols })
    t1f = t1[!cnc] # lines with wrong number of columns
    if (length(t1f)> 0)
        print(t1f)      # print them
    t1 = t1[cnc]    # and go on with correct lines
  }
  if (is.null(t1) | length(t1) == 0) {
    # nothing left after column length check
    return(list(df = df, numread = numread))
  } else {
    # convert the correct lines to data.frame
    tc   = textConnection(t1)
    df   = read.table(tc, header = header, sep = sep , quote = '\"',
               comment = '', stringsAsFactors = F, ...)
    return(list(df = df, numread = numread))
  }
}
```

Opening the Mongodb database

If one does not want to use a Mongodb database the function **ted_read_csv** can be used with the default value for *m* i.e. *NULL*. In that case **ted_read_csv** will return a *data.frame* with the selected information. Otherwise the results are written to a collection in a Mongodb database. We open the collection *tedcsv* in database *ted* on the local Mongodb server. If the database and/or collection do not exist they will be created. We will add 'documents' to the collection and therefore we use the *drop* method to ensure that the collection is empty before we start with this.
Be sure that the Mongodb server is started.

For information about Mongodb see:

- The MongoDB Manual. We used versie 3.2.2 of the software.
- The R package that we use: mongolite by Jeroen Ooms.

```
m_ted     = mongo(collection = "tedcsv", db = 'ted', verbose = F)
tryCatch({d=m_ted$drop()}, error = function(e) {invisible(NULL)})
cat(paste('documents in collection:',m_ted$count()),'\n')
```

```
## documents in collection: 0
```

Read the csv file in the MongoDB collection

We download the csv file from the TED site into a local copy

```
local_csv_file = "../newtestjes/ted/ted_2009_2015.csv"
```

```
ted_download_csv('2009_2015',local_csv_file)
```

and insert the selected information in the MongoDB collection.

```
ted_read_csv(local_csv_file, fun_sel, m_ted, blksize= 6e5, prepsel=',"NL",')
```

```
## lines read (including header) : 600002
## written to mongodb : 7639 documents
## lines read (including header) : 1200002
## written to mongodb : 8990 documents
## lines read (including header) : 1800002
## written to mongodb : 8193 documents
## lines read (including header) : 2400002
## written to mongodb : 6957 documents
## lines read (including header) : 3000002
## written to mongodb : 8967 documents
## lines read (including header) : 3369323
## written to mongodb : 5306 documents
## lines read (including header) : 3369323
```

```
cat(paste('documents in collection:',m_ted$count()),'\n')
```

```
## documents in collection: 46052
```

Or alternatively read the csv file in a *data.frame*

```
df1 = ted_read_csv(local_csv_file, fun_sel, NULL, blksize= 1e5, prepsel=',"NL",')
cat(paste('rows in data.frame:',dim(df1)[1]),'\n')
```

From now on we assume that we have read the data in a MongoDB collection

Get information from the MongoDB collection

In the next sections we will give some examples for getting information from the **tedcsv** collection and in the last example of the combination of the **tedcsv** and the **cpv** (to be created in example 6) collections. We will use the word *query* for a selection of the documents and the word *projection* for a selection of fields. The specification of information that has to be retrieved from the database is done with JSON statements.

Example 0 : empty query

When we do a empty *query* with no *projection* we get all fields from all documents in the collection. Because that is a little too much, we do not execute the query in the next box:

```
q  = '{}'
df = m_ted$find(q)
```

Example 1: empty query with limited output

The same as before but now limited to documents 4 and 5. Because of the width of the result table (Table 1 on page 7) it is printed very small. (The code to print the tables is given in the rmd file.)

```
q  = '{}'
df = m_ted$find(q, skip=3, limit=2)
```

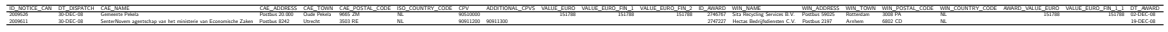

Table 1: Example 1: empty query with limited output

Example 2: empty query with selected fields

Now use a *projection* to specify which fields will be shown. In Table 2 on page 7 we see that indeed only the three fields selected are present. The object id is explicitly not included. Again we limit the number of documents to two.

```
q  = '{}'
p  = '{"_id" :0 , "ID_NOTICE_CAN" :1, "CAE_NAME" : 1 , "CPV" : 1}'
df = m_ted$find(q, p, limit=2)
```

ID_NOTICE_CAN	CAE_NAME	CPV
2009318	Nederlands Forensisch Instituut (NFI)	38433100
2009330	TNO Industrie en Techniek	38810000

Table 2: Example 2: empty query with selected fields

Example 3: empty query with selected fields and a sort

The same *query* and *projection* but we have added a *sort*: the documents that satisfy the *query* (i.e. all because the query is empty) are sorted by descending CPV. After sorting we take from documents four and five the selected fields. Table 3 on page 7 shows the results.

```
q  = '{}'
p  = '{"_id" :0 , "ID_NOTICE_CAN" :1, "CAE_NAME" : 1 , "CPV" : 1}'
s  = '{"CPV" : -1}'
df = m_ted$find(q, p, skip=3, limit=2, sort=s)
```

ID_NOTICE_CAN	CAE_NAME	CPV
20138710	Bizob namens de gemeente Cranendonck en de gemeente Heeze-Leende	98514000
20138710	Bizob namens de gemeente Cranendonck en de gemeente Heeze-Leende	98514000

Table 3: Example 3: empty query with selected fields (sorted)

Example 4: query with selected documents and fields

We use the same *projection* but now do a single *query*: select only the documents that contain the word *Amstelveen* in the *CAE_NAME* field. Again we limit the number of documents to two. Results can be found in Table 4 on page 8.

```
q  = '{"CAE_NAME":  { "$regex" : "Amstelveen"} }'
p  = '{"_id" :0 , "ID_NOTICE_CAN" :1, "CAE_NAME" : 1 , "CPV" : 1}'
df = m_ted$find(q, p,limit=2)
```

ID_NOTICE_CAN	CAE_NAME	CPV
20119344	Gemeente Amstelveen	3410000
20128325	Gemeente Amstelveen	66515100

Table 4: Example 4: single query

Example 5: composite query with selected documents and fields

As in example 4 but we further restrict the documents: we only want documents concerning 2012. Those documents have a *ID_NOTICE_CAN* field that begins with '2012'. We have to combine the old and new restriction with the *$and* operator. Results can be found in Table 5 on page 8.

```
q1 = '{"CAE_NAME":  { "$regex" : "Amstelveen"} }'
q2 = '{"ID_NOTICE_CAN":  { "$regex" : "^2012"} }'
q  = paste('{"$and" : [',q1,',',q2,']}')
p  = '{"_id" :0 , "ID_NOTICE_CAN" :1, "CAE_NAME" : 1 , "CPV" : 1}'
df = m_ted$find(q, p,limit=2)
```

ID_NOTICE_CAN	CAE_NAME	CPV
20128325	Gemeente Amstelveen	66515100
201249332	Gemeente Amstelveen	50510000

Table 5: Example 5: composite query

Example 6: create a collection with the CPV codes

In the csv file with TED information the CPV's (Common Procurement Vocabulary or activity descriptions) are given in a code. We will first download the descriptions. The Dutch description will be loaded in this section unless the code of one of the other European languages is specified in *lang_desc*. In the next example we will show how this information can be linked to the TED information when we query that information. At the end of this example we display some high level CPV code in Table 6 on page 9.

```
cpv_url = 'https://simap.ted.europa.eu/documents/10184/36234/cpv_2008_xls.zip'
exdir   = "../newtestjes/ted"
cpv_zip = paste0(exdir,"/cpv.zip")
```

```
download.file(cpv_url, destfile = cpv_zip)          # download the zip file
unzip(cpv_zip,exdir=exdir)                          # unzip the xls file
```

```
dfiles  = unzip(cpv_zip,list=T)                     # determine the name of the xls file
lang_desc = 'NL'                                    # language of description
cpv_df = read.xlsx(paste0(exdir,'/',dfiles[1]))   %>% # read first sheet of xls file
        select_(.dots = c('CODE',DESC=lang_desc)) %>% # select the Dutch description
        mutate(CODE = substr(CODE,1,8))               # extract code (8 characters)
```

```
m_cpv <- mongo("cpv_codes", db='ted', verbose = F) # open MongoDB collection
tryCatch({d=m_cpv$drop()}, error =                 # empty contents collection
    function(e) {invisible(NULL)})
d = m_cpv$insert(cpv_df)                           # insert data.frame
m_cpv$index(add = "CODE")                          # create index for CODE
```

```
##   v key._id key.CODE   name            ns
## 1 1       1       NA   _id_ ted.cpv_codes
## 2 1      NA        1 CODE_1 ted.cpv_codes
```

```
#m_cpv$index(remove = "CODE_1")
cat(paste('documents in collection:',m_cpv$count()),'\n')
```

```
## documents in collection: 9454
```

```
df = m_cpv$find('{ "CODE": { "$regex" : "0000000$"} }')
```

CODE	DESC
30000000	Kantoormachines en gegevensverwerkende apparatuur, kantooruitrusting en -benodigdheden, uitgez. meubilair en softwarepakketten
50000000	Reparatie- en onderhoudsdiensten
60000000	Vervoersdiensten (uitg. vervoer van afval)
70000000	Makelaarsdiensten
80000000	Diensten voor onderwijs en opleiding
90000000	Diensten inzake afvalwater, afval, reiniging en milieu

Table 6: Example 6: high level CPV codes

Example 7: aggregation pipeline

All the previous examples used the *find* method. By using the *aggregate* method we can specify a sequence of operations (a pipeline) that has to be done on a collection. In this example we will first do a *$match* (comparable with the *find* method), then a *$lookup* to 'join' the description of a CPV, then a *$project*, a *$skip* to omit the first three documents, a *$limit* to include only the first two documents and at last an *$out* to write the resulting information to a new collection. The collection needed for the *$lookup* was created in the previous example. Writing to a collection (as done here) is not necessary. Finally we print the contents of the new collection. See Table 7 on page 10.

```
df1 = m_ted$aggregate('[
  {"$match":{"$and" : [{"ID_NOTICE_CAN":  { "$regex" : "^2012"} },
                {"CAE_NAME":  { "$regex" : "Amstelveen"} }]}} ,
  {"$lookup" : { "from" : "cpv_codes" ,"localField" : "CPV",
                 "foreignField": "CODE", "as" : "CPV_DESC" }} ,
  {"$project" : { "_id": 0 , "ID_NOTICE_CAN": 1 , "CAE_NAME": 1,
                 "CPV": 1 , "desc" : "$CPV_DESC.DESC" } } ,
  {"$skip" : 3 } ,
  {"$limit" : 2 } ,
  {"$out": "Amstelveen" }
  ]' )

m_av = mongo("Amstelveen", db='ted', verbose = F)
df   = m_av$find()
```

ID_NOTICE_CAN	CAE_NAME	CPV	desc
2012110615	Gemeente Amstelveen	45214200	Schoolgebouwen
2012110616	Gemeente Amstelveen	45310000	Aanleg van elektriciteit

Table 7: Example 7: aggregation pipeline

Examples that show the use of MongoDB in MATLAB

Git repository matlab_mongodb shows examples of accessing MongoDB from the MATLAB environment. When you use this (Java based) interface you have to code the queries that in the **mongolite** interface can be specified directly by JSON statements (as seen above).

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] xtable_1.8-2    openxlsx_3.0.0  jsonlite_0.9.19 mongolite_0.8.1 magrittr_1.5
## [6] dplyr_0.4.3     knitr_1.12.3
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.4     digest_0.6.9    assertthat_0.1  R6_2.1.2        DBI_0.3.1
##  [6] formatR_1.3     evaluate_0.8.3  stringi_1.0-1   lazyeval_0.1.10 rmarkdown_0.9.5
## [11] tools_3.2.4     stringr_1.0.0   yaml_2.1.13     parallel_3.2.4  htmltools_0.3
```