BDDS R Tutorial; subsetting large datasets and merging metadata

This is a **Bulk Data Download Services (BDDS)** tutorial providing a walkthrough on how to process large CSV files using the **R** coding language. We selected R for it's popularity, open source access, relevance to data science and accessibility for beginners. The code only takes seconds to run (depending on the system) and uses around 150 MB of memory including R's overhead.

We assume that you have some basic understanding of coding or scripting in order to adapt the code to your needs, but we believe this tutorial is within the reach of novice coders. It is built to process any of the BDDS data packages with as few modifications as possible.

Context

We wanted to address users' feedback concerning issues when opening the larger datasets in **Excel** e.g. **Other Policy-Relevant Indicators** (**OPRI**) containing more than 2.7 million rows. Figure 1 shows the warning message popping-up when a CSV is partially loaded due to its size exceeding Excel's limit, in this case, exceeding the maximum number of rows.

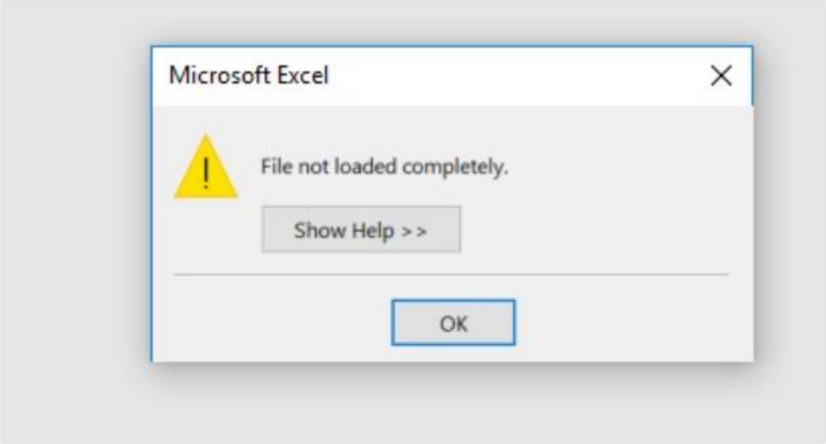

Figure 1 excel warning message: file not loaded completely.

The BDDS is meant to be accessed programmatically either with a statistical package (R, STATA, SAS, SPSS, etc.) or with a coding language. Although it is possible to load it initially in Excel with some manual workaround, we strongly discourage this practice to prevent human errors. Moreover, each BDDS package provides data point level metadata, and labels in separate CSV files. Linking these files with the core datasets requires efficiency and power that goes beyond Excel's standard capabilities.

Objective

This tutorial fulfils two main objectives. First, to process a large CSV in a programmatic way. Second, to demonstrate how the country/indicator labels and metadata are linked and merged to the data.

Specifically, we will process the **Other Policy-Relevant Indicators** BDDS files, providing a walkthrough with the following steps:

1. Read (load) the full CSV file in memory;
2. Create a subset based on lists of countries, indicators and years;
3. Merge indicator and country labels to the subset;
4. Merge metadata to a data subset;
5. Write the data subset back to CSV.

Getting started

Download the R code

Start by downloading the tutorial R code package (the link will start the download).

Download the BDDS data

You will also need the Other Policy-Relevant Indicators (OPRI) data package (available on the UIS BDDS page under 'Education').

Unzip the R code and the data files and take note of the path of the folder where those documents are saved, it will be required as an input to the code during this walkthrough.

Install the R software

There are many ways to get R, but we recommend the convenience of **Anaconda**. This software package includes multiple data science tools, such as R and other Integrated Developer Environments (IDE) like Jupyter Notebook. For this demonstration, we will use the **R Studio** IDE (within Anaconda) for running the code, looking at output tables, etc.

Go to the Anaconda download page and choose a download package for your system at the bottom of the page.

Opening the code

Install **Anaconda** → Open Anaconda → Click on the **Install** button in the R Studio section (Figure 2, the button will change from "Install" to "Launch") → **Launch** R Studio after it is installed

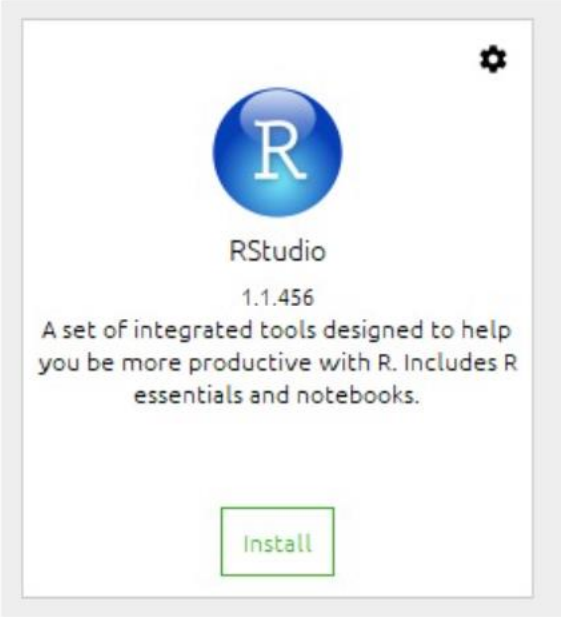

Figure 2 anaconda interface's Home page

Within R Studio: Go to **File** menu (Figure 3) → Click **Open file**→ Open the BDDS code from its saved location

Figure 3 opening a R file in R Studio IDE

This will load the R code in R Studio. Nothing is executed at this point.

Executing the code

In the following section, we will look at each block of code in detail. The code contains most of the necessary comments to understand how it works but we will go through each steps adding some colour to those comments.

The code uses two librairies (Figure 4) that include built-in functions for data manipulation.

```
5  # load libraries
6  library(dplyr)
7  library(readr)
```

Figure 4 librairies for data manipulation

- **dplyr** focusses on tools for working with data frames
- **readr** provide a fast and friendly way to read rectangular data

Specifying work directory and files names

```
# Work directory ############################################################
#Set work directory
setwd('C:/Users/15144/Downloads/OPRI/')
```

Figure 5 work directory

Figure 5 shows the work directory where the CSV files are saved on the computer.

Change the work directory to the location on your computer where the BDDS files are saved

This tutorial only uses the following CSV:

- OPRI_DATA_NATIONAL.csv - the core dataset with country data;
- OPRI_METADATA.csv - the metadata table related to the data points in the core dataset;
- OPRI_COUNTRY.csv - the table with country labels and;
- OPRI_LABEL.csv - the table with indicator labels.

```
# Loading CSV in memory ######################################################
#Read CSVs while specifying column type
dfNational <- read_csv('OPRI_DATA_NATIONAL.csv', na="", col_types = cols(
  INDICATOR_ID = col_character(),
  COUNTRY_ID = col_character(),
  YEAR = col_integer(),
  VALUE = col_double(),
  MAGNITUDE = col_character(),
  QUALIFIER = col_character()
))


dfCountryLabels <- read_csv('OPRI_COUNTRY.csv',  na="", col_types = cols(
  COUNTRY_ID = col_character(),
  COUNTRY_NAME_EN = col_character()
))

dfIndicatorLabels <- read_csv('OPRI_LABEL.csv',  na="", col_types = cols(
  INDICATOR_ID = col_character(),
  INDICATOR_LABEL_EN = col_character()
))


dfMetadata <- read_csv('OPRI_METADATA.csv',  na="", col_types = cols(
  INDICATOR_ID = col_character(),
  COUNTRY_ID = col_character(),
  TYPE = col_character(),
  METADATA = col_character()
))
```

Figure 6 loading CSV files to memory

The code in Figure 6 will load the CSV files to memory in a R data frame reproducing the original file's structure.

Modify the file names in the code when you want to load other datasets

The "OPRI_DATA_NATIONAL.csv" saved in the ***'dfNational'*** variable will look like the table in Figure 7.

	INDICATOR_ID	COUNTRY_ID	YEAR	VALUE	MAGNITUDE	QUALIFIER
1	10	ABW	2017	0	NA	NA
2	10	ABW	2018	0	NA	NA
3	10	ABW	2019	0	NA	NA
4	10	ABW	2020	0	NA	NA
5	10	AFG	2017	0	NA	NA
6	10	AFG	2018	0	NA	NA
7	10	AFG	2019	0	NA	NA
8	10	AFG	2020	0	NA	NA
9	10	AGO	2017	0	NA	NA
10	10	AGO	2018	0	NA	NA
11	10	AGO	2019	0	NA	NA
12	10	AGO	2020	0	NA	NA

Figure 7 an example of a dfNational data frame

Run the code from the beginning to the “Load CSV to memory” section (included) by selecting the lines and then pressing ctrl+enter.

You can now see the CSV loaded in the **'Environment'** window (Figure 8) within the R Studio IDE. Double click on the **dfCountryLabels** variable and the data frame it contains will pop-up in a new window.

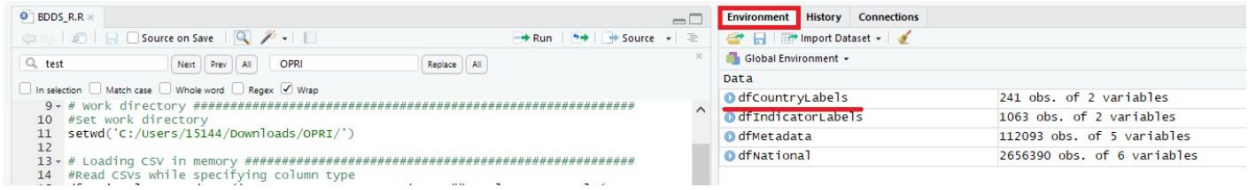

Figure 8 opening a variable containing a data frame in the 'Environment' window

Subsetting the data file

The second step is to extract a subset from a data frame based on lists of countries, years and indicators.

```
42 # Creating subsets of the data ##############################################
43 # 1) Extracting a vectors of sorted unique values for the Year, Country and Indicator variables
44 # Those vectors' values will serve as the default parameters in the following function
45 allYears        <- sort(unique(dfNational[,'YEAR'])[[1]])
46 recentYears     <- tail(allYears, n=4)
47 allCountries    <- sort(unique(dfNational[,'COUNTRY_ID'])[[1]])
48 allIndicators   <- sort(unique(dfNational[,'INDICATOR_ID'])[[1]])
```

Figure 9 extract and sort lists of unique values for years, countries and indicators

The first lines (Figure 9) sort and save lists of the unique values for countries, years and indicators found in dfNational. These lists will define the default parameters in the function named subsetData Figure 10.

```
51 # Data subset function
52 subsetData <- function(dataSet, yearList=recentYears, countryList=allCountries, indicatorList=allIndicators) {
53   # Subsets the data
54   #
55   # Parameters
56   # ----------
57   # dataSet : DataFrame
58   # a DataFrame to be subsetted
59   # yearList: a list of int, default is recentYears
60   # a list of years
61   # countryList: a list of str, defaults is allCountries
62   # a list of 3-letter ISO country code
63   # indicatorList: a list of str, default is allIndicators
64   # a list of indicator codes
65   # Returns
66   # -------
67   # DataFrame
68   # a DataFrame subsetted by a list of years, countries and indicators
69   aSubset <- dataSet %>% filter(
70     YEAR %in% yearList,
71     COUNTRY_ID %in% countryList,
72     INDICATOR_ID %in% indicatorList
73   )
74   return (aSubset)
75 }
```

Figure 10 subset function

The subsetData function (Figure 10) takes in four arguments: dataSet, yearList, countryList and indicatorList. Left at their default parameters, the subset will contain the last four years of data for all countries and all indicators.

The full dataset OPRI_DATA_NATIONAL.csv starts with more than 2.7 million rows. The default parameters output a data frame containing around 250K rows providing a subset that fits within an Excel sheet.

An example will be provided showing how to change these parameters to get a smaller (or larger) subset. Before running these examples, we will define another function for merging the metadata to the subset.

Merging the metadata to the data subset

```
# Add metadata to dataset function
addMetadata <- function(dataSub, metaDataSub, metadataType='Source:Data sources') {
  # Merges the metadata to the data
  #
  # Parameters
  # ----------
  # dataSub: DataFrame
  # a DataFrame receiving the metadata from another DataFrame
  # metaDataSub: DataFrame
  # a DataFrame giving metadata to another DataFrame
  # metadataType: str {'Source:Data sources','Under Coverage:Students or individuals'}
  # a string for specifying the type of metadata merged to the dataset (note
  # that the number of metadata type will vary across datasets and over time)
  #
  # Returns
  # -------
  # DataFrame
  # a DataFrame with an extra column of metadata
  metadataSubByType <- filter(metaDataSub, TYPE == metadataType) %>%  #filter metadata on metadata type
    group_by(YEAR,COUNTRY_ID, INDICATOR_ID, TYPE) %>%                 #Var on which to Group
    summarise(METADATA=paste(METADATA,collapse='|'))                  #Var that will be grouped
  dataSubsetWithMeta <- dataSub %>% left_join(metadataSubByType, by=c('YEAR', 'COUNTRY_ID', 'INDICATOR_ID'))
  return (dataSubsetWithMeta)
}
```

Figure 11 merging the metadata function

The **addMetadata** function (Figure 11) will take a dataset and merge it with its metadata.

The function takes in three arguments: dataSub, metadataSub and metadataType. Left at its default 3rd parameter, the function will add a column holding the data source to the subset.

The metadata is data point specific and the function will try matching a data point with the same YEAR/COUNTRY_ID/INDICATOR_ID combination within the metadata file.

Any data point can have zero, one or multiple metadata values associated with it. In the OPRI dataset, there are two types of metadata at the time of writing so a single data point could have a source or an under coverage entry associated with it. As such, it is important to run this function multiple time to get all the metadata types required for your needs. This function could also be modified to merge all metadata in one run.

Furthermore, a single datapoint can have multiple entries for the same type of metadata. This is also taken into account and if such a case occurs, the function will combine the entries within a single cell and each entries will be separated by the "|" symbol..

Run the code sections from "Creating subsets of the data" to "Merging metadata and data subsets" (included) by selecting the lines and then pressing 'ctrl+enter'. This will save the new functions to memory.

Example subsetting and merging the metadata to the subset

So far, we have constructed two functions that allow to subset the core data set and then match the metadata to each data point of the subset.

The next blocks of code provide examples running these two functions.

Example 1, running the subsetData function with the default parameters

```
# Example 1: Subsetting using the function's default parameters
# Data subset
defaultDataSubset = subsetData(dfNational)

# Metadata subset
defaultMetadataSubset = subsetData(dfMetadata)

# Merging metadata with data subset using default metadata type
defaultSubsetWithSource = addMetadata(defaultDataSubset, defaultMetadataSubset)
```

Figure 12 executing the subset and metadata functions using the default parameters

In this example (Figure 12), we only need to specify the dataset to be subsetted since we will otherwise run the **subsetData** function on its default parameters. This means we will extract the last 4 years of data, all indicators and all countries for both the core data set and the metadata set.

Next, we use the **addMetadata** function specifying the data subset and the metadata subset as the first two parameters. For the purpose of this example, we do not specify the third parameter since we will merge the data with the default metadata type, the source of the data.

RUN "Example 1" (Figure 12) by selecting the lines and then pressing 'ctrl+enter'.

This will save the data frames to variables (defaultDataSubset, defaultMetadataSubset, defaultSubsetWithSource) which can be visualized in the 'Environment' window.

Example 2, running the subsetData function with custom parameters

```
# Example 2:# Subsetting by specifying all the parameters into a list
# Defining the list of year, country and indicator of interest
yearsSubset   <- c(2012, 2014, 2015, 2017)
countrySubset <- c('ARG', 'KWT', 'SWE', 'ZWE')
indicSubset   <- c('NART.1.Q1.F.LPIA', 'MYS.1T8.AG25T99', 'GER.1', 'SAP.1', 'AIR.1.Glast')

# Data subset
myDataSubset = subsetData(dfNational, yearsSubset, countrySubset, indicSubset)

# Metadata subset
myMetadataSubset = subsetData(dfMetadata, yearsSubset, countrySubset, indicSubset)

# Merging metadata with data subset using specified metadata type
mySubsetWithUnderCov = addMetadata(myDataSubset, myMetadataSubset, 'Under Coverage:Students or individuals')

mySubsetWith_UnderCov_Source = addMetadata(mySubsetWithUnderCov, myMetadataSubset)
```

Figure 13 executing the subset and metadata functions using custom parameters

Figure 13 shows how the function works with all the parameters and the parameters' inputs explicitly stated.

First, custom lists of years, countries and indicators are specified and saved in variables (yearsSubset, countrySubset and indicSubset). These variables are then passed in the **subsetData** function as the second, third and fourth parameters respectively.

Once the dataset and the metadata set have been subsetted, the code merges the metadata and data subsets a first time with the **addMetadata** function. The third parameters is a string specifying 'Under Coverage: Students or individuals' as the type of metadata merged to the subset.

To further the example, the **addMetadata** is then used a second time to add the default metadata type (data source).

The final output is a subset with extra columns with both the source and the under-coverage metadata.

Run "Example 2" (Figure 13) section by selecting the lines and then pressing 'ctrl+enter'.

This will save the data frames to variables.

Adding labels

```
131  # Add label to dataset function
132  addLabels <- function(dataSetNoLabel, labelSet, keyVariable) {
133    # Adds labels to a dataset
134    # Adds an additional column with the country or indicators name.
135    #
136    # Parameters
137    # ----------
138    # dataSetNoLabel: DataFrame
139    # the DataFrame containing the data
140    # labelSet: DataFrame
141    # the DataFrame containing the labels
142    # keyVariable: str {'INDICATOR_ID', 'COUNTRY_ID'}
143    # a string specifying the key variable for the merge
144    #
145    # Returns
146    # -------
147    # DataFrame
148    # a DataFrame with extra columns for labels
149    dataSetWithLabels <- dataSetNoLabel %>% left_join(labelSet, by=keyVariable)
150    return (dataSetWithLabels)
151  }
```

Figure 14 adding labels function with an example

Before converting back the subset with metadata to CSV, the label for the country and indicator can be attached to the subset. This function will merge the labels from the label dataset to the subset.

The function (Figure 14) takes in three arguments and has no default parameters. The first and second arguments are for specifying the data frames for the subset and label tables. The third argument indicates the key on which to merge the data frames.

To get both countries and indicators labels you will need to run the function twice.

The output is a dataset with extra columns with those labels.

Run the "Adding labels" section (function and example) by selecting the lines and then pressing 'ctrl+enter'.

This will save the function in memory and run the examples for merging labels.

The function is run twice to include both country and indicator labels. You can now visualize the data frame with labels in the 'Environment' window.

Exporting subset to CSV

```
166  # Export subset to CSV ##########################################################################
167  write.csv(mySubsetWith Meta allLabels, na="", "R tutorial test.csv")
```

Figure 15 exporting the final subset to CSV

Lastly, we export the subset back to CSV (Figure 15). Note that R processes missing values as an R object named NA. In order not to confuse it with the UIS dataset string "NA" (used for not applicable) we specify the R NA object as an empty string "" when writing the subset back to CSV.

Modify the file name in the code to a name of your choice and run the last line of code to export the final subset to CSV.

You now have a subset with the metadata and labels included. The subset is saved in the work directory specified at the beginning of the code.