Data Handling: Import, Cleaning and Visualisation

Lecture 7: Data Sources, Data Gathering, Data Import

Prof. Dr. Ulrich Matter
(University of St.Gallen)

18/11/2021

---

1 Putting it all together

In this lecture we put the key concepts learned so far (text-files for data storage, parsers, encoding, data structures) together and apply them in order to master the first key bottleneck in the data pipeline: how to *import* raw data from various sources and *export/store* them for further processing in the pipeline.

1.1 Sources/formats

In the first two lectures we have learned how data is stored in text-files and how different data structures/formats/syntaxes help to organize the data in these files. Along the way, we have encountered key data formats that are used in various settings to store and transfer data:

- CSV (typical for rectangular/table-like data)
- Variants of CSV (tab-delimited, fix length, etc.)
- XML and JSON (useful for complex/high-dimensional data sets)
- HTML (a markup language to define the structure and layout of webpages)
- Unstructured text

Depending on the *data source*, data might come in one or the other form. With the increasing importance of the Internet as a data source for economic research, handling XML, JSON, and HTML properly is becoming more important. However, in applied economic research various other, and more specific formats can be encountered:

- Excel spreadsheets (`.xls`)
- Formats specific to statistical software packages (SPSS: `.sav`, STATA: `.dat`, etc.)
- Built-in R datasets
- Binary formats

While we will cover/revisit how to import all of these formats here, it is important to keep in mind that the learned fundamental concepts are as important (or even more important) than knowing which function to call in R for each of these cases. New formats might evolve and become more relevant in the future for which

no R function yet exists. However, the underlying logic of how formats to structure data work will hardly change.

2 Data gathering procedure

Before we set out to gather/import data from diverse sources, we should start organizing the procedure in a R script. This script will be the beginning of our pipeline!

First, open a new R script in RStudio and save it as `import_data.R` in your `code` folder. Take your time to meaningfully describe what the script is all about in the first few lines of the file:

```
##############################################################################
# Data Handling Course: Example Script for Data Gathering and Import
#
# Imports data from ...
# Input: links to data sources (data comes in ... format)
# Output: cleaned data as CSV
#
# U. Matter, St.Gallen, 2019
##############################################################################
```

RStudio recognizes different sections in a script, whereby section headers are indicated by `----------`. This helps to further organize the script into different tasks. Usually, it makes sense to start with a 'meta' section in which all necessary packages are loaded and fix variables initiated.

```
##############################################################################
# Data Handling Course: Example Script for Data Gathering and Import
#
# Imports data from ...
# Input: links to data sources (data comes in ... format)
# Output: cleaned data as CSV
#
# U. Matter, St.Gallen, 2019
##############################################################################


# SET UP --------------
# load packages
library(tidyr)

# set fix variables
INPUT_PATH <- "/rawdata"
OUTPUT_FILE <- "/final_data/datafile.csv"
```

Finally we add sections with the actual code (in the case of a data import script, maybe one section per data source).

```
##############################################################################
# Project XY: Data Gathering and Import
#
# This script is the first part of the data pipeline of project XY.
# It imports data from ...
# Input: links to data sources (data comes in ... format)
# Output: cleaned data as CSV
#
```

```
# U. Matter, St.Gallen, 2019
###############################################################################


# SET UP ---------------
# load packages
library(tidyr)

# set fix variables
INPUT_PATH <- "/rawdata"
OUTPUT_FILE <- "/final_data/datafile.csv"


# IMPORT RAW DATA FROM CSVs -------------
```

3 Loading/importing rectangular data[1]

3.1 Loading built-in datasets

We start with the simplest case of loading/importing data. The basic R installation provides some example datasets to try out R's statistics functions. In the introduction to visualization techniques with R as well as in the statistics examples in the lectures to come, we will rely on some of these datasets for the sake of simplicity. Note that the usage of these simple datasets shipped with basic R are very helpful when practicing/learning R on your own. Many R packages use these datasets over and over again in their documentation and examples. Moreover, extensive documentations and tutorials online also use these datasets (see for example the ggplot2 documentation). And, they are very useful when searching help on Stackoverflow in the context of data analysis/manipulation with R, as you should provide a code example based on some data that everybody can easily load and access.

In order to load such datasets, simply use the `data()`-function:

```
data(swiss)
```

In this case, we load a dataset called `swiss`. After loading it, the data is stored in a variable of the same name as the dataset (here '`swiss`'). We can inspect it and have a look at the first few rows:

```
# inspect the structure
str(swiss)
```

```
## 'data.frame':    47 obs. of  6 variables:
##  $ Fertility       : num  80.2 83.1 92.5 85.8 76.9 76.1 83.8 92.4 82.4 82.9 ...
##  $ Agriculture     : num  17 45.1 39.7 36.5 43.5 35.3 70.2 67.8 53.3 45.2 ...
##  $ Examination     : int  15 6 5 12 17 9 16 14 12 16 ...
##  $ Education       : int  12 9 5 7 15 7 7 8 7 13 ...
##  $ Catholic        : num  9.96 84.84 93.4 33.77 5.16 ...
##  $ Infant.Mortality: num  22.2 22.2 20.2 20.3 20.6 26.6 23.6 24.9 21 24.4 ...
```

```
# look at the first few rows
head(swiss)
```

```
##              Fertility Agriculture Examination Education Catholic Infant.Mortality
## Courtelary        80.2        17.0          15        12     9.96             22.2
## Delemont          83.1        45.1           6         9    84.84             22.2
```

[1] This section is based on Matter (2018a).

```
## Franches-Mnt       92.5       39.7          5          5    93.40            20.2
## Moutier            85.8       36.5         12          7    33.77            20.3
## Neuveville         76.9       43.5         17         15     5.16            20.6
## Porrentruy         76.1       35.3          9          7    90.57            26.6
```

To get a list of all the built-in datasets simply type `data()` into the console and hit enter. To get more information about a given dataset use the help function (e.g., `?swiss`)

3.2 Importing rectangular data from text-files

In most cases of applying R for data analysis, students and researchers rely on importing data from files stored on the hard disk. Typically, such datasets are stored in a text-file-format such as 'Comma Separated Values' (CSV). In economics one also frequently encounters data being stored in specific formats of commercial statistics/data analysis packages such as SPSS or STATA. Moreover, when collecting data on your own, you might rely on a spreadsheet tool like Microsoft Excel. Data from all of these formats can quite easily be imported into R (in some cases, additional packages have to be loaded, though). Thereby, what happens 'under the hood' is essentially the same for all of these formats. Somebody has implemented the respective *parser* as an R function which accepts a character string with the path or url to the data source as input.

3.2.1 Comma Separated Values (CSV)

Recall how in this format data values of one observation are stored in one row of a text file, while commas separate the variables/columns. For example, the following code-block shows how the first two rows of the `swiss`-dataset would look like when stored in a CSV:

```
"District","Fertility","Agriculture","Examination","Education","Catholic","Infant.Mortality"
"Courtelary",80.2,17,15,12,9.96,22.2
```

The function `read.csv()` imports such files from disk into R (in the form of a `data.frame`). In this example the `swiss`-dataset is stored locally on our disk in the folder `data`:

```
swiss_imported <- read.csv("data/swiss.csv")
```

Alternatively, we could use the newer `read_csv()` function which would return a `tibble`.

3.2.2 Spreadsheets/Excel

In order to read excel spreadsheets we need to install an additional R package called `readxl`.

```
# install the package
install.packages("readxl")
```

Then we load this additional package ('library') and use the package's `read_excel()`-function to import data from an excel-sheet. In the example below, the same data as above is stored in an excel-sheet called `swiss.xlsx`, again in a folder called `data`.

```
## New names:
## * `` -> ...1
```

```
# load the package
library(readxl)

# import data from a spreadsheet
swiss_imported <- read_excel("data/swiss.xlsx")
```

3.2.3 Data from other data analysis software

The R packages `foreign` and `haven` contain functions to import data from formats used in other statistics/data analysis software, such as SPSS and STATA.

In the following example we use `haven`'s `read_spss()` function to import a version of the `swiss`-dataset stored in SPSS' `.sav`-format (again stored in the folder called `data`).

```
# install the package (if not yet installed):
# install.packages("haven")

# load the package
library(haven)

# read the data
swiss_imported <- read_spss("data/swiss.sav")
```

4 Import and parsing with `readr`[2]

The `readr` package is automatically installed and loaded with the installation/loading of `tidyverse`. It provides a set of functions to read different types of rectangular data formats and is usually more robust and faster than similar functions provided in the basic R distribution. Each of these functions basically expects either a character string with path pointing to a file or a character string directly containing the data.

4.1 Basic usage of `readr` functions

For example, we can parse the first lines of the swiss dataset directly like this

```
library(readr)

read_csv('"District","Fertility","Agriculture","Examination","Education","Catholic","Infant.Mortality"
"Courtelary",80.2,17,15,12,9.96,22.2')
```

```
## # A tibble: 1 x 7
##   District   Fertility Agriculture Examination Education Catholic Infant.Mortality
##   <chr>          <dbl>       <dbl>       <dbl>     <dbl>    <dbl>            <dbl>
## 1 Courtelary      80.2          17          15        12     9.96             22.2
```

or read the entire `swiss` dataset by pointing to the file

```
swiss <- read_csv("data/swiss.csv")
```

```
## Parsed with column specification:
## cols(
##   District = col_character(),
##   Fertility = col_double(),
##   Agriculture = col_double(),
##   Examination = col_double(),
##   Education = col_double(),
##   Catholic = col_double(),
##   Infant.Mortality = col_double()
## )
```

[2] This is a summary of Chapter 8 in Wickham and Grolemund (2017).

In either case, the result is a `tibble`:

```
swiss
```

```
## # A tibble: 47 x 7
##    District     Fertility Agriculture Examination Education Catholic Infant.Mortality
##    <chr>            <dbl>       <dbl>       <dbl>     <dbl>    <dbl>            <dbl>
##  1 Courtelary        80.2        17            15        12     9.96             22.2
##  2 Delemont          83.1        45.1           6         9    84.8              22.2
##  3 Franches-Mnt      92.5        39.7           5         5    93.4              20.2
##  4 Moutier           85.8        36.5          12         7    33.8              20.3
##  5 Neuveville        76.9        43.5          17        15     5.16             20.6
##  6 Porrentruy        76.1        35.3           9         7    90.6              26.6
##  7 Broye             83.8        70.2          16         7    92.8              23.6
##  8 Glane             92.4        67.8          14         8    97.2              24.9
##  9 Gruyere           82.4        53.3          12         7    97.7              21
## 10 Sarine            82.9        45.2          16        13    91.4              24.4
## # ... with 37 more rows
```

The other `readr` functions have practically the same syntax and behavior. They are used for fixed-width files or csv-type files with other delimiters than commas.

4.2 Parsing and data types

From inspecting the `swiss` tibble printed out above, we recognize that `read_csv` not only correctly recognizes observations and columns (parses the csv correctly) but also automatically guesses the data type of the values in each column. The first column is of type double, the second one of type integer, etc. That is, `read_csv` also parses each column-vector of the data set with the aim of recognizing which data type it is. For example, the data value `"12:00"` could be interpreted simply as a character string. Alternatively, it could also be interpreted as a `time` format.

If `"12:00"` is an element of the vector `c("12:00", "midnight", "noon")` it must be interpreted as a character string. If however it is an element of the vector `c("12:00", "14:30", "20:01")` we probably want R to import this as a `time` format. Now, how can `readr` handle the two cases? In simple terms, the package first guesses for each column vector which type is most appropriate. Then, it uses a couple of lower-level parsing functions (one written for each possible data type) in order to parse each column according to the respective guessed type. We can demonstrate this for the two example vectors above.

```
read_csv('A,B
         12:00, 12:00
         14:30, midnight
         20:01, noon')
```

```
## # A tibble: 3 x 2
##   A      B
##   <time> <chr>
## 1 12:00  12:00
## 2 14:30  midnight
## 3 20:01  noon
```

Under the hood `read_csv()` used the `guess_parser()`- function to determine which type the two vectors likely contain:

```
guess_parser(c("12:00", "midnight", "noon"))
```

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

```
guess_parser(c("12:00", "14:30", "20:01"))
```

```
## [1] "time"
```

5 Importing web data formats

5.1 XML in R[3]

There are several XML-parsers already implemented in R packages specifically written for working with XML data. We thus do not have to understand the XML syntax in every detail in order to work with this data format in R. The already familiar package `xml2` (automatically loaded when loading `rvest`) provides the `read_xml()` function which we can use to read the exemplary XML-document.

```
# load packages
library(xml2)

# parse XML, represent XML document as R object
xml_doc <- read_xml("data/customers.xml")
xml_doc
```

```
## {xml_document}
## <customers>
## [1] <person>\n  <name>John Doe</name>\n  <orders>\n    <product> x </product>\n    <product> y </ ..
## [2] <person>\n  <name>Peter Pan</name>\n  <orders>\n    <product> a </product>\n    <product> x < ..
```

The same package also comes with various functions to access, extract, and manipulate data from a parsed XML document. In the following code example, we have a look at the most useful functions for our purposes (see the package's vignette for more details).

```
# navigate through the XML document (recall the tree-like nested structure similar to HTML)
# navigate downwards
# 'customers' is the root-node, persons are their'children'
persons <- xml_children(xml_doc)
# navigate sidewards
xml_siblings(persons)
```

```
## {xml_nodeset (2)}
## [1] <person>\n  <name>Peter Pan</name>\n  <orders>\n    <product> a </product>\n    <product> x < ..
## [2] <person>\n  <name>John Doe</name>\n  <orders>\n    <product> x </product>\n    <product> y </ ..
```

```
# navigate upwards
xml_parents(persons)
```

```
## {xml_nodeset (1)}
## [1] <customers>\n  <person>\n    <name>John Doe</name>\n    <orders>\n      <product> x </product ..
```

```
# find data via XPath
customer_names <- xml_find_all(xml_doc, xpath = ".//name")
# extract the data as text
xml_text(customer_names)
```

```
## [1] "John Doe"  "Peter Pan"
```

[3]This section is based on Matter (2018b).

5.2 JSON in R[4]

Again, we can rely on an R package (`jsonlite`) providing high-level functions to read, manipulate, and extract data when working with JSON-documents in R. An important difference to working with XML- and HTML-documents is that XPath is not compatible with JSON. However, as `jsonlite` represents parsed JSON as R objects of class `list` and/or `data-frame`, we can work with the parsed document as with any other R-object of the same class. The following example illustrates this point.

```
# load packages
library(jsonlite)

# parse the JSON-document shown in the example above
json_doc <- fromJSON("data/person.json")

# look at the structure of the document
str(json_doc)

# navigate the nested lists, extract data
# extract the address part
json_doc$address
# extract the gender (type)
json_doc$gender$type
```

```
## List of 6
##  $ firstName  : chr "John"
##  $ lastName   : chr "Smith"
##  $ age        : int 25
##  $ address    :List of 4
##   ..$ streetAddress: chr "21 2nd Street"
##   ..$ city         : chr "New York"
##   ..$ state        : chr "NY"
##   ..$ postalCode   : chr "10021"
##  $ phoneNumber:'data.frame': 2 obs. of  2 variables:
##   ..$ type  : chr [1:2] "home" "fax"
##   ..$ number: chr [1:2] "212 555-1234" "646 555-4567"
##  $ gender     :List of 1
##   ..$ type: chr "male"
```

```
## $streetAddress
## [1] "21 2nd Street"
##
## $city
## [1] "New York"
##
## $state
## [1] "NY"
##
## $postalCode
## [1] "10021"
```

```
## [1] "male"
```

[4] This section is based on Matter (2018b).

5.3 Tutorial (advanced): Importing data from a HTML table (on a website)

In the lecture on Big Data from the Web we have discussed the *Hypertext Markup Language (HTML)* as code to define the structure/content of a website as well HTML-documents as semi-structured data sources. In the following tutorial, we revisit the basic steps involved to import data from a HTML table into R.

The aim of the tutorial is to generate a CSV file containing data on 'divided government' in US politics. We use the following Wikipedia page as a data source: https://en.wikipedia.org/wiki/Divided_government_in_the_United_States. The page contains a table indicating the president's party, and the majority party in the US House and the US Senate per Congress (2-year periods). The first few rows of the cleaned data is supposed to look like this:

```
##    year  president senate house
## 1: 1861    Lincoln      R     R
## 2: 1862    Lincoln      R     R
## 3: 1863    Lincoln      R     R
## 4: 1864    Lincoln      R     R
## 5: 1865 A. Johnson      R     R
## 6: 1866 A. Johnson      R     R
```

In a first step, we initiate fix variables for paths and load additional R packages needed to handle data stored in HTML documents.

```
# SETUP ------------------

# load packages
library(rvest)
library(data.table)

# fix vars
SOURCE_PATH <- "https://en.wikipedia.org/wiki/Divided_government_in_the_United_States"
OUTPUT_PATH <- "data/divided_gov.csv"
```

Now we write the part of the script that fetches the data from the Web. This part consists of three steps. First we fetch the entire website (HTML document) from Wikipedia with (`read_html()`). Second, we extract the part of the website containing the table with the data we want (via `html_node()`). Finally, we parse the HTML table and store its content in a data frame called `tab`. The last line of the code chunk below simply removes the last row of the data frame (you can see on the website that this row is not needed)

```
# FETCH/FORMAT DATA -------------------

# fetch from web
doc <- read_html(SOURCE_PATH)
tab <- html_table(doc,fill=TRUE)[[2]]
tab <- tab[-nrow(tab), ] # remove last row (not containing data)
```

Now we clean the data in order to get a data set more suitable for data analysis. Note that the original table contains information per congress (2-year periods). However, as the sample above shows, we aim for a panel at the year level. The following code iterates through the rows of the data frame, and generates for each row per congress several two rows (one for each year in the congress).[5]

```
# generate year-level data.frame

# prepare loop
all_years <- list() # the container
```

[5] See `?strsplit`, `?unlist`, and this introduction to regular expressions for the background of how this is done in the code example here.

```
n <- length(tab$Year) # number of cases to iterate through
length(all_years) <- n
# generate year-level observations. row by row.
for (i in 1:n){
      # select row
      row <- tab[i,]
      y <- row$Year
      #
      begin <- as.numeric(unlist(strsplit(x = y, split = "[\\-\\-]", perl = TRUE))[1])
      end <- as.numeric(unlist(strsplit(x = y, split =  "[\\-\\-]"))[2])
      tabnew <- data.frame(year=begin:(end-1), president=row$President, senate=row$Senate, house=row$Hou
      all_years[[i]] <- tabnew # store in container
}

# stack all rows together
allyears <- bind_rows(all_years)
```

In a last step, we inspect the collected data and write it to a CSV file.

```
# WRITE TO DISK ------------------------------------------------

# inspect
head(allyears)
```

```
##   year  president senate house
## 1 1861    Lincoln      R     R
## 2 1862    Lincoln      R     R
## 3 1863    Lincoln      R     R
## 4 1864    Lincoln      R     R
## 5 1865 A. Johnson      R     R
## 6 1866 A. Johnson      R     R
```

```
# write to csv
write_csv(allyears, path=OUTPUT_PATH)
```

References

Matter, Ulrich. 2018a. "A Brief Introduction to Programming with R." Lecture notes. St. Gallen: University of St. Gallen.

———. 2018b. "Introduction to Web Data Mining for Social Scientists." Lecture notes. St. Gallen: University of St. Gallen.

Wickham, Hadley, and Garrett Grolemund. 2017. Sebastopol, CA: O'Reilly. http://r4ds.had.co.nz/.