Data Handling: Import, Cleaning and Visualisation

Lecture 3: Data Storage and Data Structures

Prof. Dr. Ulrich Matter

04/10/2018

A core part of the practical skills taught in this course has to do with writing, executing, and storing *computer code* (i.e., instructions to a computer, in a language that it understands) as well as storing and reading *data*. The way data is typically stored on a computer follows quite naturally from the outlined principles of how computers process data (both technically speaking and in terms of practical applications). Based on a given standard of how `0`s and `1`s are translated into the more meaningful symbols we see on our keyboard, we simply write data (or computer code) to a *text file* and save this file on the hard-disk drive of our computer. Again, what is stored on disk is in the end only consisting of `0`s and `1`s. But, given the standards outlined above, these `0`s and `1`s properly map to characters that we can understand when reading the file again from disk and look at its content on our computer screen.

1 Unstructured data in text files

In the simplest case, what we want to store is literally a text. For example, we store the following phrase

```
Hello World!
```

in a text file named `helloworld.txt`. The ending `.txt` indicates to the operating system of our computer what kind of data is stored in this file. When clicking on the file's symbol, the operating system will open the file (read it into memory) with the default program assigned to open `.txt`-files (in this course, usually atom or RStudio). If the text editor displays `Hello World!` as the content of this file, the program uses the correct character encoding to convert the raw `0`s and `1`s yet again to symbols that we can read more easily. Similarly we can show the content of the file in the command-line terminal (here OSX or Linux):

```
cat helloworld.txt; echo
```

```
## Hello World!
```

However, we can also use a command-line program (here, `xxd`) to display the content of `helloworld.txt` without actually 'translating' it into ASCII characters.[1] That is, we can directly look at how the content looks like as `0`s and `1`s:

```
xxd -b helloworld.txt
```

```
## 00000000: 01001000 01100101 01101100 01101100 01101111 00100000  Hello
## 00000006: 01010111 01101111 01110010 01101100 01100100 00100001  World!
```

Similarly we can display the content in hexadecimal values:

```
xxd  data/helloworld.txt
```

```
## 00000000: 4865 6c6c 6f20 576f 726c 6421            Hello World!
```

Now consider another text file `hastamanana.txt` which was written and stored with another character encoding. When looking at the content, assuming the now common UTF-8 encoding, the content seems a little odd:

```
cat hastamanana.txt; echo
```

[1] To be precise, this program shows both the raw binary content as well as its ASCII representation.

```
## Hasta Ma?ana!
```

We see that it is a short phrase written in Spanish. But oddly, it contains the character `?` in the middle of a word. The occurrence of special characters in unusual places of text files is an indication of using the wrong character encoding to display the text. Let's check what the file's encoding is.

```
file -b data/hastamanana.txt
```

```
## ISO-8859 text
```

This tells us that the file is encoded with ISO-8859 ('Latin1'), a character set for several Latin languages (including Spanish). Knowing this, we can check how the content looks like when we change the encoding to UTF-8 (which then properly displays the content on the screen)[2]:

```
iconv -f iso-8859-1 -t utf-8 hastamanana.txt | cat
```

```
## Hasta Mañana!
```

When working with data, we thus have to make sure that we use the proper standards to translate the binary coded values into a character/text-representation that is easier to understand and work with. In recent years, much more general (or even 'universal') standards have been developed and adopted around the world, particularly UTF-8. Thus, if we deal with recently generated data sets, we usually can expect that they are encoded in UTF-8, independent of the data's country of origin. However, when working on a research project where slightly older data sets from different sources are involved, encoding issues still occur quite frequently. It is thus crucial to understand early on what the origin of the problem is, when the data seem to display weird characters. Encoding issues are among the most basic problems that can occur when importing data.

So far, we have only looked at very simple examples of data stored in text files, essentially only containing short phrases of text. Nevertheless, most common formats of storing and transferring data (CSV, XML, JSON, etc.) build on exactly the same principle: characters in a text-file. The difference between such formats and the simple examples above is that their content is structured in a very specific way. That is, on top of the lower-level conversion of `0`s and `1`s into characters/text we add another standard, a data format, giving the data more *structure*, making it even simpler to interpret and work with the data on a computer. Once we understand how data is represented at the low level (the topic of the sections above), it is much easier to understand and distinguish different forms of storing structured data.

2 Structured data formats

As nicely pointed out by Murrell (2009), most commonly used software today is designed in a very 'user-friendly' way, leading to situations in which we as users are 'told' by the computer what to do (rather than the other way around). A prominent symptom of this phenomenon is that specific software is automatically assigned to open files of specific formats, so we don't have to bother about what the file's format actually is but simply have to click on the icon representing the file.

However, if we actually want to engage seriously with a world driven by data we have to go beyond the habit of simply clicking on (data-)files and let the computer choose what to do with it. Since most common formats to store data are in essence text files (in research contexts usually in ASCII or UTF-8 encoding), a good way to start engaging with a data set is to look at the raw text file containing the data in order to get an idea of how the data is structured.

For example, let's have a look at the file `ch_gdp.csv`. When opening the file in a text editor we see the following:

```
year,gdp_chfb
1980,184
```

[2]Note that changing the encoding in this way only works well if we actually know what the original encoding of the file is (i.e., the encoding used when the file was initially created)

```
1985,244
1990,331
1995,374
2000,422
2005,464
```

At first sight, the file contains a collection of numbers and characters. Having a closer look, certain structural features become apparent. The content is distributed over several rows, with each row containing a comma character (`,`). Moreover, the first row seems systematically different from the following rows: it is the only row containing alphabetical characters, all the other rows contain numbers. Rather intuitively, we recognize that the specific way in which the data in this file is structured might in fact represent a table with two columns: one with the variable `year` describing the year of each observation (row), the other with the variable `gdp_chfb` with numerical values describing another characteristic of each observation/row (we can guess that this variable is somehow related to GDP and CHF/Swiss franks).

Now, how can we work with these data? Recall that this is simply a text file. All the information stored in it, is displayed above. How could we explain the computer that this is actually a table with two columns, containing two variables describing six observations? We would have to come up with a sequence of instructions/rules (i.e., an algorithm) of how to distinguish columns and rows, when all that a computer can do is sequentially read one character after the other (more precisely one `0/1` after the other, representing characters).

This algorithm could be something along the lines of:

1. Start with an empty table consisting of one cell (1 row/column).
2. While the end of the input file is not yet reached, do the following: Read characters from the input file, and add them one-by-one to the current cell. If you encounter the character '`,`', ignore it, create a new field, and jump to the new field. If you encounter the end of the line, create and jump to the next row.

Consider the usefulness of this algorithm. It would certainly work quite well for the particular file we are looking at. But what if another file we want to read data from does not contain any '`,`' but only '`;`'? We would have to tweak the algorithm accordingly. Hence, again, it is extremely useful if we can agree on certain standards of how data structured as a table/matrix is stored in a text file.

2.1 CSVs and fixed-width format

Incidentally, the example we are looking at here is in line with such a standard, i.e., Comma-Separated Values (CSV, therefore `.csv`). In technical terms, this simple algorithm is a CSV parser. That is, a software component which takes a text file with CSV standard structure as input and builds a data structure in the form of a table. If the input file does not follow the agreed on CSV standard, the parser will likely fail to properly parse the input.

CSV files are very common to transfer and store data in a table-like format. They essentially are based on two rules defining the structure of the data: commas delimit values/fields in a row and the end of a line indicates the end of a row.[3]

The comma is clearly visible when looking at the raw text content of `ch_gdp.csv`. However, how does the computer know that a line is ending? By default most programs to edit with text do not show an explicit symbol for line endings but instead display the text on the next line. Under the hood, these line endings are, however, non-printable characters. We can see this when investigating `ch_gdp.csv` in the command-line terminal (via `xxd`):

```
xxd ../../data/ch_gdp.csv
```

```
## 00000000: efbb bf79 6561 722c 6764 705f 6368 6662  ...year,gdp_chfb
```

[3] In addition, if a field contains itself a comma (the comma is part of the data), the field needs to be surrounded by double-quotes (`"`). If a field contains double-quotes it also has to be surrounded by double-quotes, and the double quotes in the field must be preceded by an additional double-quote.

```
## 00000010: 0d31 3938 302c 3138 340d 3139 3835 2c32  .1980,184.1985,2
## 00000020: 3434 0d31 3939 302c 3333 310d 3139 3935  44.1990,331.1995
## 00000030: 2c33 3734 0d32 3030 302c 3432 320d 3230  ,374.2000,422.20
## 00000040: 3035 2c34 3634                           05,464
```

When comparing the hexadecimal values with the characters they represent on the right, we recognize that right before every year, a full stop (`.`) is printed to the output, as well as that this `.` corresponds to `0d` in the hexadecimal code. `0d` is indeed the sequence of `0`s and `1` indicating the end of a line. Because this character does not actually correspond to a symbol printed on the screen, it is replaced by `.` in the printed output of the text.[4]

While CSV files have become a very common way to store 'flat'/table-like data in plain text files, several similar formats can be encountered in practice. Most commonly they either use a different delimiter (for example, tabs/white space) to separate fields in a row, or fields are defined to consist of a fixed number of characters (so-called fixed-width formats). In addition, various more complex standards to store and transfer digital data are widely used to store data. Particularly in the context of web data (data stored and transferred online), files storing data in these formats (e.g., XML, JSON, YAML, etc.) are in the end just plain text files. However, they contain a larger set of special characters/delimiters (or combinations of these) to indicated the structure of the data stored in them.

3 Units of information/data storage

With the question of how to store digital data on computers, we obviously also raise the question of storage capacity. How much can be stored on a computer and how do we quantify the size of a data set?

Because every type of digital data can in the end only be stored as `0`s and `1`s, it makes perfectly sense to define the smallest unit of information in computing as consisting of either a `0` or a `1`. We call this basic unit a *bit* (from *bi*nary dig*it*; abbrev. 'b'). Recall that the decimal number `139` corresponds to the binary number `10001011`. In order to store this number on a hard disk, we require a capacity of 8 bits, or one *byte* (1 byte = 8 bits; abbrev. 'B'). Historically, one byte encoded a single character of text (i.e., in the ASCII character encoding system).

Bigger units for storage capacity usually build on bytes:

- 1 kilobyte (KB) = 1000^1 bytes
- 1 megabyte (MB) = 1000^2 bytes
- 1 gigabyte (GB) = 1000^3 bytes

Similarly, the data-transfer rate, that is, the speed with which data can be transferred in the network is commonly measured in units of bits per second (bit/s):

- 1 kilobit per second (kbit/s) = 1000^1 bit/s
- 1 megabit per second (mbit/s) = 1000^2 bit/s
- 1 gigabit per second (gbit/s) = 1000^3 bit/s

4 How information is stored in computer memory

So far, we have focused on how data is stored on the hard disk. That is, we discovered how `0`s and `1`s correspond to characters (following specific standards: character encodings), how a sequence of characters in a text file is a useful way to store data on a hard disk, and how specific standards are used to structure the data (with the help of special characters) in a meaningful way. When thinking about how to store data on a hard disk, we usually are concerned with how much space (in bytes) the data set needs, how well it is

[4] The same applies to other sequences of `0`s and `1`s that do not correspond to a printable character.

transferable/understandable, and how well we can retrieve information from it. All of these aspects go into the decision of what structure/format to choose etc.

However, none of this is actually taking into consideration how data structures relate to how we actively work with the data. For example, when we want to sum up the `gdp_chfb` column in `ch_gdp.csv` (the table example above), do we actually work within the CSV data structure? The answer is usually no.

Recall from the basics of data processing that data stored on the hard disk is loaded into RAM to work with (analysis, manipulation, cleaning, etc.). Now, the question is, how is the data structured in RAM? That is, what data structures/formats are used to actively work with the data?

We distinguish two basic characteristics:

- Data **types**: integers; real numbers ('numeric values', floating point numbers); text ('string', 'character values').
- Basic **data structures** in RAM:
  - *Vectors*
  - *Arrays/Matrices*
  - *Lists*
  - *Data frames* (very `R`-specific)

Depending on what data structure/format the data is stored in on the hard disk, the data will be more or less usefully represented in one of the above structures in RAM. For example, in the `R` language it is quite common to represent data stored in CSVs on disk as data frames in RAM. Similarly, it is quite common to represent a more complex format on disk, such as XML, as a nested list in RAM.

Importing data from the hard disk (or another mass storage device) into RAM in order to work with it in `R`, essentially means reading the sequence of characters (in the end, of course, `0`s and `1`s) and mapping them, given the structure they are stored in, into one of these structures for representation in RAM.[5].

References

Murrell, Paul. 2009. *Introduction to Data Technologies*. London, UK: CRC Press.

[5]In the lecture on data gathering and data import, we will cover this crucial step in more detal