Data Handling: Import, Cleaning and Visualisation

Lecture 6:
Non rectangular data

Prof. Dr. Ulrich Matter, updated by Dr. Aurélien Sallin

Complex Data Structures

This lecture is about complex data structures, or non-rectangular data.

A rectangular data set

```
father mother  name     age  gender
               John      33  male
               Julia     32  female
John   Julia   Jack       6  male
John   Julia   Jill       4  female
John   Julia   John jnr   2  male
               David     45  male
               Debbie    42  female
David  Debbie  Donald    16  male
David  Debbie  Dianne    12  female
```

- Not entirely clear which observations “belong together”.
- Might be better represented in another format that allows for *hierarchical structures*.
- “Hierarchy” in this example/context would mean, there are records of families (higher level), each family consists of several persons, each of these persons has a number of characteristics (such as age, name, etc.).

Limitations of rectangular data

- Only *two dimensions*.
  - Observations (rows)
  - Characteristics/variables (columns)
- Hard to represent hierarchical structures.
  - Might introduce redundancies.
  - Machine-readability suffers (standard parsers won’t recognize it).

Alternative formats

- JavaScript Object Notation (JSON) (https://en.wikipedia.org/wiki/JSON)
- Extensible Markup Language (XML) (https://en.wikipedia.org/wiki/XML)
- Origin and most common domain of application: The Web!
  - Need to *transfer* complex data (between machines).
  - Need to *embed* complex data (in human friendly layout).

Deciphering XML

Revisiting air quality data

Recall the air quality data you have worked with as part of the exercises. We can store this data in the form of a CSV file as illustrated below. Commas separate columns, new lines separate rows, and the first row contains the column/variable names.

```
unique_id,indicator_id,name,measure,measure_info,geo_type_name,
  geo_join_id,geo_place_name,time_period,start_date,data_value
216498,386,Ozone (O3),Mean,ppb,CD,
  313,Coney Island (CD13),Summer 2013,2013-06-01T00:00:00,34.64
216499,386,Ozone (O3),Mean,ppb,CD,
  313,Coney Island (CD13),Summer 2014,2014-06-01T00:00:00,33.22
219969,386,Ozone (O3),Mean,ppb,Borough,
  1,Bronx,Summer 2013,2013-06-01T00:00:00,31.25
```

The same data can also be stored as an XML file. The first few lines of this file could look like this:

```
<row>
<unique_id>216498</unique_id>
<indicator_id>386</indicator_id>
<name>Ozone (O3)</name>
<measure>Mean</measure>
<measure_info>ppb</measure_info>
<geo_type_name>CD</geo_type_name>
<geo_join_id>313</geo_join_id
><geo_place_name>Coney Island (CD13)</geo_place_name>
<time_period>Summer 2013</time_period>
<start_date>2013-06-01T00:00:00</start_date>
<data_value>34.64</data_value>
</row>

<unique_id>216499</unique_id>
<indicator_id>386</indicator_id>
...

<\row>
```

- A predefined set of special characters (here primarily `<`, `>`, and `/`) give the data structure.
- So-called *XML-tags* are used to define variable names and encapsulate data values: `<variablename>value</variablename>`.
- Tags can be nested, which allows for the definition of hierarchical structures.

For example, the entire air quality dataset content we know from the csv example above is nested between the '*row*``'-tags:

```
  <records>
...
  </records>
```

- In this example, *row* is the "root-element" of the XML-document.
- * row``* contains several * row * elements, which in turn contain several tags/variables describing a unique record (such as time_peri

It becomes obvious that xml files follow a tree structure:

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

XML syntax: attribute-based or tag-based

There are two principal ways to link variable names and data values in XML:

1. Values are stored between tags: `<variablename>value</variablename>`. In the example below: `<filename>ISCCPMonthly_avg.nc</filename>`.
2. Values are stored as XML-attributes (key-value pairs) within tags: `<observation variablename="value">`. In the example below: `<case date="16-JAN-1994" temperature="9.200012" />`

```
   <variable>Monthly Surface Clear-sky Temperature (ISCCP) (Celsius)</variable>
   <filename>ISCCPMonthly_avg.nc</filename>
   <filepath>/usr/local/fer_data/data/</filepath>
   <badflag>-1.E+34</badflag>
   <subset>48 points (TIME)</subset>
   <longitude>123.8W(-123.8)</longitude>
   <latitude>48.8S</latitude>
   <case date="16-JAN-1994" temperature="9.200012" />
   <case date="16-FEB-1994" temperature="10.70001" />
   <case date="16-MAR-1994" temperature="7.5" />
   <case date="16-APR-1994" temperature="8.100006" />
```

The same information can be stored either way, as the following example shows:

Attributes-based:

```
   <case date="16-JAN-1994" temperature="9.200012" />
   <case date="16-FEB-1994" temperature="10.70001" />
   <case date="16-MAR-1994" temperature="7.5" />
   <case date="16-APR-1994" temperature="8.100006" />
```

Tag-based:

```
  <cases>
    <case>
      <date>16-JAN-1994<date/>
      <temperature>9.200012<temperature/>
    <case/>
    <case>
      <date>16-FEB-1994<date/>
      <temperature>10.70001<temperature/>
    <case/>
    <case>
      <date>16-MAR-1994<date/>
      <temperature>7.5<temperature/>
    <case/>
    <case>
      <date>16-APR-1994<date/>
      <temperature>8.100006<temperature/>
    <case/>
  <cases/>
```

Insights: CSV vs. XML

Note the key differences of storing data in XML format in contrast to a flat, table-like format such as CSV:

- Represent much more *complex (multi-dimensional)* data in XML-files than what is possible in CSVs.
  - Arbitrarily complex nesting structure.
  - Flexibility to label tags.
- Self-explanatory syntax: *machine-readable and human-readable*.
  - Parsers/computers can more easily handle complex data structures.
  - Humans can intuitively understand what the data is all about just by looking at the raw XML file.

Potential drawback of XML: *inefficient* storage:

- Tags are part of the syntax, thus, part of the actual file.
  - Tags (variable labels) are *repeated* again and again!
  - CSV: variable labels are mentioned once.
  - Potential solution: data compression (e.g., zip).
- If the data is actually two dimensional, a CSV is more practical.

JSON vs XML syntax

- Key difference to XML: no tags, but *attribute-value pairs*.
- A substitute for XML (often encountered in similar usage domains).

The following two data samples show the same information once stored in an XML file and once in a JSON file:

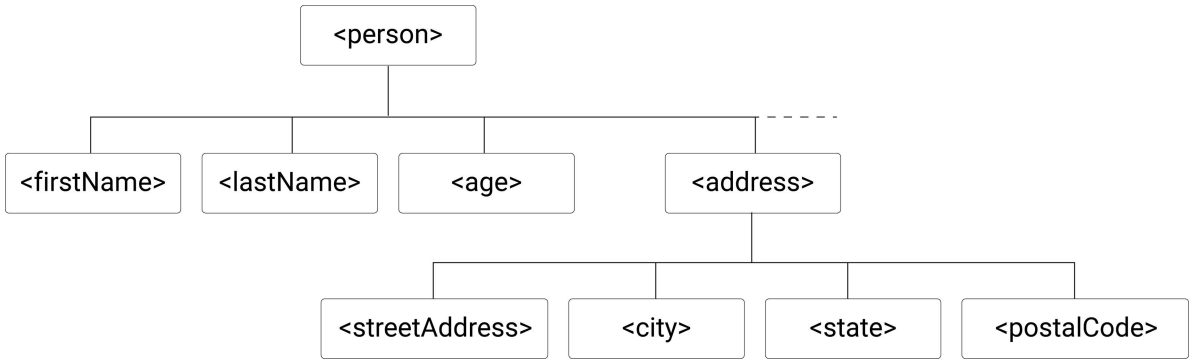
XML:

```
<person>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <age>25</age>
  <address>
    <streetAddress>21 2nd Street</streetAddress>
    <city>New York</city>
    <state>NY</state>
    <postalCode>10021</postalCode>
  </address>
  <phoneNumber>
    <type>home</type>
    <number>212 555-1234</number>
  </phoneNumber>
  <phoneNumber>
    <type>fax</type>
    <number>646 555-4567</number>
  </phoneNumber>
  <gender>
    <type>male</type>
  </gender>
</person>
```

JSON:

```
{"firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ],
  "gender": {
    "type": "male"
  }
}
```

Data structured according to either XML or JSON syntax can be thought of as following a tree-like structure:



Parsing XML and JSON in R

Parsing XML in R

The following examples are based on the example code shown above (the two text-files `persons.json` and `persons.xml`)

```
# load packages
library(xml2)

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

```
## {xml_document}
## <person>
## [1] <firstName>John</firstName>
## [2] <lastName>Smith</lastName>
## [3] <age>25</age>
## [4] <address>\n  <streetAddress>21 2nd Street</streetAddress>\n  <city>New York</city>\n  <state> ...
## [5] <phoneNumber>\n  <type>home</type>\n  <number>212 555-1234</number>\n</phoneNumber>
## [6] <phoneNumber>\n  <type>fax</type>\n  <number>646 555-4567</number>\n</phoneNumber>
## [7] <gender>\n  <type>male</type>\n</gender>
```

'customers' is the root-node, 'persons' are its children:

```
# navigate downwards
persons <- xml_children(xml_doc)
persons
```

```
## {xml_nodeset (7)}
## [1] <firstName>John</firstName>
## [2] <lastName>Smith</lastName>
## [3] <age>25</age>
## [4] <address>\n  <streetAddress>21 2nd Street</streetAddress>\n  <city>New York</city>\n  <state> ...
## [5] <phoneNumber>\n  <type>home</type>\n  <number>212 555-1234</number>\n</phoneNumber>
## [6] <phoneNumber>\n  <type>fax</type>\n  <number>646 555-4567</number>\n</phoneNumber>
## [7] <gender>\n  <type>male</type>\n</gender>
```

You can thus navigate downwards from the root-node to specific leaf-nodes via `xml_children()`. In addition you can navigate horizontally or upwards via `xml_siblings()` and `xml_parents()`, respectively.

```
# navigate sidewards
persons[1]
```

```
## {xml_nodeset (1)}
## [1] <firstName>John</firstName>
```

```
xml_siblings(persons[[1]])
```

```
## {xml_nodeset (6)}
## [1] <lastName>Smith</lastName>
## [2] <age>25</age>
## [3] <address>\n  <streetAddress>21 2nd Street</streetAddress>\n  <city>New York</city>\n  <state> ...
## [4] <phoneNumber>\n  <type>home</type>\n  <number>212 555-1234</number>\n</phoneNumber>
## [5] <phoneNumber>\n  <type>fax</type>\n  <number>646 555-4567</number>\n</phoneNumber>
## [6] <gender>\n  <type>male</type>\n</gender>
```

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

```
## {xml_nodeset (1)}
## [1] <person>\n  <firstName>John</firstName>\n  <lastName>Smith</lastName>\n  <age>25</age>\n  <ad ...
```

Advanced topic: extract specific parts of the data via XPATH. `xml_find_all()` allows you to find any data values with specific characteristics as defined by the `xpath` -argument.[1]

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

```
## character(0)
```

Parsing JSON in R

Similar to the case of XML, there are several R-packages providing functions to import and work with JSON. Here, we work with the easy-to-use `jsonlite` -package.

```
# 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)
```

```
## 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"
```

The nesting structure is represented as a *nested list*:

```
# navigate the nested lists, extract data
# extract the address part
json_doc$address
```

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

```
# extract the gender (type)
json_doc$gender$type
```

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

HTML: Computer Code Meets Data

HTML: Code to build webpages

HyperText Markup Language (HTML) (https://en.wikipedia.org/wiki/HTML) is designed to be read and rendered by a web browser. Yet, web pages (HTML-documents) also contain tables, raw text, and images and thus they are also a file format to store data.

- Web designer's perspective: "HTML is a tool to design the layout of a webpage (and the resulting HTML document is *code*)."
- From a data scientist's perspective: "HTML gives the *data* contained in a webpage (the actual content) a certain degree of structure which can be exploited to systematically extract the data from the webpage."
- HTML documents/webpages consist of *'semi-structured data'*:
  - A webpage can contain a HTML-table (*structured data*)…
  - …but likely also contains just raw text (*unstructured data*).

The following short HTML-file constitues a very simple web page:

```
<!DOCTYPE html>

<html>
    <head>
        <title>hello, world</title>
    </head>
    <body>
        <h2> hello, world </h2>
    </body>
</html>
```

Characteristics of HTML

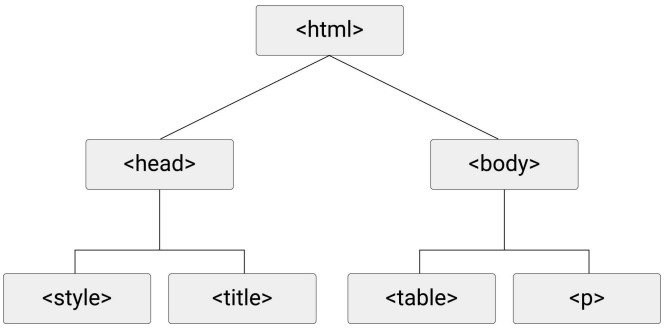
1. *Annotate/'mark up'* data/text (with tags)

- Defines *structure* and hierarchy
- Defines content (pictures, media)

2. *Nesting* principle
   - `head` and `body` are nested within the `html` document
   - Within the `head`, we define the `title`, etc.
3. Expresses what is what in a document.
   - Doesn't explicitly 'tell' the computer what to do
   - HTML is a markup language, not a programming language.

HTML document as a 'tree'

- 'Root': `<html>..</html>`
- 'Children' of the root node: `<head>...</head>`, `<body>...</body>`
- 'Siblings' of each other: `<head>...</head>`, `<body>...</body>`



HTML (DOM) tree diagram.

Two ways to read a webpage into R

In this example, we look at Wikipedia's Economy of Switzerland page (https://en.wikipedia.org/wiki/Economy_of_Switzerland).

Year	GDP (billions of CHF)	US Dollar Exchange
1980	184	1.67 Francs
1985	244	2.43 Francs
1990	331	1.38 Francs
1995	374	1.18 Francs
2000	422	1.68 Francs
2005	464	1.24 Francs
2006	491	1.25 Francs
2007	521	1.20 Francs
2008	547	1.08 Francs
2009	535	1.09 Francs
2010	546	1.04 Francs
2011	659	0.89 Francs
2012	632	0.94 Francs
2013	635	0.93 Francs
2014	644	0.92 Francs
2015	646	0.96 Francs
2016	659	0.98 Francs
2017	668	1.01 Francs
2018	694	1.00 Francs

Source: https://en.wikipedia.org/wiki/Economy_of_Switzerland (https://en.wikipedia.org/wiki/Economy_of_Switzerland).

Read the HTML line-by-line

```
swiss_econ <- readLines("https://en.wikipedia.org/wiki/Economy_of_Switzerland")
```

```
## Warning in readLines("https://en.wikipedia.org/wiki/Economy_of_Switzerland"): incomplete final line
## found on 'https://en.wikipedia.org/wiki/Economy_of_Switzerland'
```

Look at the first few imported lines:

```
head(swiss_econ)
```

```
## [1] "<!DOCTYPE html>"
## [2] "<html class=\"client-nojs vector-feature-language-in-header-enabled vector-feature-language-in-main-page-header-disa
bled vector-feature-sticky-header-disabled vector-feature-page-tools-pinned-disabled vector-feature-toc-pinned-clientpref-1
vector-feature-main-menu-pinned-disabled vector-feature-limited-width-clientpref-1 vector-feature-limited-width-content-enab
led vector-feature-zebra-design-disabled vector-feature-custom-font-size-clientpref-0 vector-feature-client-preferences-disa
bled vector-feature-typography-survey-disabled vector-toc-available\" lang=\"en\" dir=\"ltr\">"
## [3] "<head>"
## [4] "<meta charset=\"UTF-8\">"
## [5] "<title>Economy of Switzerland - Wikipedia</title>"
## [6] "<script>(function(){var className=\"client-js vector-feature-language-in-header-enabled vector-feature-language-in-m
ain-page-header-disabled vector-feature-sticky-header-disabled vector-feature-page-tools-pinned-disabled vector-feature-toc-
pinned-clientpref-1 vector-feature-main-menu-pinned-disabled vector-feature-limited-width-clientpref-1 vector-feature-limite
d-width-content-enabled vector-feature-zebra-design-disabled vector-feature-custom-font-size-clientpref-0 vector-feature-cli
ent-preferences-disabled vector-feature-typography-survey-disabled vector-toc-available\";var cookie=document.cookie.match(/
(?:^|; )enwikimwclientpreferences=([^;]+)/);if(cookie){cookie[1].split('%2C').forEach(function(pref){className=className.rep
lace(new RegExp('(^| )'+pref.replace(/-clientpref-\\w+$|[^\\w-]+/g,'')+'-clientpref-\\\\w+( |$)'),'$1'+pref+'$2');});}docume
nt.documentElement.className=className;}());RLCONF={\"wgBreakFrames\":false,\"wgSeparatorTransformTable\":[\"\",\"\"],\"wgDi
gitTransformTable\":[\"\",\"\"],"
```

Select specific lines (select specific parts of the data):

```
swiss_econ[231]
```

```
## [1] "\t\t\t\t"
```

Parse the HTML!

- Navigate the document like an XML document!
  - Same logic (but tags are pre-defined)…
  - Traverse the tree.
- Access specific parts of the contained data directly.
- *Make use of the structure!*

Parsing a Webpage with R

```
# install package if not yet installed
# install.packages("rvest")

# load the package
library(rvest)
```

```
# parse the webpage, show the content
swiss_econ_parsed <- read_html("https://en.wikipedia.org/wiki/Economy_of_Switzerland")
swiss_econ_parsed
```

```
## {html_document}
## <html class="client-nojs vector-feature-language-in-header-enabled vector-feature-language-in-main-page-header-disabled v
ector-feature-sticky-header-disabled vector-feature-page-tools-pinned-disabled vector-feature-toc-pinned-clientpref-1 vector
-feature-main-menu-pinned-disabled vector-feature-limited-width-clientpref-1 vector-feature-limited-width-content-enabled ve
ctor-feature-zebra-design-disabled vector-feature-custom-font-size-clientpref-0 vector-feature-client-preferences-disabled v
ector-feature-typography-survey-disabled vector-toc-available" lang="en" dir="ltr">
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">\n<meta charset="U ...
## [2] <body class="skin-vector skin-vector-search-vue mediawiki ltr sitedir-ltr mw-hide-empty-elt n ...
```

Parsing a Webpage with R

Now we can easily separate the data/text from the html code. For example, we can extract the HTML table containing the data we are interested in as a `data.frames`.

```
tab_node <- html_node(swiss_econ_parsed,
                      xpath = "//*[@id='mw-content-text']/div/table[2]")
tab <- html_table(tab_node)
tab
```

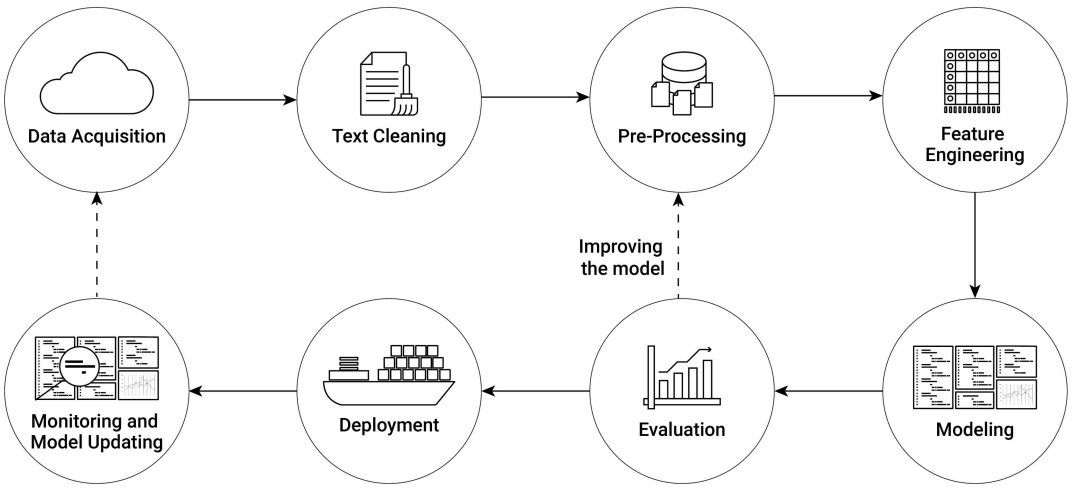
```
## # A tibble: 19 × 3
##     Year `GDP (billions of CHF)` `US dollar exchange`
##    <int>                   <int> <chr>
##  1  1980                     184 1.67 Francs
##  2  1985                     244 2.43 Francs
##  3  1990                     331 1.38 Francs
##  4  1995                     374 1.18 Francs
##  5  2000                     422 1.68 Francs
##  6  2005                     464 1.24 Francs
##  7  2006                     491 1.25 Francs
##  8  2007                     521 1.20 Francs
##  9  2008                     547 1.08 Francs
## 10  2009                     535 1.09 Francs
## 11  2010                     546 1.04 Francs
## 12  2011                     659 0.89 Francs
## 13  2012                     632 0.94 Francs
## 14  2013                     635 0.93 Francs
## 15  2014                     644 0.92 Francs
## 16  2015                     646 0.96 Francs
## 17  2016                     659 0.98 Francs
## 18  2017                     668 1.01 Francs
## 19  2018                     694 1.00 Francs
```

Text as Data

Handling text data for analysis

First few steps in a text analysis/natural language processing (NLP) pipeline:

1. Data acquisition: text data are collected from disparate sources. Examples are webpage scrapping, numerization of old administrative records, collection of tweets through an API, etc.
2. Text cleaning.
3. Text preprocessing require the analyst to prepare the text in such a way that text information can be read by a statistical software. At this step, text information is “transformed” into a matrix.
4. Feature engineering is the process of using domain knowledge to extract features (characteristics, properties, attributes) from raw data.



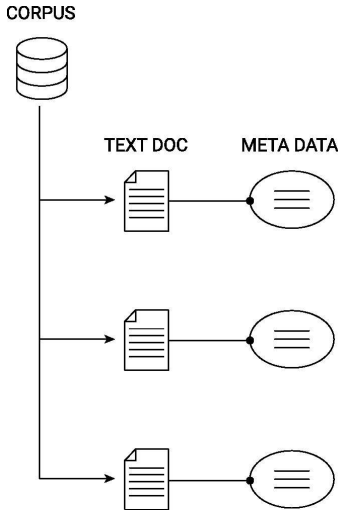
Working with text data in R: Quanteda

The package `quanteda` is the most complete and go-to package for text analysis in R. In order to run `quanteda`, several packages need to be installed. You can use the following command to make sure that missing packages are installed.

```
pacman::p_load(
    tidytext,
    quanteda,
    readtext,
    stringr,
    quanteda.textstats,
    quanteda.textplots
)
```

From raw text to corpus: step (1)

The base, raw material, of quantitative text analysis is a **corpus.** A corpus is, in NLP, *a collection of authentic text organized into datasets*.



In the specific case of `quanteda` , a corpus is a **a data frame consisting of a character vector for documents, and additional vectors for document-level variables**. In other words, a corpus is a data frame that contains, in each row, a text document, and additional columns with the corresponding metadata about the text.

In the examples below, we will use the `inauguration` corpus from `quanteda` , which is a standard corpus used in introductory text analysis. It contains the inauguration discourses of the five first US presidents. This text data can be loaded from the `readtext` package. The metadata of this corpus is the year of the inauguration and the name of the president taking office.

```
# set path
path_data <- system.file("extdata/", package = "readtext")

# import csv file
dat_inaug <- read.csv(paste0(path_data, "/csv/inaugCorpus.csv"))
names(dat_inaug)
```

```
## [1] "texts"     "Year"      "President" "FirstName"
```

```
# Create a corpus
corp <- corpus(dat_inaug, text_field = "texts")
print(corp)
```

```
## Corpus consisting of 5 documents and 3 docvars.
## text1 :
## "Fellow-Citizens of the Senate and of the House of Representa..."
##
## text2 :
## "Fellow citizens, I am again called upon by the voice of my c..."
##
## text3 :
## "When it was first perceived, in early times, that no middle ..."
##
## text4 :
## "Friends and Fellow Citizens: Called upon to undertake the du..."
##
## text5 :
## "Proceeding, fellow citizens, to that qualification which the..."
```

```
# Look at the metadata in the corpus using `docvars`
docvars(corp)
```

```
##   Year  President FirstName
## 1 1789 Washington    George
## 2 1793 Washington    George
## 3 1797      Adams      John
## 4 1801  Jefferson    Thomas
## 5 1805  Jefferson    Thomas
```

```
# In quanteda, the metadata in a corpus can be handled like data frames.
docvars(corp, field = "Century") <- floor(docvars(corp, field = "Year") / 100) + 1
```

From corpus to tokens: steps (2) and (3)

Once we have a corpus, we want to extract the substance of the text. This means, in `quanteda` language, that we want to extract **tokens**, i.e. to isolate the elements that constitute a corpus in order to quantify them. Basically, tokens are expressions that form the building blocks of the text. Tokens can be single words or phrases (several subsequent words, so-called *N-grams*).

```
toks <- tokens(corp)
head(toks[[1]], 20)
```

```
##  [1] "Fellow-Citizens" "of"              "the"             "Senate"          "and"
##  [6] "of"              "the"             "House"           "of"              "Representatives"
## [11] ":"               "Among"           "the"             "vicissitudes"    "incident"
## [16] "to"              "life"            "no"              "event"           "could"
```

```
# Remove punctuation
toks <- tokens(corp, remove_punct = TRUE)
head(toks[[1]], 20)
```

```
##  [1] "Fellow-Citizens" "of"              "the"             "Senate"          "and"
##  [6] "of"              "the"             "House"           "of"              "Representatives"
## [11] "Among"           "the"             "vicissitudes"    "incident"        "to"
## [16] "life"            "no"              "event"           "could"           "have"
```

```
# Remove stopwords
stopwords("en")
```

```
##   [1] "i"          "me"         "my"         "myself"     "we"         "our"        "ours"
##   [8] "ourselves"  "you"        "your"       "yours"      "yourself"   "yourselves" "he"
##  [15] "him"        "his"        "himself"    "she"        "her"        "hers"       "herself"
##  [22] "it"         "its"        "itself"     "they"       "them"       "their"      "theirs"
##  [29] "themselves" "what"       "which"      "who"        "whom"       "this"       "that"
##  [36] "these"      "those"      "am"         "is"         "are"        "was"        "were"
##  [43] "be"         "been"       "being"      "have"       "has"        "had"        "having"
##  [50] "do"         "does"       "did"        "doing"      "would"      "should"     "could"
##  [57] "ought"      "i'm"        "you're"     "he's"       "she's"      "it's"       "we're"
##  [64] "they're"    "i've"       "you've"     "we've"      "they've"    "i'd"        "you'd"
##  [71] "he'd"       "she'd"      "we'd"       "they'd"     "i'll"       "you'll"     "he'll"
##  [78] "she'll"     "we'll"      "they'll"    "isn't"      "aren't"     "wasn't"     "weren't"
##  [85] "hasn't"     "haven't"    "hadn't"     "doesn't"    "don't"      "didn't"     "won't"
##  [92] "wouldn't"   "shan't"     "shouldn't"  "can't"      "cannot"     "couldn't"   "mustn't"
##  [99] "let's"      "that's"     "who's"      "what's"     "here's"     "there's"    "when's"
## [106] "where's"    "why's"      "how's"      "a"          "an"         "the"        "and"
## [113] "but"        "if"         "or"         "because"    "as"         "until"      "while"
## [120] "of"         "at"         "by"         "for"        "with"       "about"      "against"
## [127] "between"    "into"       "through"    "during"     "before"     "after"      "above"
## [134] "below"      "to"         "from"       "up"         "down"       "in"         "out"
## [141] "on"         "off"        "over"       "under"      "again"      "further"    "then"
## [148] "once"       "here"       "there"      "when"       "where"      "why"        "how"
## [155] "all"        "any"        "both"       "each"       "few"        "more"       "most"
## [162] "other"      "some"       "such"       "no"         "nor"        "not"        "only"
## [169] "own"        "same"       "so"         "than"       "too"        "very"       "will"
```

```
toks <- tokens_remove(toks, pattern = stopwords("en"))
head(toks[[1]], 20)
```

```
##  [1] "Fellow-Citizens" "Senate"          "House"           "Representatives" "Among"
##  [6] "vicissitudes"    "incident"        "life"            "event"           "filled"
## [11] "greater"         "anxieties"       "notification"    "transmitted"     "order"
## [16] "received"        "14th"            "day"             "present"         "month"
```

```
# We can keep words we are interested in
tokens_select(toks, pattern = c("peace", "war", "great*", "unit*"))
```

```
## Tokens consisting of 5 documents and 4 docvars.
## text1 :
## [1] "greater" "United"  "Great"   "United"  "united"  "great"   "great"   "united"
##
## text2 :
## [1] "united"
##
## text3 :
##  [1] "war"    "great"  "United" "great"  "great"  "peace"  "great"  "peace"  "peace"  "United"
## [11] "peace"  "peace"
## [ ... and 2 more ]
##
## text4 :
##  [1] "greatness" "unite"     "unite"     "greater"   "peace"     "peace"     "peace"     "war"
##  [9] "peace"     "greatest"  "greatest"  "great"
## [ ... and 1 more ]
##
## text5 :
## [1] "United" "peace"  "great"  "war"    "war"    "War"    "peace"  "peace"  "peace"
```

```
# Remove "fellow" and "citizen"
toks <- tokens_remove(toks, pattern = c(
    "fellow*",
    "citizen*",
    "senate",
    "house",
    "representative*",
    "constitution"
))

# Build N-grams (onegrams, bigrams, and 3-grams)
toks_ngrams <- tokens_ngrams(toks, n = 2:3)

# Build N-grams based on a structure: keep n-grams that containt a "not"
toks_neg_bigram_select <- tokens_select(toks_ngrams, pattern = phrase("never_*"))
head(toks_neg_bigram_select[[1]], 30)
```

```
## [1] "never_hear"            "never_expected"        "never_hear_veneration" "never_expected_nation"
```

From tokens to document-term-matrix (dtm): steps (3) and (4)

- To make our collection of tokens usable for quantitative analysis, we turn the collection of tokens into a document-term-matrix (dtm, also known as document-feature-matrix, dfm).
- dtms have as rows the document, and as columns the tokens. They contain the count frequency, or sometimes an indicator for whether a given token appears in a document.

To create a dtm, we can use `quanteda`'s `dfm` command, as shown below.

```
dfmat <- dfm(toks)
print(dfmat)
```

```
## Document-feature matrix of: 5 documents, 1,818 features (72.28% sparse) and 4 docvars.
##        features
## docs    among vicissitudes incident life event filled greater anxieties notification transmitted
##   text1     1            1        1    1     2      1       1         1            1           1
##   text2     0            0        0    0     0      0       0         0            0           0
##   text3     4            0        0    2     0      0       0         0            0           0
##   text4     1            0        0    1     0      0       1         0            0           0
##   text5     7            0        0    2     0      0       0         0            0           0
## [ reached max_nfeat ... 1,808 more features ]
```

- Our dtm has five rows for our five documents, and 6,694 (!) columns. Each column is a single token. The dtm is 79.88% sparse, which means that 79.88% of the cells are 0.
- Because our dtm is too large and not informative, we want to trim it and remove columns based on their frequencies. When removing tokens that appear less than two times, we are left with a dtm of 72 columns.

```
dfmat <- dfm(toks)
dfmat <- dfm_trim(dfmat, min_termfreq = 2) # remove tokens that appear less than 1 times
```

From dtm to analysis and insights

- Dtms are the basis of all text analyses.
- They are used to train machine learning methods to predict the sentiment of a text, to classify the documents into clusters, to retrieve missing information, or to predict the autorship.
- Very basic statistics about documents are the **top features** of each document, the frequency of expressions in the corpus

```
topfeatures(dfmat, 10)
```

```
## government        may     public        can     people      shall    country      every         us
##         40         38         30         27         27         23         22         20         20
##    nations
##         18
```

```
# compute word frequencies as top feature
tstat_freq <- textstat_frequency(dfmat, n = 5)

# visualize frequencies in word cloud
textplot_wordcloud(dfmat, max_words = 100)
```

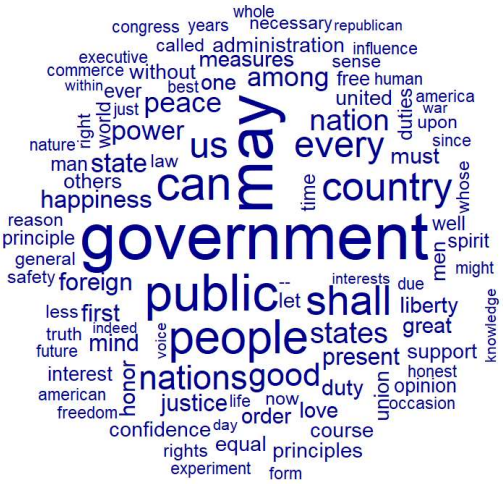


Image Data

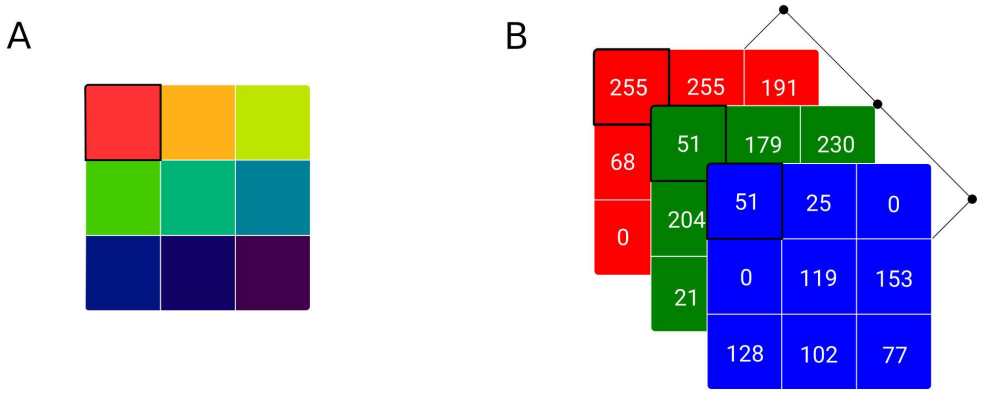
Basic data structure

- There are two important variants of storing digital images: *raster-based images* (for example, jpg files), and *vector-based images* (for example, eps files).
- Raster image: a matrix of pixels, as well as the color of each pixel.
- Vector-based images: text files that store the coordinates of points on a surface and how these dots are connected (or not) by lines.

Raster images

- Images are stored as arrays $X \times Y \times Z$ (X and Y define the number of pixels in each column and row, Z defines the number of layers.)

- Greyscale images usually have only one layer, whereas most colored images have 3 layers.
- For RGB-Images, which are the most common format of colored images, the layers define the intensity of red, green, and blue of each pixel.



(ref:rgb)

Vector images

- Textfiles using a hierarchical data structure to define the shapes, colors and coordinates of the objects shown in an image.
- Typically used for computer drawings, plots, maps, and blueprints of technical infrastructure.
- Typically based on a syntax that is similar to or a version of XML.

Raster-images in R

```
# Load two common packages
library(raster) # for raster images
```

```
## Loading required package: sp
```

```
library(magick) # for wide range of raster and vector-based images
```

```
## Linking to ImageMagick 6.9.12.98
## Enabled features: cairo, freetype, fftw, ghostscript, heic, lcms, pango, raw, rsvg, webp
## Disabled features: fontconfig, x11
```

We can generate images directly in R by populating arrays and saving the plots to disk.

Generating a red image (RGB code: 255,0,0)

```
# Step 1: Define the width and height of the image
width = 300;
height = 300

# Step 2: Define the number of layers (RGB = 3)
layers = 3

# Step 3: Generate three matrices corresponding to Red, Green, and Blue values
red = matrix(255, nrow = height, ncol = width)
green = matrix(0, nrow = height, ncol = width)
blue = matrix(0, nrow = height, ncol = width)

# Step 4: Generate an array by combining the three matrices
image.array = array(c(red, green, blue), dim = c(width, height, layers))
dim(image.array)
```

```
## [1] 300 300   3
```

```
# Step 5: Create RasterBrick
image = brick(image.array)
print(image)
```

```
## class      : RasterBrick
## dimensions : 300, 300, 90000, 3  (nrow, ncol, ncell, nlayers)
## resolution : 0.003333333, 0.003333333  (x, y)
## extent     : 0, 1, 0, 1  (xmin, xmax, ymin, ymax)
## crs        : NA
## source     : memory
## names      : layer.1, layer.2, layer.3
## min values :     255,       0,       0
## max values :     255,       0,       0
```

```
# Step 6: Plot RGB
plotRGB(image)
```

```
## Warning in .couldBeLonLat(x, warnings = warnings): CRS is NA. Assuming it is longitude/latitude
```

```
# Step 7: (Optional) Save to disk
png(filename = "red.png", width = width, height = height, units = "px")
plotRGB(image)
```

```
## Warning in .couldBeLonLat(x, warnings = warnings): CRS is NA. Assuming it is longitude/latitude
```

```
dev.off()
```

```
## png
##   2
```

Vector-images in R

```
# Common Packages for Vector Files
library(xml2)

# Download and read svg image from url
URL <- "https://upload.wikimedia.org/wikipedia/commons/1/1b/R_logo.svg"
Rlogo_xml <- read_xml(URL)

# Data structure
Rlogo_xml
```

```
## {xml_document}
## <svg preserveAspectRatio="xMidYMid" width="724" height="561" viewBox="0 0 724 561" xmlns="http://www.w3.org/2000/svg" xml
ns:xlink="http://www.w3.org/1999/xlink">
## [1] <defs>\n  <linearGradient id="gradientFill-1" x1="0" x2="1" y1="0" y2="1" gradientUnits="obje ...
## [2] <path d="M361.453,485.937 C162.329,485.937 0.906,377.828 0.906,244.469 C0.906,111.109 162.329 ...
## [3] <path d="M550.000,377.000 C550.000,377.000 571.822,383.585 584.500,390.000 C588.899,392.226 5 ...
```

```
xml_structure(Rlogo_xml)
```

```
## <svg [preserveAspectRatio, width, height, viewBox, xmlns, xmlns:xlink]>
##   <defs>
##     <linearGradient [id, x1, x2, y1, y2, gradientUnits, spreadMethod]>
##       <stop [offset, stop-color, stop-opacity]>
##       <stop [offset, stop-color, stop-opacity]>
##     <linearGradient [id, x1, x2, y1, y2, gradientUnits, spreadMethod]>
##       <stop [offset, stop-color, stop-opacity]>
##       <stop [offset, stop-color, stop-opacity]>
##   <path [d, fill, fill-rule]>
##   <path [d, fill, fill-rule]>
```

```
# Raw data
Rlogo_text <- as.character(Rlogo_xml)

# Plot
svg_img = image_read_svg(Rlogo_text)
image_info(svg_img)
```

```
## # A tibble: 1 × 7
##   format width height colorspace matte filesize density
##   <chr>  <int>  <int> <chr>      <lgl>    <int> <chr>
## 1 PNG      724    561 sRGB       TRUE         0 72x72
```

```
svg_img
```

Use case: Optical Character Recognition (OCR)

- Common context to encounter image data in empirical economic research is the digitization of old texts.
- Optical character recognition (OCR) is used to extract text from scanned images.
- R provides a straightforward approach to OCR in which the input is an image file (e.g., a png-file) and the output is a character string.

```
# For Optical Character Recognition
library(tesseract)

# fetch and show image
img <- image_read("https://s3.amazonaws.com/libapps/accounts/30502/images/new_york_times.png")
print(img)
```

```
## # A tibble: 1 × 7
##   format width height colorspace matte filesize density
##   <chr>  <int>  <int> <chr>      <lgl>    <int> <chr>
## 1 PNG      806    550 sRGB       FALSE   714189 38x38
```


```
# zoom in on headline
headline <-
  image_crop(image = img, geometry = '806x180')

headline
```


```
# Extract text via OCR
headline_text <- image_ocr(headline)
cat(headline_text)
```

```
## The New Work Times. [==S=
##
## TITANIC SINKS FOUR HOURS AFTER HITTING ICEBERG;
## 866 RESCUED BY CARPATHIA, PROBABLY 1250 PERISH;
## ISMAY SAFE, MRS. ASTOR MAYBE, NOTED NAMES MISSING
```

References

1. XPATH goes beyond the basic introduction to XML covered in this course and is thus not an exam-relevant topic. If you are interested in learning more about XPATH, W3-schools provides a great introductory tutorial to the topic: https://www.w3schools.com/xml/xpath_intro.asp (https://www.w3schools.com/xml/xpath_intro.asp).↩