Data Handling: Import, Cleaning and Visualisation

Lecture 8: Data Preparation

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

21/11/2018

1 Wrangling with data

Importing a dataset properly is just the first of several milestones until an analysis-ready dataset is generated. In some cases, cleaning the raw data is a necessary step to facilitate/enable proper parsing of the data set in order to import it. However, most of the cleaning/preparing ('wrangling') with the data follows after the proper parsing of structured data. Many aspects of data wrangling are specific to certain datasets and an entire curriculum could be filled with different approaches and tools to address specific problems. Moreover, proficiency in data wrangling is generally a matter of experience in working with data, gained over many years. Here, we focus on two quite general and broadly applicable techniques that are central to cleaning and preparing a dataset for analysis: Simple string operations (find/replace parts of text strings) and reshaping rectangular data (wide to long/long to wide). The former is focused on individual variables at a time, while the latter typically happens at the level of the entire dataset.

1.1 Cleaning data with basic string operations

Recall that most of the data we read into R for analytic purposes is essentially a collection of raw text (structured with special characters). When parsing the data in order to read it into R with high-level functions such as the ones provided in the `readr`-package, both the structure and the types of the data are considered. The resulting `data.frame`/`tibble` might thus contain variables (different columns) of type `character`, `factor`, or `integer`, etc. At this stage it often happens that the raw data is not clean enough for the parser to recognize the data types in each column correctly, and it resorts to just parsing it as `character`. Indeed, if we have to deal with a very messy dataset it can make a lot of sense to constrain the parser such that it reads each column as `character`.

As we will rely on functions provided in the `tidyverse`, we first load this package.

```r
library(tidyverse)
```

Let's create a sample dataset to illustrate some of the typical issues regarding unclean data that we might encounter in empirical economic research (and many similar domains of data analysis).[1]

```r
messy_df <- data.frame(last_name = c("Wayne", "Trump", "Karl Marx"),
                       first_name = c("John", "Melania", ""),
                       gender = c("male", "female", "Man"),
                       date = c("2018-11-15", "2018.11.01", "2018/11/02"),
                       income = c("150,000", "250000", "10000"), stringsAsFactors = FALSE)
```

Assuming we have managed to read in this dataset from a local file (with all columns as type `character`), the next step is to clean each of the columns such that the dataset is ready for analysis. Thereby we want to make sure that each variable (column) is set to a meaningful data type, once it is cleaned. The *cleaning* of the parsed data is often easier to do when the data is of type `character`. Once it is cleaned, however, we can set it to a type that is more useful for the analysis part. For example, in the final dataset a column

[1] The option `stringsAsFactors = FALSE` ensures that all of the columns in this data frame are of type `character`.

containing numeric values should be stored as `numeric` or `integer`, so we can perform math operations on it later on (compute sums, means, etc.).

1.1.1 Find/replace character strings/recode factor levels

Our dataset contains a typical categorical variable: `gender`. In R it is good practice to store such variables as type `factor`. Without really looking at the data values, we might thus be inclined to do the following:

```
messy_df$gender <- as.factor(messy_df$gender)
messy_df$gender
```

```
## [1] male   female Man
## Levels: female male Man
```

The column is now of type `factor`. And we see that R defined the factor variable such that an observation can be one of three categories ('levels'): `female`, `male`, or `Man`. In terms of content, that probably does not make too much sense. If we were to analyze the data later on and compute the share of males in the sample, we would only count one instead of two. Hence, we better *recode* the gender variable of male subjects as `male` and not `Man`. How can this be done programatically?

One approach is to select all entries in `messy_df$gender` that are equal to `"Man"` and replace these entries with `"male"`.

```
messy_df$gender[messy_df$gender == "Man"] <- "male"
messy_df$gender
```

```
## [1] male   female male
## Levels: female male Man
```

Note, however, that this approach is not really perfect, because R still considers `Man` as a valid possible category in this column. This can have consequences for certain types of analyses we might want to run on this dataset later on.[2] Alternatively, we can use a function `fct_recode()` (provided in `tidyverse`), specifically made for such operations with factors.

```
messy_df$gender <- fct_recode(messy_df$gender, "male" = "Man")
messy_df$gender
```

```
## [1] male   female male
## Levels: female male
```

The latter can be very useful when several factor levels need to be recoded at once. Note that in both cases, the underlying logic is that we search for strings that are identical to `"Man"` and replace those values with `"male"`. Now, the gender variable is ready for analysis.

1.1.2 Removing individual characters from a string

The `income` column contains numbers, so let's try to set this column to type `integer`.

```
as.integer(messy_df$income)
```

```
## Warning: NAs introduced by coercion
```

```
## [1]     NA 250000  10000
```

R is warning us that something did not go well when executing this code. We see that the first value of the original column has been replaced with `NA` ('Not Available'/'Not Applicable'/'No Answer'). The reason is that the original value contained a comma (`,`) which is a special character. The function `as.integer()` does

[2] If we perform the same operation on this variable *before* coercing it to a `factor`, this problem does not occur.

not know how to translate such a symbol to a number. Hence, the original data value cannot be translated into a number (integer). In order to resolve this issue, we have to remove the comma (`,`) from this string. Or, more precisely, we will locate this specific character *within* the string and replace it with an empty string (`""`) In order to do so, we'll use the function `str_replace()` (for 'string replace').

```
messy_df$income <- str_replace(messy_df$income, pattern = ",", replacement = "")
```

Now we can succesfully set the column as type ingeger.

```
messy_df$income <- as.integer(messy_df$income)
```

1.1.3 Splitting strings

From looking at the `last_name` and `first_name` columns of our messy dataset, it becomes clear that the last row is not accurately coded. `Karl` should show up in the `first_name` column. In order to correct this, we have to extract a part of one string and store this sub-string in another variable. There are several ways to do this. Here, it probably makes sense to split the original string into two parts, as the white space between `Karl` and `Marx` indicates the separation of first and last name. For this, we can use the function `str_split()`.

First, we split the strings at every occurence of white space (`" "`). Setting the option `simplify=TRUE`, we get a matrix containing the individual sub-strings after the splitting.

```
splitnames <- str_split(messy_df$last_name, pattern = " ", simplify = TRUE)
splitnames
```

```
##      [,1]    [,2]
## [1,] "Wayne" ""
## [2,] "Trump" ""
## [3,] "Karl"  "Marx"
```

As the first two observations did not contain any white space, there was nothing to split there and the function simply returned empty strings `""`. In a second step, we replace empty observations in the `first_name` column with the corresponding values in `splitnames`.

```
problem_cases <- messy_df$first_name == ""
messy_df$first_name[problem_cases] <- splitnames[problem_cases, 1]
```

Finally, we have to correct the `last_name` column by replacing the respective values.

```
messy_df$last_name[problem_cases] <- splitnames[problem_cases, 2]
messy_df
```

```
##   last_name first_name gender       date income
## 1     Wayne       John   male 2018-11-15 150000
## 2     Trump    Melania female 2018.11.01 250000
## 3      Marx       Karl   male 2018/11/02  10000
```

1.1.4 Parsing dates

Finally, we take a look at the `date`-column of our dataset. For many data preparation steps as well as visualization and analysis, it is advantageous to have times and dates properly parsed as type `Date`. In practice, dates and times are often particularly messy because no unique standard has been used to define the format in the data collection phase. This seems also to be the case in our dataset. In order to work with dates, we load the `lubridate` package.

```
library(lubridate)
```

This package provides several functions to parse and manipulate date and time data. From looking at the `date`-column we see that the format is basicall year, month, day. We can thus use the `ymd()`-function provided in the `lubridate`-package in order to parse the column as `Date` type.

```
messy_df$date <- ymd(messy_df$date)
```

Note how this function automatically recognizes how different special characters have been used in different observations to separate years from months/days.

Now, our dataset is cleaned up and ready to go.

```
messy_df
```

```
##   last_name first_name gender       date income
## 1     Wayne       John   male 2018-11-15 150000
## 2     Trump    Melania female 2018-11-01 250000
## 3      Marx       Karl   male 2018-11-02  10000
```

```
str(messy_df)
```

```
## 'data.frame':    3 obs. of  5 variables:
##  $ last_name : chr  "Wayne" "Trump" "Marx"
##  $ first_name: chr  "John" "Melania" "Karl"
##  $ gender    : Factor w/ 2 levels "female","male": 2 1 2
##  $ date      : Date, format: "2018-11-15" "2018-11-01" "2018-11-02"
##  $ income    : int  150000 250000 10000
```

1.2 Reshaping datasets

Apart from cleaning and standardizing individual data columns, preparing a dataset for analysis often involves bringing the entire dataset in the right 'shape'. Typically, what we mean by this is that in a table-like (two-dimensional) format such as `data.frames` and `tibbles`, data with repeated observations for the same unit can be displayed/stored in either *long* or *wide* format. It is often seen as good practice to prepare data for analysis in *long* ('tidy') format. This way we ensure that we follow the ('tidy') paradigm of using the rows for individual observations and the columns to describe these observations.[3] Tidying/reshaping a dataset in this way thus involves transforming columns into rows (i.e., *melting* the dataset). In the following, we first have a close look at what this means conceptually and then apply this technique in two examples.

1.2.1 Tidying messy datasets.

Consider the following stylized example (Wickham 2014).

person	treatmenta	treatmentb
John Smith	NA	2
Jane Doe	16	11
Mary Johnson	3	1

The table shows observations of three individuals participating in an experiment. In this experiment, the subjects might have been exposed to treatment a and/or treatment b. Their reaction to either treatment is measured in numeric values (the results of the experiment). From looking at the raw data in its current shape, this is not really clear. While we see which numeric value corresponds to which person and treatment, it is not clear what this value is. One might, for example, wrongly assume that the numeric values refer to the treatment intensity of a and b. Such interpretation would be in line with the idea of columns containing variables and rows observations. But, considering what the numeric values actually stand for, we realize that

[3] Depending on the dataset, however, an argument can be made that storing the data in wide format might be more efficient (using up less memory) than long format.

the columns actually are not *names of variables* but *values* of a variable (the categorical variable `treatment`, with levels `a` and `b`).

Now consider the same data in 'tidy' format (variables in columns and observations in rows).

person	treatment	result
John Smith	a	NA
Jane Doe	a	16
Mary Johnson	a	3
John Smith	b	2
Jane Doe	b	11
Mary Johnson	b	1

This *long*/*tidy* shape of the dataset has several advantages. First, it is now clear what the numeric values refer to. Second, in this format it is much easier to filter/select the observations.

1.2.2 Gathering ('wide to long')

In the `tidyverse` context, we call the transformation of columns to rows ('wide to long') 'gathering'. That is we 'gather' columns into keys and values. A most typical situation where this has to be done in applied data analysis is when a dataset contains for the same subjects several observations over time. To illustrate how *gathering* works in practice, consider the following example dataset (extending on the example above).

```
wide_df <- data.frame(last_name = c("Wayne", "Trump", "Marx"),
                       first_name = c("John", "Melania", "Karl"),
                       gender = c("male", "female", "male"),
                       income.2018 = c("150000", "250000", "10000"),
                      income.2017 = c( "140000", "230000", "15000"),stringsAsFactors = FALSE)
wide_df
```

```
##   last_name first_name gender income.2018 income.2017
## 1     Wayne       John   male      150000      140000
## 2     Trump    Melania female      250000      230000
## 3      Marx       Karl   male       10000       15000
```

The two last columns contain both information on the same variable (`income`), but for different years. We thus want to gather these two columns in a new `year` and `income` column, ensuring that columns correspond to variables and rows correspond to observations. For this, we call the `gather()`-function as follows:

```
long_df <- gather(wide_df, income.2018, income.2017, key = "year", value = "income")
long_df
```

```
##   last_name first_name gender        year income
## 1     Wayne       John   male income.2018 150000
## 2     Trump    Melania female income.2018 250000
## 3      Marx       Karl   male income.2018  10000
## 4     Wayne       John   male income.2017 140000
## 5     Trump    Melania female income.2017 230000
## 6      Marx       Karl   male income.2017  15000
```

We can further clean the `year` column to only contain the respective numeric values.

```
long_df$year <- str_replace(long_df$year, "income.", "")
long_df
```

```
##   last_name first_name gender year income
## 1     Wayne       John   male 2018 150000
## 2     Trump    Melania female 2018 250000
```

```
## 3      Marx       Karl   male 2018  10000
## 4     Wayne       John   male 2017 140000
## 5     Trump    Melania female 2017 230000
## 6      Marx       Karl   male 2017  15000
```

1.2.3 Spreading ('long to wide')

As we want to adhere to the 'tidy' paradigm of keeping our data in long format, the transformation of 'long to wide' is less common. However, it might be necessary if the dataset at hand is particularly messy. The following example illustrates such a situation.

```r
weird_df <- data.frame(last_name = c("Wayne", "Trump", "Marx", "Wayne", "Trump", "Marx",
                                     "Wayne", "Trump", "Marx"),
                       first_name = c("John", "Melania", "Karl", "John", "Melania", "Karl",
                                      "John", "Melania", "Karl"),
                       gender = c("male", "female", "male", "male", "female", "male",
                                  "male", "female", "male"),
                       value = c("150000", "250000", "10000", "2000000", "5000000", "NA",
                                 "50", "25", "NA"),
                       variable = c( "income", "income", "income", "assets", "assets", "assets",
                                     "age", "age", "age"), stringsAsFactors = FALSE)
weird_df
```

```
##   last_name first_name gender   value variable
## 1     Wayne       John   male  150000   income
## 2     Trump    Melania female  250000   income
## 3      Marx       Karl   male   10000   income
## 4     Wayne       John   male 2000000   assets
## 5     Trump    Melania female 5000000   assets
## 6      Marx       Karl   male      NA   assets
## 7     Wayne       John   male      50      age
## 8     Trump    Melania female      25      age
## 9      Marx       Karl   male      NA      age
```

While the data is somehow in long format, the rule that each column should correspond to a variable (and vice versa) is ignored. Data on income, assets, as well as the age of the individuals in the dataset are all put in the same column. We can call the function `spread()` with the two parameters `key` and `value` to correct this.

```r
tidy_df <- spread(weird_df, key = "variable", value = "value")
tidy_df
```

```
##   last_name first_name gender age  assets income
## 1      Marx       Karl   male  NA      NA  10000
## 2     Trump    Melania female  25 5000000 250000
## 3     Wayne       John   male  50 2000000 150000
```

References

Wickham, Hadley. 2014. "Tidy Data." *Journal of Statistical Software, Articles* 59 (10): 1–23. https://doi.org/10.18637/jss.v059.i10.