

Lecture 3: Data Wrangling

Economía Laboral

Junghanss, Juan Cruz

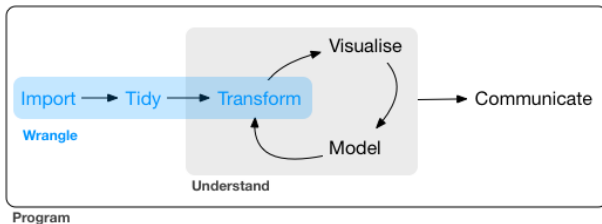
Universidad del CEMA

2nd Semester 2024

Data Wrangling

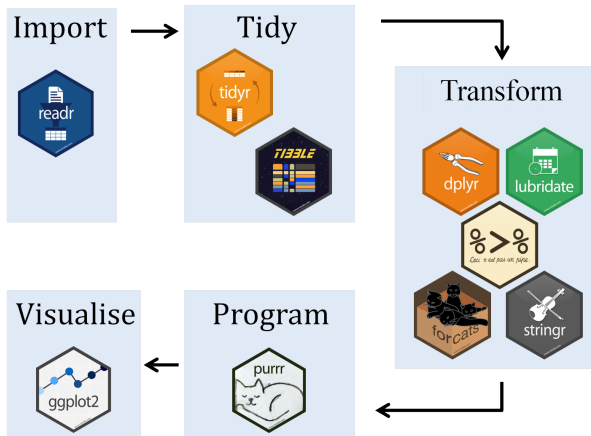
Wrangle \implies import, tidy & transform

Data-wrangling focuses on changing the data format by translating “raw” data into a more usable form. The goal of data wrangling is to assure quality and useful data.



Data Wrangling

The workflow in terms of packages:

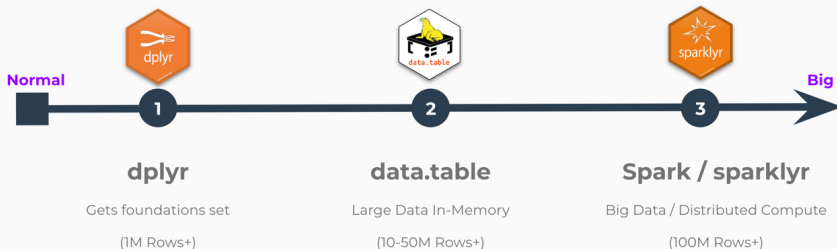


It's not just cleaning OR transforming. A data wrangling process consists of reorganizing, transforming and mapping data from one raw form into another in order to make it more *usable* and *valuable* for a variety of downstream uses including analytics.

In contrast, for example, data cleaning focuses on removing erroneous data from your dataset and data transformation focuses on converting data from one format into another.



Data Wrangling Tools by Dataset Size



Step 1: Import

To import data you use the **readr** package. Functions are concerned with turning flat files into data frames:

- `read_csv()` for comma separated files, `read_csv2()` for semicolon-separated files.
- `read_fwf()` for “fixed-width” files

For *rectangular data* you can use:

- **haven** for reading SPSS, Stata and SAS files.
- **readxl** for Excel files.

Tidy data

Step 2: Tidy

Recall from last lecture:

A dataset is tidy if it fulfills the following conditions:

- 1 Each variable must have its own column.
- 2 Each observation must have its own row.
- 3 Each value must have its own cell.

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20095360
Brazil	1999	37737	17206362
Brazil	2000	80488	17404898
China	1999	212258	1272015272
China	2000	216766	128042583

variables

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20095360
Brazil	1999	37737	17206362
Brazil	2000	80488	17404898
China	1999	212258	1272015272
China	2000	216766	128042583

observations

country	year	cases	population
Afghanistan	99	745	19987071
Afghanistan	00	2666	20095360
Brazil	99	37737	17206362
Brazil	00	80488	17404898
China	99	212258	1272015272
China	00	216766	128042583

values

How to tidy data? Gathering and Spreading

*Functions from the package **tidyr***

Gathering: you gather data when a variable is spread across multiple columns.

country	year	cases
Afghanistan	1999	745
Afghanistan	2000	2666
Brazil	1999	37737
Brazil	2000	80488
China	1999	212258
China	2000	213766

country	1999	2000
Afghanistan	745	2666
Brazil	37737	80488
China	212258	213766

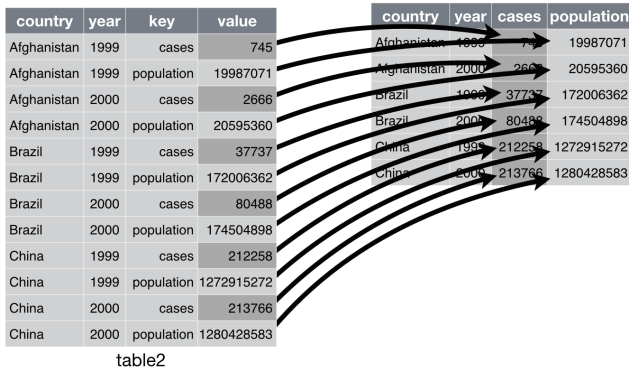
table4

How to tidy data? Gathering and Spreading

Another example of gathering:

How to tidy data? Gathering and Spreading

Spreading: you spread the data when an observation is scattered across multiple rows.



How to tidy data? Separating and Uniting

Separating: you separate, i.e. pull apart, one column into multiple columns, by splitting wherever a separator character appears.




country	year	rate
Afghanistan	1999	745 / 19987071
Afghanistan	2000	2666 / 20595360
Brazil	1999	37737 / 172006362
Brazil	2000	80488 / 174504898
China	1999	212258 / 1272915272
China	2000	213766 / 1280428583

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

table3

How to tidy data? Separating and Uniting

Uniting: you unite, i.e. combine, multiple columns into a single column. It is the inverse of `separate()`.



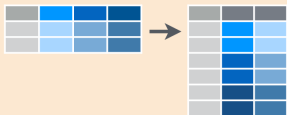
country	year	rate
Afghanistan	1999	745 / 19987071
Afghanistan	2000	2666 / 20595360
Brazil	1999	37737 / 172006362
Brazil	2000	80488 / 174504898
China	1999	212258 / 1272915272
China	2000	213766 / 1280428583

country	century	year	rate
Afghanistan	19	99	745 / 19987071
Afghanistan	20	0	2666 / 20595360
Brazil	19	99	37737 / 172006362
Brazil	20	0	80488 / 174504898
China	19	99	212258 / 1272915272
China	20	0	213766 / 1280428583

table6

How to tidy data?

Most important functions:



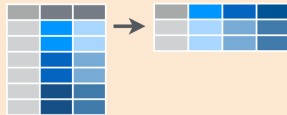
tidyr::gather(cases, "year", "n", 2:4)

Gather columns into rows.



tidyr::separate(storms, date, c("y", "m", "d"))

Separate one column into several.



tidyr::spread(pollution, size, amount)

Spread rows into columns.



tidyr::unite(data, col, ..., sep)

Unite several columns into one.

Data Transformation

We already saw some data transformation in our last lecture. The most important functions from **dplyr** package are:

select



filter



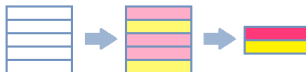
arrange



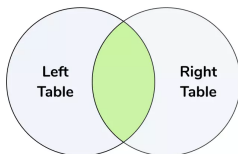
mutate



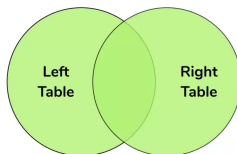
summarise



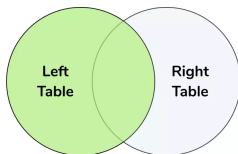
Types of Joins



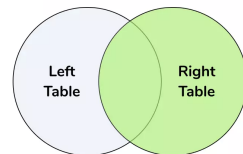
Inner Join



Full Join



Left Join



Right Join

Transforming Relational Data

Combine Data Sets

a		b	
x1	x2	x1	x3
A	1	A	T
B	2	B	F
C	3	D	T

+

=

Mutating Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NA

dplyr::left_join(a, b, by = "x1")

Join matching rows from b to a.

x1	x3	x2
A	T	1
B	F	2
D	T	NA

dplyr::right_join(a, b, by = "x1")

Join matching rows from a to b.

x1	x2	x3
A	1	T
B	2	F

dplyr::inner_join(a, b, by = "x1")

Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NA
D	NA	T

dplyr::full_join(a, b, by = "x1")

Join data. Retain all values, all rows.

Filtering Joins

x1	x2
A	1
B	2

dplyr::semi_join(a, b, by = "x1")

All rows in a that have a match in b.

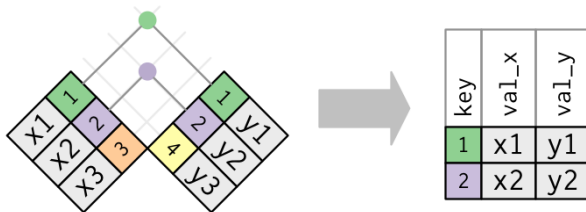
x1	x2
C	3

dplyr::anti_join(a, b, by = "x1")

All rows in a that do not have a match in b.

Inner Join

The simplest type of join is the **inner join**. An inner join matches pairs of observations whenever their keys are equal:



Unmatched rows are not included in the result. It's easy to lose observations.

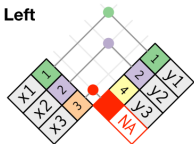
Outer Join

An inner join keeps observations that appear in both tables. An **outer join** keeps observations that appear in at least one of the tables. There are three types of outer joins:

- A **left join** keeps all observations in x.
- A **right join** keeps all observations in y.
- A **full join** keeps all observations in x and y.

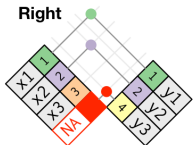
Outer Join

Left



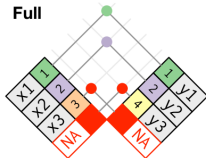
key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA

Right



key	val_x	val_y
1	x1	y1
2	x2	y2
4	NA	y3

Full



key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA
4	NA	y3

Outer Join

The most commonly used join is the **left join**: you use this whenever you look up additional data from another table, because it preserves the original observations even when there isn't a match. The left join should be your default join: use it unless you have a strong reason to prefer one of the others.

Mutating Joins in R

R base::**merge()** can perform all four types of mutating join:

dplyr	merge
<code>inner_join(x, y)</code>	<code>merge(x, y)</code>
<code>left_join(x, y)</code>	<code>merge(x, y, all.x = TRUE)</code>
<code>right_join(x, y)</code>	<code>merge(x, y, all.y = TRUE)</code>
<code>full_join(x, y)</code>	<code>merge(x, y, all.x = TRUE, all.y = TRUE)</code>

The advantages of the specific dplyr verbs is that they more clearly convey the intent of your code: the difference between the joins is really important but concealed in the arguments of `merge()`. dplyr's joins are considerably faster and don't mess with the order of the rows.

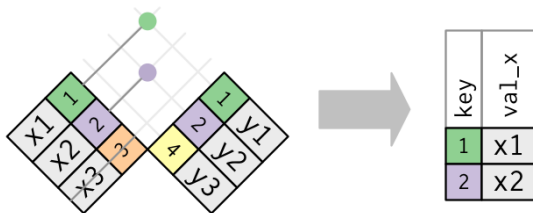
Filtering Joins in R

Filtering joins match observations in the same way as mutating joins, but affect the observations, not the variables. There are two types:

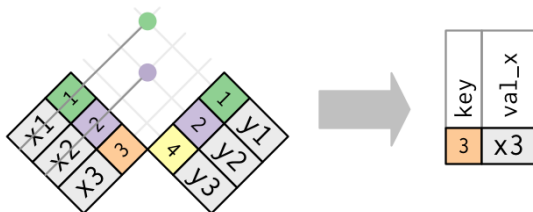
- **semi_join(x, y)** keeps all observations in x that have a match in y.
- **anti_join(x, y)** drops all observations in x that have a match in y.

Filtering Joins in R

A semi-join looks like this: only the existence of a match is important; it doesn't matter which observation is matched.



The inverse of a semi-join is an anti-join. An anti-join keeps the rows that don't have a match:



There are three most common reasons for data to be missing:

- **Missing at random:** sucede cuando la propensión a que un punto falte no está relacionada con los datos faltantes, pero si se relaciona con algunos datos observados.
- **Missing completely at random:** el hecho de que ciertos valores falten no tiene nada que ver con su valor hipotético y con los valores de las otras variables.
- **Missing not at random:** hay dos posibles razones. Los NAs dependen de los valores hipotéticos del dato faltante o dependen del valor de otra variable.

Data Transformation for Missing Values

Values can be missing in two ways:

- **Explicitly:** flagged with NA or NAN.
- **Implicitly:** just not presented in the dataset.

Think about it like this: “An explicit missing value is the presence of the absence and an implicit missing value is the absence of the presence.”

Missing Values - Tests

We can make some tests in R for missing values:

- 1 To identify NAs use **is.na()** which returns a logical vector with TRUE in the element locations that contain missing values represented by NA. The function will work on vectors, lists, matrices and dfs.
- 2 To identify the location or the number of NAs we can leverage the **which()** and **sum()** functions

Example (identify location of NAs in df)

```
which(is.na(df))
```

Example (identify count of NAs in df)

```
sum(is.na(df))
```

Missing Values - Tests

For data frames, a convenient shortcut to compute the total missing values in each column is to use **colSums()**:

Example (total NAs in each df column)

```
colSums(is.na(df))
```

Missing Values - Treatment

How to deal with missing values? There are several ways, none of them is the correct, but the optimal depends on your data, model, preferences, etc.

- ① **Fill**¹: you can fill the missing values with the following values;
 - Median: compute the median of the variable.
 - Mean: take the average of the variable.
 - Last observation carried forward: you replace it by the most recent nonmissing value. You can use the function `fill()`.
 - Alternatively, you can use the function `complete()` in R and just fill with “NA” over the empty cells.
- ② **Drop**: you can just drop the entire observation, i.e. row. Use the function **`drop_na()`**.

You will face a bias trade-off by treating with missing values.

¹En estadística se denomina “imputación” de datos

Missing Values - Treatment for Time Series Data

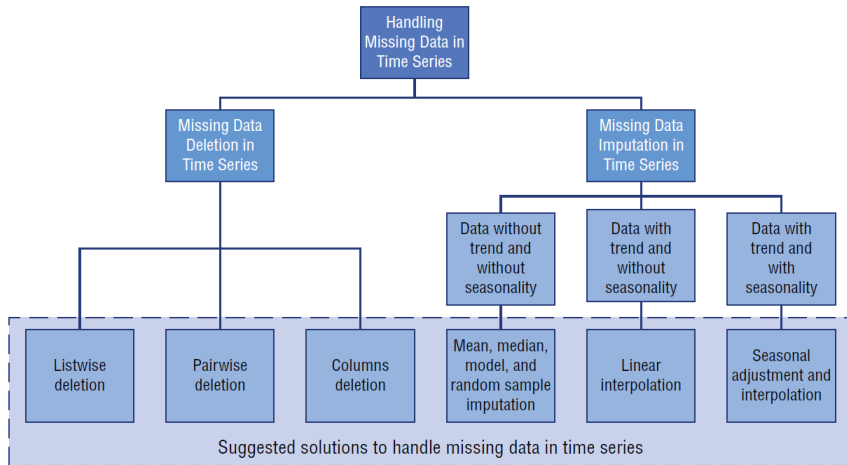


Figure: Fuente: Machine Learning for Time Series Forecasting with Python-Wiley (2020)

Missing Values - Treatment for Time Series Data

- *Linear interpolation*: este método trabaja bien para una serie de tiempo con algo de tendencia, pero no es ajustable para datos con estacionalidad.
- *Ajuste estacional e Interpolación lineal*: este método es mejor para datos con estacionalidad y tendencia.
- *Media, Mediana y Moda*: Computando la media general, mediana o moda es un método muy básico de imputación, y es la única función probada que no aprovecha las características de la serie temporal o la relación entre las variables. Es muy rápido, pero tiene también desventajas. Una de estas es que la imputación con la media reduce la varianza en el conjunto de datos.

Missing Values - Some tips for Time Series data

Concerning raw data quality (for time series): the **missing value rate** should not be greater than 10 percent for any given time period:

$$\text{MVR} = \frac{n_{NA}}{N} \cdot 100$$

donde MVR es el Missing Value Rate, n_{NA} es el número de valores faltantes (missing) y N es la cantidad total de observaciones.

Appendix: Data Storage

To storage data in different formats you can use the package **readr**

- `write_csv()` to generate a comma separated file.
- `write_delim()` to write a `DataFrame` to a delimited file. The default delimiter is space.

Or you can use the package **writexl**;

- `write_xlsx()` to write an Excel file.

More on loading and saving data in R:

<https://rstudio-education.github.io/hopr/dataio.html>

Review Questions

- 1 How do you import data? How do you storage data?
- 2 What means data wrangling?
- 3 What is tidy data? Which methods are mostly used to make data tidy?
- 4 What are the differences between mutating and filtering joins? Explain each type of join.
- 5 How are missing values classified? And how are they treated?