Tibbles & Tidy data

Lecture 3

Federica Fusi

MSCA, UIC

Updated: 2021-01-26

Introduction

Tidyverse package

We will start working with the tidyverse package which is another approach (syntax) to R.

```
#If you haven't installed it yet
install.packages("tidyverse")
```

```
# if you have installed it already
library(tidyverse)
```

```
# You might occassionally need to update your packages
# R generally gives you a warning when you use the library command about
update.packages("tidyverse")

# If you don't specify the package, it will update all your packages
update.packages()

# If you need to remove a package
remove.packages("tidyverse")
```

Some data

We are going to use this dcldata package for some data that we are going to use for examples

Source: https://rdrr.io/github/dcl-docs/dcldata/

Book with examples: Data Wrangling, Sara Altman & Bill Behrman

```
install.packages("remotes")
remotes::install_github("dcl-docs/dcldata")
```

```
library("dcldata")
```

Tibbles

Last week we worked with dataframes. This week, we will learn about tibbles which are the dataframes of tidyverse.

Let's upload the same dataset in both dataframe and tibble format.

```
data_df = as.data.frame(congress) # dataframe

data_tb = as_tibble(congress) # tibble
```

> Start noticing that tidyverse prefer an underscore in functions compared to traditional R where we use a dot
>
> Note that you can always switch from one to the other using these functions.

When you open a dataset

These are some of the first things that you always want to do when you open a dataset.

Check its dimensions, colnames, and have a look at the data

```
# Dimensions of your dataset
dim(data_df)
nrow(data_df)
ncol(data_tb)

# Column names
colnames(data_tb)

# Full picture of your dataset
str(data_tb)
summary(data_tb)

# View pieces of your data (default is 5 rows)
head(data_tb)
head(data_tb, 10)
tail(data_tb, 1)

# View your data
View(data_tb)
```

Tibbles vs dataframes

Let's start by printing th dataframe and the tibble and note a few differences in how they appear.

```
data_df

data_tb
```

An advantage of tibble is that they don't print out the entire dataset - which is very inconvenient when, by mistake, you type the name of very large datasets.

Note, again, that you can visualize both tibble and dataframe using the View function.

```
View(data_df)

View(data_tb)
```

Tibbles vs dataframes

Note that, in tibbles, each column reports its type

- int stands for integers (4, 5, 6)
- dbl stands for doubles, or real numbers (4.4, 5.55, 6.66)
- chr stands for character vectors, or strings ("federica", "fusi")
- dttm stands for date-times (a date + a time) (12:24:2020:08:06)
- lgl stands for logical, vectors that contain only TRUE or FALSE.
- fctr stands for factors, which R uses to represent categorical variables with fixed possible values ("car", "bus", "train")
- date stands for dates (01/12/2020)

Tibbles vs dataframes

The functions that we learned for dataframes work for tibbles too.

You can call a column by its name and perform functions with it

```
table(data_df$state)

table(data_tb$state)
```

You can also extract a column based on its position (remember numerical indexing?)

```
data_df[1, 3]

data_tb[1, 3]
```

Tibbles vs. dataframes

But note the class of the ouput:

```
class(data_df[1, 3])

class(data_tb[1, 3])
```

Tibbles: Do not change class of the output: a tibble always returns a tibble

Another small difference: let's say you want to look at the column "division" but you don't remember its full name, so you just write 'divis'.

```
table(data_df$divis)

table(data_tb$divis)
```

Note that in the dataframe, R tries to guess your column. It doesn't not in the tibble dataframe (more precision).

Tibbles vs dataframes

- Do not change class of the output: a tibble always returns a tibble
- Do not try to guess the column names: always matches it!
- Are easy to print

BUT

- They don't work with all commands so...don't forget about dataframes!!
- You can always go from a tibble to a dataframe

From now on, we will mostly work with tibbles unless we have a reason not to.

Column names

Just a couple of important things about column names before we get into tidy data.

We already saw that column names work in the same way they do for dataframes.

You can always rename a column using the rename function:

rename(newname = oldname)

```
data_tb2 <- # New tibble with renamed columns
  rename(data_tb,  # Original tibble
         first_lastname = name, # new and old name
         geo_division = division) # new and old name
```

Column names

Note that you can potentially start a column name with a symbol or a number but it will require you to use a special symbol.

```
#Very silly example!!
data_tb2 <- rename(data_tb2, `%vote` = age)

data_tb2
```

I would advise against using these names but it might happen that you find them in datasets. If that's the case, I would rename the columns.

```
data_tb2 <- rename(data_tb2, age = `%vote`)

data_tb2
```

Column names

Some other helpful commands when you are re-naming columns

tolower and toupper they, respectively, transform a string into all lower case or all upper case.

If you want to change all the column names, you can use them with the colnames command.

colnames(YourTibble) <- toupper(colnames(YourTibble))

```
colnames(data_tb2)

colnames(data_tb2) <- toupper(colnames(data_tb2))

colnames(data_tb2) # Double check if you achieved your task
```

This might be helpful if the columns mix upper and lower case and you cannot remember how to spell them out.

Final notes

Always make sure that your dataset has well named columns.

Same rules as for object names:

1. Column names should be intuitive
2. Keep them short
3. Try to be consistent (all lower or upper case, similar separation sign, or all attached)

Break

Tidy data

Tidy data

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

Note:

- A variable is a quantity, quality, or property that you can measure.
- An observation is one unique unit of analysis (e.g., a country, a country in a given year). This depends on your analysis
- Values are all the individual cells in your datasets which are stored in variables or observations.

Let's see some examples

The 'lockdown' dataset

What is the problem with this dataset?

What are the variables of this dataset?

Activity	Time_Reality	Time_Expectations
Working out	30	0
Cooking	0	60
Eating	60	0
Reading	45	0
Learning Spanish	0	60
Painting my room	0	60

The 'lockdown' dataset

Three variables:

1. Activity
2. Time
3. Reality / Expectations

The column names 'Time_Reality' and 'Time_Expectations' are actually a variable indicating whether you are referring to a 'real' activity or an 'expected' one but the are store as column name.

The 'lockdown' dataset

We want a dataset that looks like this:

Activity	Type	Time
Working out	Time_Reality	30
Cooking	Time_Expectations	60
Eating	Time_Reality	60
Reading	Time_Reality	45
Learning Spanish	Time_Expectations	60
Painting my room	Time_Expectations	60

Another example

What is the problem with this dataset?

How would you solve it? I.e., how would you make your data tidy?

Name	Sex_Age
Mary	F_33
John	M_40
Eve	F_45
Simon	M_31
Daniel	M_23

Another example

Age and sex are two variables stored in one column.

The column 'Sex_Age' includes two variables.

Name	Sex	Age
Mary	F	33
John	M	40
Eve	F	45
Simon	M	31
Daniel	M	23

A quick consideration

Consider that these two examples are slightly different in the way they work

Activity	Time_Reality	Time_Expectations
Working out	30	0
Cooking	0	60
Eating	60	0

Activity	Type	Time
Working out	Time_Reality	30
Cooking	Time_Expectations	60
Eating	Time_Reality	60

We take the two column names (or labels) and we transform them into a variable called 'Type'.

We store the values from these columns in a new one called 'Time'.

A quick consideration

In this other case, we take a column and we divided into two variables.

Name	Sex_Age
Mary	F_33
John	M_40
Eve	F_45

Name	Sex	Age
Mary	F	33
John	M	40
Eve	F	45

Last example

What are the issues with this dataset?

How would you fix them? I.e., how would you make your data tidy?

City_State	Crime rate_1	Crime rate_2	Year_1	Year_2
Chicago_IL	1100	1089	2019	2020
New York_NY	538	550	2019	2020
Las Vegas_NE	618	580	2019	2020
Phoenix_AZ	761	743	2019	2020
Los Angeles_CA	761	765	2019	2020
San Francisco_CA	715	720	2019	2020

Last example

The column 'City_State' contains two variables. We need to split it.

We need to make the dataset wider by adding one more column.

City	State	Crime rate_1	Crime rate_2	Year_1	Year_2
Chicago	IL	1100	1089	2019	2020
New York	NY	538	550	2019	2020
Las Vegas	NE	618	580	2019	2020
Phoenix	AZ	761	743	2019	2020
Los Angeles	CA	761	765	2019	2020
San Francisco	CA	715	720	2019	2020

Last example

Each row contains two observations. Unit of analysis is city-year. One observation is one city in a given point of time. So, each row should reflect this.

We are making the dataset longer by adding new rows.

City	State	Crime rate	Year
Chicago	IL	1100	2019
Chicago	IL	1089	2020
New York	NY	538	2019
New York	NY	550	2020
Las Vegas	NE	618	2019
Las Vegas	NE	580	2020
Phoenix	AZ	761	2019
Phoenix	AZ	743	2020
Los Angeles	CA	761	2019
Los Angeles	CA	765	2020
San Francisco	CA	715	2019

How we do it in R

Let's have a look at table4a

```
table4a
```

What are the issues with this dataset?

Each row is more than one observation (country-year).

The column labels are not labels but variables.

How would you fix them? I.e., how would you make your data tidy?

How we do it in R

county	year	num_cases
Afghanistan	1999	745
Afghanistan	2000	2666
Brazil	1999	37373
Brazil	2000	80488
China	1999	212258
China	2000	213766

- We want to take the two variable values, 1999 and 2000, and store them in a new variable called "year".
- The values that are currently stored within those two 'variables' will be stored in a new variable called "num_cases".
- Note that we are making the table longer.

pivot_longer

Takes your column names (labels) and transform them into a variable

```
# Always save your dataset as a new one. R has no 'go back' option
table_4aR =
# Call your dataset
pivot_longer(table4a,
             # Identify the column of interest
             c(`1999`, `2000`),
             # Create a new column where you'll store the names of the c
             names_to = "year",
             # Create a new column where you'll store the value of a col
             values_to = "num_cases")
```

Note: Pivot longer makes the dataset longer by increasing the number of rows and decreasing the number of columns.

Your turn

Look at the dataset *example_eagle_pairs*.

The dataset shows the number of observed bald eagle breeding pairs across years and states.

Use pivot_longer to re-organize the data.

```
# Always save your dataset as a new one. R has no 'go back' option
___________ =
# Call your dataset
pivot_longer(_____________,
             # Identify the column of interest
             _____________,
             # Create a new column where you'll store the names of the c
             names_to = ___________,
             # Create a new column where you'll store the value of a col
             values_to = __________)
```

pivot_wider

Let's now have a look at this table *table2*.

What are the issues with this dataset?

How would you fix them? I.e., how would you make your data tidy?

```
table2
```

pivot_wider

Observations are spread across two rows, one for income and one for rent (UofA: country-year)

'Cases" and "population' are two variables but they are collapsed in one column instead than each having its own column.

How would a tidy dataset look like?

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

pivot_wider

```
# Save your output as a new dataset
table2_R =
# Call the dataset that you want to modify
pivot_wider(table2,
            # Indicate the column from where the name(s) of the new colu
            names_from = "type" ,
            # Indicate the column(s) from where the values of the new co
            values_from = "count")
```

Your turn

Use pivot_wider to tidy the data from the table us_rent_income

Remember to visualize the data first in your head (even better: on a piece of paper). Then think about the code.

```
# Save your output as a new dataset
_____________=
# Call the dataset that you want to modify
pivot_wider(_________________,
            # Indicate the column from where the name(s) of the new colu
            names_from = ___________ ,
            # Indicate the column(s) from where the values of the new co
            values_from = _____________)
```

Arguments

Both functions have several arguments:

- https://tidyr.tidyverse.org/reference/pivot_longer.html (pivot_longer)
- https://tidyr.tidyverse.org/reference/pivot_wider.html (pivot_wider)

I am going to review some of the most interesting one.

names_to (pivot_longer)

names_to specify the name(s) of the column to be created to store the column names

```
# Always save your dataset as a new one. R has no 'go back' option
table_4aR =
# Call your dataset
pivot_longer(table4a,
             # Identify the column of interest
             c(`1999`, `2000`),
             # Create a new column where you'll store the names of the c
               names_to = "year",
             # Create a new column where you'll store the value of a col
             values_to = "num_cases")
```

names_prefix (pivot_longer)

names_prefix remove a regular expression from the start of each variable names

```
example_gymnastics_1

# What is the problem with this dataset?
```

```
# Save a new tibble
example_gymnastics_1R =
  # Call tibble to be modified
  pivot_longer(example_gymnastics_1,
               #Identified columns
               c("score_vault", "score_floor"),
               # New column to save names
               names_to = "exercise_type",
               # New column to save values
               values_to = "exercise_score")
```

names_prefix (pivot_longer)

We might not want to keep "score" in each column. We might want to have clean variable's values (vault, floor...)

```
example_gymnastics_1R =
  # Call current tibble
  pivot_longer(example_gymnastics_1,
               #Identify columns
               c("score_vault", "score_floor"),
               #Create new column to store previous column names
               names_to = "exercise_type",
               #Eliminate the same prefix from all column names
                 names_prefix = "score_",
               #Create new column to store values from previous columns
               values_to = "exercise_score")
```

names_sep (pivot_longer)

names_sep you can broke the column names into multiple parts based on a consistent symbols

```
example_gymnastics_2
```

Each column name store two variables!

```
example_gymnastics_2R =
  pivot_longer(example_gymnastics_2,
               # You can use this notation to indicate all columns minus
               cols = -country,
               #Indicate the separator to split the column names
               names_sep = "_",
               names_to = c("exercise_type", "year"),
               values_to = "exercise_score")
```

names_pattern (pivot_longer)

More flexible: names_pattern (syntax is much more complex, we will talk about it with strings)

names_transform (pivot_longer)

names_transform or values_trasform specify the new class of the names (values) colunm (default: character)

names_transform = list(varName = as.numeric)

```
# Look at the class of new columns that we created
example_gymnastics_2R
```

Let's say that you want "year" to be a number

```
example_gymnastics_2R =
  pivot_longer(example_gymnastics_2,
               cols = -country,
               #Indicate the separator to split the column names
               names_sep = "_",
               names_to = c("exercise_type", "year"),
               values_to = "exercise_score",
               # transform the class of the column
               names_transform = list(year = as.numeric))
```

Your turn

Look at table *example_gymnastics_3*

Make sure the the column 'year' stays numberic.

pivot_wider arguments

names_prefix and *names_sep* can be used with pivot_wider as well but they perform slightly different tasks.

This dataset includes pairs of twins. Let's say we want to group them, so that each row represent a pair (or a family) instead than an individual.

```
example_twins
```

```
example_twinsR =
pivot_wider(example_twins,
            names_from = "n",
            values_from = "name")
```

pivot_wider arguments

Let's say that we don't like the new column names as they don't mean much. We want to rename as "Twin_1" and "Twin_2"

```
example_twinsR =
pivot_wider(example_twins,
            names_from = "n",
            names_prefix = "Twin_",
            values_from = "name")
```

pivot_wider arguments

We can group names from multiple columns' values and separate them with a common separator

```
example_acs_2
```

```
example_acs_2 =
pivot_wider(example_acs_2,
            names_from = c("variable", "measure"),
            names_sep = "_",
            values_from = "value")
```

Quick note

These are *not* all the arguments included in the pivot_longer and pivot_wider functions.

As you use them on your data, you should always double check if there are other arguments that might help you clean your data.

Break

Join data

Uploading data in R

If you used R before, you are probably used to these two commands to upload data:

- setwd() to indicate the folder where your data are located
- getwd to find the folder that R is currently using

What is the problem with this approach? If you send your R code to someone else, they need to change the directory. If you upload data from multiple folders, it gets much more complicated.

R projects

Easy solution: using R projects.

- Projects create folder where you can save your code + data.
- Every time, you should open your data from your project folder instead than R studio.
- This process automatically set the directory so you can just open your data
- If someone else put the code in the project folder, they can replicate your code.

Open our data

```
immigration = read_csv("Immigration.csv")

education = read_csv("Education.csv")

income = read_csv("Income.csv")

relationship = read_csv("RelationshipStatus.csv")
```

Joining datasets

A common action is to join and merge datasets with each others.

```
SomeJoinFunction(tibbleX, tibbleY, by = "ColumnName.x" = "ColumnName.y")
```

You need to know how the merging needs to work:

1. Do you want to keep all rows from both datasets? Or only one? Or only the ones common across both datasets?

2. Do you want to keep all columns from both datasets? Or only some?

3. Is there a unique identifier that it's common across both datasets? How should recognize that two rows refer to the same observation?

You cannot perform a merge without thinking about these questions first.

Answers to these questions depend *entirely* from your data questions.

Join functions

full_join() return all rows and all columns from both x and y . Where there are not matching values, returns NA for the one missing.

inner_join() return all rows from x where there are matching values in y, and all columns from x and y. If there are multiple matches between x and y, all combination of the matches are returned.

left_join() return all rows from x, and all columns from x and y. Rows in x with no match in y will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.

right_join() return all rows from y, and all columns from x and y. Rows in y with no match in x will have NA values in the new columns. If there are multiple matches between x and y, all combinations of the matches are returned.

Try on your own

1. You want to merge the immigration and education datasets. You want to keep all matching rows.

2. You want to merge the immigration and education datasets and keep as much information as possible from the largest dataset.

3. You want to merge the datasets education and income and keep as much information as possible from both datasets.

Solutions

Q1

```
data_1 =
inner_join(immigration, education)

nrow(data_1)
```

Q2

```
data_2 =
  left_join(immigration, education)
```

Q3

```
data_3 =
  full_join(education, income)
```

Just one try

Now try to merge the immigration and relationship datasets so that we keep all matching rows.

What happens?

```
data_4 =
  inner_join(immigration, relationship, by = c("id" = "id"))
```

By default, join functions match all columns with the same name across both datasets. If you don't want this default, you need to specify which columns should be considered:

- This could be fewer columns (e.g., in this case)
- This could be more columns - for instance, columns whose names do not match.

Other joint commands

Source: https://dplyr.tidyverse.org/reference/join.html

semi_join() return all rows from x where there are matching values in y, keeping just columns from x. A semi join differs from an inner join because an inner join will return one row of x for each matching row of y, where a semi join will never duplicate rows of x.

anti_join() return all rows from x where there are not matching values in y, keeping just columns from x

Save your data

Option 1: CSV

When you create a new dataset as you are cleaning your data, you might want to save it.

write.csv(YourData, "SavedNameDataSet.csv")

```
#Let's say we want to save the latest dataset that we created
write.csv(data4, "Data_0126.csv")
```

Save your data

Option 2: .RData files

.RData files are specific to R and can store **as many objects as you'd like within a single file**. Think about that. If you are conducting an analysis with 10 different dataframes and 5 hypothesis tests, you can save all of those objects in a single file called ExperimentResults.RData.

```
save(YourObject1, YourObject2, file = "SavedNameDataSet.RData")
```

```
save(data_4, data_3, file = "Data_0126.RData")
```

```
load(file = "Data_0126.RData")
```

Break

Lat topic

Unite and separate

unite is a function that allows you to combine the content of two or more columns. separate allows you to separate the content of one column into two or more columns.

```
unite(YourData, NewColumn, OldColumns, sep = " ")
```

```
separate(YourData, OldColumn, into = "NewColumn", sep = " ")
```

Note the argument sep. It's pretty common and it's always used to indicate a symbol by which a string should be separated (common ones: |, _ , a blank space, a common...)

Separate

Let's see a quick example

```
table3
```

We need to separate the column *rate* into two columns, cases and population.

```
table3_R =
  separate(table3, rate, into = c("cases", "population"))

# In this case, the default works but to be sure you can specify your 's
table3_R =
  separate(table3, rate, into = c("cases", "population"), sep = "/")

# The default output is a character column; if you want a better match,
table3_R =
  separate(table3, rate, into = c("cases", "population"), sep = "/", con
```

Unite

Let's now do the operation in reverse...

```
# See the default separator
table3_R2 =
unite(table3_R, rate, c(cases, population))

# You can change it
table3_R2 =
unite(table3_R, rate, c(cases, population), sep = "-" )
```

Next week

- Complete assignment #3
- We will start using dplyr to perform some of the operations that we are currently doing with $