

Data Managment and Automation Statistical Analysis with R:

Miriam Mota and Santi Pérez

Statistics and Bioinformatics Unit. Vall d'Hebron Institut de
Recerca

Outline: Data Exploration*

- Data managements with dplyr
- The pipe operator %>%
- Merging datasets

*Based on this presentation: *Data Managment, UCLA*.

Data Management packages

tidyverse: a collection of packages with tools for most aspects of data analysis, particularly strong in data import, management, and visualization. Packages within tidyverse:

- **dplyr** - subsetting, sorting, transforming variables, grouping
- **tidyr** - restructuring rows and columns
- **magrittr** - piping a chain of commands
- **stringr** - string variable manipulation

```
# install.packages("tidyverse", dependencies = TRUE)  
library(tidyverse)
```

tidyr

- Combines “**tidy**” + **R**, reflecting the **tidy data principles**:
 - One variable per column.
 - One observation per row.
 - One value per cell.

tidyr helps reshape and clean data structures into tidy format.

Example dataset

name	weight	height	age	disease
John	185	69	34.5	TRUE
Emily	150	62	55.6	FALSE
Mary	120	65	21.1	TRUE
Dan	225	72	51.1	TRUE

Each row is an
observation

Two-dimensional
Heterogeneous
Rectangular

Each column vector
is a variable

Section 1

Data managements with dplyr

dplyr

`dplyr` is derived from “**data plier**”, like a tool for handling data. -
Vowels were removed: `data plier` → `dplyr`. - Introduces
intuitive verbs: `filter()`, `mutate()`, `select()`, `summarise()`,
etc.

Makes `data.frames` manipulation readable and expressive.

The dplyr package

The **dplyr** package provides tools for some of the most common data management tasks. Its primary functions are “verbs” to help you think about what you need to do to your dataset:

- **filter()**: select rows according to conditions
- **select()**: select columns (you can rename as you select)
- **arrange()**: sort rows
- **mutate()**: add new columns

The **dplyr** package is automatically loaded with **library(tidyverse)**.

Selecting rows with filter

The dplyr function `filter()` provides a cleaner syntax for subsetting datasets. Conditions separated by `,` are joined by `&` (logical AND).

```
require(readxl)
diab <- read_excel("datasets/diabetes.xls")
diab_filt <- filter(diab, tabac == "No fumador", edat >= 50)
head(diab_filt, n = 4)
```

```
## # A tibble: 4 x 11
```

##	numpacie	mort	tempsviu	edat	bmi	edatdiag	tabac	sbp	dbp
##	<dbl>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<chr>	<dbl>	<dbl>
## 1	7	Vivo	12.4	50	36.5	48	No fuma~	140	86
## 2	12	Vivo	10.8	54	42.9	43	No fuma~	128	74
## 3	56	Vivo	10.2	64	30.1	58	No fuma~	138	76
## 4	59	Muerto	6.7	62	34.6	58	No fuma~	138	78

Selecting columns with select

Use dplyr function `select()` to keep only the variables you need.

```
diab_small <- select(diab, mort, edat, tabac, sbp)
head(diab_small, n = 4)
```

```
## # A tibble: 4 x 4
##   mort    edat tabac      sbp
##   <chr> <dbl> <chr>    <dbl>
## 1 Vivo     44 No fumador  132
## 2 Vivo     49 Fumador    130
## 3 Vivo     49 Fumador    108
## 4 Vivo     47 No fumador  128
```

Sorting rows with arrange

Sort the order of rows by variable values using **arrange()** from dplyr.

Be default, ascending order will be used. Surround a sorting variable with **desc()** to sort by descending order instead.

```
# sort, with males before 'vivo', then by age, youngest first  
diab_sort <- arrange(diab, desc(mort), edat)  
head(diab_sort, n = 4)
```

```
## # A tibble: 4 x 11  
##   numpacie mort   tempsviu   edat   bmi edatdiag tabac      sbp   dbp  
##   <dbl> <chr>     <dbl> <dbl> <dbl>   <dbl> <chr>     <dbl> <dbl>  
## 1    114 Vivo      14.8   31  38.8     29 Ex fumad~  136    76  
## 2    110 Vivo      15.4   33  34      33 Fumador    120    78  
## 3     27 Vivo       8.6   34  33.9     30 Fumador    124    66  
## 4     20 Vivo      14.1   35  47      33 Ex fumad~  134    78
```

R Logical operators and functions

Here are some operators and functions to help with selection:

- **==**: equality
- **>**, **>=**: greater than, greater than or equal to
- **!**: not
- **&**: AND
- **|**: OR
- **%in%**: matches any of (2 %in% c(1,2,3) = TRUE)
- **is.na()**: equality to NA
- **near()**: checking for equality for floating point (decimal) numbers, has a built-in tolerance

Transforming variables into new variables

The function **mutate()** allows us to transform many variables in one step without having to respecify the data frame name over and over.

Useful R functions for transforming:

- **log()**: logarithm
- **min_rank()**: rank values
- **cut()**: cut a continuous variable into intervals with new integer value signifying into which interval original value falls
- **scale()**: standardizes variable (subtracts mean and divides by standard deviation)
- **cumsum()**: cumulative sum
- **rowMeans()**, **rowSums()**: means and sums of several columns

Example: mutate()

create age category variable, and highbmi binary variable

```
diab_mut <- mutate(diab,  
  edatcat = cut(edat, breaks = c(0,40,50,60,70,120)),  
  highbmi = bmi > mean(bmi))  
tail(diab_mut, n = 4)
```

```
## # A tibble: 4 x 13  
##   numpacie mort   tempsviu  edat  bmi edatdiag tabac    sbp  dbp ecg  chd  
##   <dbl> <chr>    <dbl> <dbl> <dbl>    <dbl> <chr>    <dbl> <dbl> <chr> <chr>  
## 1    146 Vivo      11    40  34      38 Fumador   132   76 Norm~ No  
## 2    147 Vivo      7.3   61 19.9     37 No fuma~  120   60 Fron~ Si  
## 3    148 Muerto  10.6  62 30.6     49 No fuma~  160   86 Fron~ Si  
## 4    149 Vivo      10.5  49 30.8     47 Ex fuma~  146   86 Norm~ No  
## # i 2 more variables: edatcat <fct>, highbmi <lgl>
```

```
table(diab_mut$edatcat, diab_mut$highbmi)
```

EXERCISE

- ❶ Find all individual that:
 - 1.1 Had a sbp higher than 160 (**filter()**)
 - 1.2 Had a sbp higher than 160 or tabac was 'Fumador'
- ❷ What happens if you include the name of a variable multiple times in a **select()** call?
- ❸ Sort individual to find the most 'tempsviu'. (**arrange()**)

Section 2

The pipe operator %>%

magrittr

- The package name `magrittr` is a tribute to the surrealist artist **René Magritte** ("This is not a pipe").
- Just as the painting represents a pipe, the %>% operator represents a **flow of operations**.



"We are not seeing a pipe, but a representation of one."

The pipe operator %>%

A data management task may involve many steps to reach the final desired dataset. Often, during intermediate steps, datasets are generated that we don't care about or plan on keeping. For these multi-step tasks, the pipe operator provides a useful, time-saving and code-saving shorthand.

Naming datasets takes time to think about and clutters code. Piping makes your code more readable by focusing on the functions used rather than the name of datasets.

Using the pipe operator

The pipe operator “pipes” the dataset on the left of the %>% operator to the function on the right of the operator.

The code `x %>% f(y)` translates to `f(x,y)`, that is, `x` is treated by default as the first argument of `f()`. If the function returns a data frame, we can then pipe this data frame into another function. Thus `x %>% f(y) %>% g(z)` translates to `g(f(x,y), z)`.

Examples of using the pipe operator

As a first example, perhaps we want to create a dataset of just Vivo under 40, with only the age and pain variables selected. We could do this in 2 steps, like so:

```
diab40 <- filter(diab, mort == "Vivo" & edat < 40)
diab40_small <- select(diab40, edat, dbp)
head(diab40_small, n = 4)
```

```
## # A tibble: 4 x 2
##   edat    dbp
##   <dbl> <dbl>
## 1     36    88
## 2     38    98
## 3     35    78
## 4     34    66
```

Examples of using the pipe operator

While that works fine, the intermediate dataset f40 is not of interest and is cluttering up memory and the workspace unnecessarily.

We could use %>% instead:

```
diab40_small <- diab %>%  
  filter(mort == "Vivo" & edat < 40) %>%  
  select(edat, dbp)  
head(diab40_small, n = 4)
```

```
## # A tibble: 4 x 2  
##   edat    dbp  
##   <dbl> <dbl>  
## 1     36    88  
## 2     38    98  
## 3     35    78  
## 4     34    66
```

EXERCISE

Replicate the last exercise using 'pipes'

```
df <- filter(diab, sbp > 160 | tabac == "Fumador")  
  
dfs <- select(df, tempsviu ,bmi,sbp,sbp)  
  
dfsa <- arrange(dfs, desc(tempsviu))
```

Section 3

Example

Example

This dataset contains patient information from a hospital survey or registry.

It was provided by a researcher and needs a lot of cleaning.

GOAL

- Transform this chaotic dataset into a clean, structured, and analyzable format using R:
- `clean_names()` (from `janitor`)
- `filter()`, `select()`, `mutate()`, `arrange()` (from `dplyr`)

Horrible data base

```
## ID.Paciente Edad_en_años_.. Fecha_de_Ingreso__hospital. Sexo_.M.F.
## 1 ID001 34 2020/01/15 Masculino
## 2 ID002 29 na femenino
## 3 ID003 Na 2015/02/28 FEM
## 4 ID003 29 2018/01/20 femen
## 5 TOTAL N/A
## 6 ID004 45 2020/01/20 Masculino
## Grupo.de.Intervención___.o.no. Variable_que_no_se_usa__x9_
## 1 Control NA
## 2 intervencion NA
## 3 Intervencion NA
## 4 ctrl NA
## 5 TOTAL NA
## 6 Intervención NA
## Notas...adicionales__
## 1 NA
## 2 Buen estado
## 3
```

Horrors in the dataset:

- Long, unclear column names with strange characters.
- Extra rows like repeated headers or summary lines.
- Incorrectly typed variables (numbers stored as text, inconsistent date formats).
- Categories with spelling mistakes, inconsistent capitalization, or extra spaces.
- Missing and duplicate data.
- Useless columns.

Load Packages

```
require(pacman)
```

```
## Cargando paquete requerido: pacman
```

```
p_load(dplyr, janitor)
```

Clean names

Use `clean_names()` to fix the column names

```
names(horrible_base)[1:3]
```

```
## [1] "ID.Paciente"          "Edad_en_años_..."  
## [3] "Fecha_de_Ingreso__hospital."
```

```
horrible_base_clean <- horrible_base %>%  
  janitor::clean_names()  
names(horrible_base_clean)[1:3]
```

```
## [1] "id_paciente"          "edad_en_anos"  
## [3] "fecha_de_ingreso_hospital"
```

Filter rows

Use `filter()` to remove unwanted or extra rows.

You can remove:

- Rows with “TOTAL” in any column.
- Full duplicate rows.

```
dim(horrible_base_clean)
```

```
## [1] 6 7
```

```
horrible_base_clean <- horrible_base_clean %>%  
  filter(!grepl("TOTAL", id_paciente)) %>%  
  filter(!duplicated(.))  
dim(horrible_base_clean)
```

```
## [1] 5 7
```

Select columns

Use select() to keep only useful columns

```
dim(horrible_base_clean)
```

```
## [1] 5 7
```

```
horrible_base_clean <- horrible_base_clean %>%  
  select(id_paciente,  
         edad_en_anos,  
         fecha_de_ingreso_hospital,  
         sexo_m_f,  
         grupo_de_intervencion_o_no,  
         notas_adicionales)  
dim(horrible_base_clean)
```

```
## [1] 5 6
```

Mutate variables

Use `mutate()` to fix data types and clean up categories

- Convert age to numeric.
- Parse various date formats.
- Standardize text values in categorical variables.

Mutate variables. Numerical

Convert age to numeric.

```
head(horrible_base_clean$edad_en_anos)
```

```
## [1] "34" "29" "Na" "29" "45"
```

```
horrible_base_clean <- horrible_base_clean %>%  
  mutate(  
    edad_en_anos = as.numeric(edad_en_anos) )
```

```
head(horrible_base_clean$edad_en_anos)
```

```
## [1] 34 29 NA 29 45
```


Mutate variables. Categorical

Standardize text values in categorical variables.

```
head(horrible_base_clean$sexo_m_f)
```

```
## [1] "Masculino" "femenino"  "FEM"       "femen"     "Masculino"
```

```
horrible_base_clean <- horrible_base_clean %>%
```

```
  mutate(
```

```
    sexo_m_f = case_when(
```

```
      grepl("mascul", tolower(sexo_m_f)) ~ "Male",
```

```
      grepl("fem", tolower(sexo_m_f)) ~ "Female",
```

```
      TRUE ~ NA_character_
```

```
    ),
```

```
    grupo_de_intervencion_o_no = case_when(
```

```
      tolower(grupo_de_intervencion_o_no) %in% c("intervencion", "inter
```

```
      tolower(grupo_de_intervencion_o_no) %in% c("control", "ctrl") ~ "
```

```
      TRUE ~ NA_character_
```

```
    )
```

```
)
```

Mutate variables. Date

Parse various date formats.

```
head(horrible_base_clean$fecha_de_ingreso_hospital)
```

```
## [1] "2020/01/15" "na" "2015/02/28" "2018/01/20" "2020/01/20"
```

```
horrible_base_clean <- horrible_base_clean %>%  
  mutate(  
    fecha_de_ingreso_hospital = as.Date(fecha_de_ingreso_hospital, c("%  
  )
```

```
head(horrible_base_clean$fecha_de_ingreso_hospital)
```

```
## [1] "2020-01-15" NA "2015-02-28" "2018-01-20" "2020-01-20"
```

Arrange data base

Use `arrange()` to sort the data For example, by admission date and age:

```
horrible_base_clean <- horrible_base_clean %>%  
  arrange(fecha_de_ingreso_hospital, edad_en_anos)  
head(horrible_base_clean %>% select(id_paciente, fecha_de_ingreso_hospit
```

##	id_paciente	fecha_de_ingreso_hospital	edad_en_anos
## 1	ID003	2015-02-28	NA
## 2	ID003	2018-01-20	29
## 3	ID001	2020-01-15	34
## 4	ID004	2020-01-20	45
## 5	ID002	<NA>	29

Final result: A cleaner dataset

Now you have a much cleaner dataset with:

- Clear and consistent column names.
- Correct data types.
- Unwanted rows removed.
- Clean and standardized categories.
- Would you like me to wrap this all into one clean code block for easy reuse?

Section 4

Merging datasets

Merging datasets

Appending adds more rows of observations, whereas merging adds more columns of variables. Datasets to be merged should be matched on some id variable(s).

patid	docid	co2
101	1-1	2.36
102	1-2	1.11
103	1-2	0.45



docid	d_sex
1-1	male
1-2	female



patid	docid	co2	d_sex
101	1-1	2.36	male
102	1-2	1.11	female
103	1-2	0.45	female

We can merge observations if they share a **matching variable**

Data example

```
band_members
```

```
## # A tibble: 3 x 2
##   name  band
##   <chr> <chr>
## 1 Mick  Stones
## 2 John  Beatles
## 3 Paul  Beatles
```

```
band_instruments
```

```
## # A tibble: 3 x 2
##   name  plays
##   <chr> <chr>
## 1 John  guitar
## 2 Paul  bass
## 3 Keith guitar
```

Append row bind_rows()

```
bind_rows(band_members, band_instruments)
```

```
## # A tibble: 6 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles <NA>
## 3 Paul  Beatles <NA>
## 4 John  <NA>    guitar
## 5 Paul  <NA>    bass
## 6 Keith <NA>    guitar
```


Append columns bind_cols()

!!!!!!!

```
bind_cols(band_members, band_instruments)
```

```
## New names:
```

```
## * 'name' -> 'name...1'
```

```
## * 'name' -> 'name...3'
```

```
## # A tibble: 3 x 4
```

```
##   name...1 band   name...3 plays
```

```
##   <chr>    <chr>  <chr>   <chr>
```

```
## 1 Mick    Stones  John    guitar
```

```
## 2 John    Beatles Paul    bass
```

```
## 3 Paul    Beatles Keith   guitar
```

!!!!!!!

Merging datasets with dplyr joins

The **dplyr** “join” functions perform such merges and will use any same-named variables between the datasets as the id variables by default. Use the `by=` argument to specify specific matching id variables.

These joins all return a table with all columns from `x` and `y`, but differ in how they deal with mismatched rows:

- **inner_join(x, y)**: returns all rows from `x` where there is a matching value in `y` (returns only matching rows).
- **left_join(x, y)**: returns all rows from `x`, unmatched rows in `x` will have NA in the columns from `y`. Unmatched rows in `y` not returned.
- **full_join(x, y)**: returns all rows from `x` and from `y`; unmatched rows in either will have NA in new columns

Mutating joins

inner_join(x, y): returns all rows from x where there is a matching value in y (returns only matching rows).

```
band_members %>%  
  inner_join(band_instruments, by = "name")
```

```
## # A tibble: 2 x 3  
##   name  band    plays  
##   <chr> <chr>   <chr>  
## 1 John  Beatles guitar  
## 2 Paul  Beatles  bass
```

Mutating joins

Other joins : left_join, right_join, full_join

```
band_members %>%  
  left_join(band_instruments)
```

```
## Joining with 'by = join_by(name)'
```

```
## # A tibble: 3 x 3  
##   name  band    plays  
##   <chr> <chr>   <chr>  
## 1 Mick  Stones  <NA>  
## 2 John  Beatles guitar  
## 3 Paul  Beatles bass
```

EXERCISE

What happens if you run these lines?

```
band_members %>%  
  right_join(band_instruments)
```

```
band_members %>%  
  full_join(band_instruments)
```