

R for Data Science (III): Data Managment

Alex Sanchez, Miriam Mota, Ricardo Gonzalo and Mireia Ferrer

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

Outline: Data Exploration*

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

*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)
```

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

Data managements with dplyr

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 ecg
##   <dbl> <chr>    <dbl> <dbl> <dbl>   <dbl> <chr> <dbl> <dbl> <chr>
## 1      7 Vivo      12.4   50  36.5     48 No f~  140   86 Fro
## 2     12 Vivo      10.8   54  42.9     43 No f~  128   74 Nor
## 3     56 Vivo      10.2   64  30.1     58 No f~  138   76 Fro
## 4     59 Muer~      6.7   62  34.6     58 No f~  138   78 Ano
## # ... with 1 more variable: chd <chr>
```

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 ecg
##   <dbl> <chr>    <dbl> <dbl> <dbl>    <dbl> <chr> <dbl> <dbl> <chr>
## 1    114 Vivo     14.8   31  38.8      29 Ex f~  136   76 Nor
## 2    110 Vivo     15.4   33  34        33 Fuma~  120   78 Nor
## 3     27 Vivo      8.6   34  33.9      30 Fuma~  124   66 Nor
## 4     20 Vivo     14.1   35  47        33 Ex f~  134   78 Nor
## # ... with 1 more variable: chd <chr>
```

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(40,50,60,70,120)),  
  highbmi = bmi > mean(bmi))  
head(diab_mut, n = 4)
```

```
## # A tibble: 4 x 13  
##   numpacie mort   tempsviu  edat   bmi edatdiag tabac   sbp   dbp ecg  
##   <dbl> <chr>   <dbl> <dbl> <dbl>   <dbl> <chr> <dbl> <dbl> <chr>  
## 1     1 Vivo    12.4  44  34.2     41 No f~  132   96 Norm~  
## 2     2 Vivo    12.4  49  32.6     48 Fuma~  130   72 Norm~  
## 3     3 Vivo     9.6  49  22      35 Fuma~  108   58 Norm~  
## 4     4 Vivo     7.2  47  37.9     45 No f~  128   76 Fron~  
## # ... with 3 more variables: chd <chr>, 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()**)

The pipe operator %>%

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 these lines using 'pipes'

```
filter(diab, sbp > 160 | tabac == "Fumador")  
  
select(diab, edat, bmi, sbp, sbp)  
  
arrange(diab, desc(tempsviu))
```

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)
```

```
## # A tibble: 3 x 4  
##   name band   name1 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)
```

```
## # 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)
```

Spreading and gathering

Case 1: Use `gather()` to create a variable out of column headings and restructure the dataset

To use `gather()`, we select a set of columns for reshaping:

- the column headings are reshaped into column variable
- the values in the columns are gathered and stacked into a single column
- This process is also known as “reshaping long”.

Arguments to `gather()`:

- the dataset
- **key=**: name of the new column that will hold the values of the selected column headings
- **value=**: the name of the new column that will hold the stacked values of the selected columns

Use gather() example

```
gather(key="year", value="grad", -id)
```

id	2015	2016	2017
biology	207	211	259
math	96	75	99
physics	112	126	125



id	year	grad
biology	2015	207
math	2015	96
physics	2015	112
biology	2016	211
math	2016	75
physics	2016	126
biology	2017	259
math	2017	99
physics	2017	125

For selected columns,
gather() forms a
new variable from the
column headings and
stacks the values in a
new variable

Use gather() example

```
dept <- read_csv("datasets/dept1.csv")
dept_by_year <- dept %>%
  gather(key = "year", value = "grad", -id)
dept_by_year
```

```
## # A tibble: 9 x 3
##   id      year  grad
##   <chr>  <chr> <int>
## 1 biology 2015   207
## 2 math    2015    96
## 3 physics 2015   112
## 4 biology 2016   211
## 5 math    2016    75
## 6 physics 2016   126
## 7 biology 2017   259
## 8 math    2017    99
## 9 physics 2017   125
```


Case 2: Multiple variables in one column, spread()

Columns should contain values that represent one variable, but we often encounter datasets where multiple variables are stored in the same column.

Let's take a look at a dataset of worms, who have had their age, length, and weight measured, but all stored in one column.

```
worms <- read_csv("datasets/worms.csv")  
head(worms, n = 6)
```

```
## # A tibble: 6 x 3  
##   worm feature measure  
##   <int> <chr>      <dbl>  
## 1     1 age         5  
## 2     1 length      3.2  
## 3     1 weight      4.1  
## 4     2 age         4  
## 5     2 length      2.6  
## 6     2 weight      3.5
```

spread() a single column into multiple columns

The spread() function serves as the complement to gather(), spreading key-value pairs (feature-measurement pairs in our example) across columns. This process is sometimes known as “reshaping wide”.

```
by_worm <- worms %>%  
  spread(key = feature, value = measure)  
by_worm
```

```
## # A tibble: 3 x 4  
##   worm   age length weight  
##   <int> <dbl> <dbl>   <dbl>  
## 1     1     5   3.2     4.1  
## 2     2     4   2.6     3.5  
## 3     3     5   3.6     5.5
```

Data managements with dplyr
The pipe operator %>%
Merging datasets
Spreading and gathering
Your turn

Your turn