Introduction to the tidyverse: `dplyr`, `tidyr` and `purrr`

Aitor Ameztegui, Víctor Granda

February 2019

Introduction

Thi is a `RMarkdown` document generated to illustrate the exercices suggested as part of the Workshop "Introduction to the tidyverse", that will be celebrated in Barcelona within the 1st Meeting of the Iberian Ecological Society and the XIV AEET Meeting. All the code and the data needed to produce this document can be found in GitHub (https://github.com/ameztegui/dplyr_workshop). For any doubt about the exercises, you can contact Aitor Ameztegui (ameztegui@gmail.com) or V?ctor Granda (ameztegui@gmail.com).

The packages`tidyr`, `dplyr`, and `purrr` are part of a set of packages known as the `tidyverse`, created by Hadley Wickham, Chief Scientist in RStudio. The `tidyverse` was created to easen data analysis and data management. It consists on packages to import and read data, to organize and modify them, to analyze and model them and to visualize them. In this seminar we will focus on `tidyr`, conceived to help organize data, and `dplyr`, that focuses on data transformation. We must firs install these packages (if we haven't done it yet). To install all the packages from the `tidyverse` at once we need to write `install.packages("tidyverse")`. Then we need to load the packages by typing:

```
library(tidyverse)
```

We also need to load the dataset we will use for this workshop, we can download them from GitHub.

```
load("../data/data_workshop.Rdata")

trees <- tbl_df(trees)
plots <- tbl_df(plots)
species <- tbl_df(species)
coordinates <- tbl_df(coordinates)
```

Through the function `tbl_df` we will convert normal data frames into *tibbles*. A *tibble* is just a data frame with some particularities: for example, they only print the first 10 rows by default (instead of the whole data frame), and printing them provides information on all the variables and their class. Besides that, we can treat tibbles as normal data frames, because they behave like them at all effects.

In these exercises we will use four data frames with information from the 2nd and 3rd Spanish National Forest Inventory (IFN2 e IFN3) in Catalonia. The data frames are:

- **plots** [11,858 x 15]: all the IFN3 plots in Catalonia, with info about the date and time of measurement, soil texture and soil pH, total canopy cover and tree canopy cover, etc.
- **trees** [111,756 x 12]: contains all the adult trees (diam $>$ 7.5 cm) measured both in IFN2 and IFN3. Contains info about the plot, the species, diameter class, diameter measured at IFN2 and IFN3...
- **species** [14,778 x 15]: contains the number of trees/ha per species and diameter class.
- **coordinates** [11,858 x 6]: contains the X & Y coordinates of each IFN3 plot.

First thing to do is to have a look at the data, to get familiar with the data they contain. We will use the function `glimpse` for that.

```
glimpse(plots)
glimpse(trees)
glimpse(species)
glimpse(coordinates)
```

dplyr: transforming data frames

dplyr can be used to transform our data frames in the way we need it in each case: we can create bew variables, select those of interest, execute filters, etc. The dplyr package contains 5 main verbs:

- filter selects rows based on a given set of conditions
- select select columns based on their name
- arrange sort the data frame based on one or several variables
- mutate create new variables
- summarise create new variables that summarize values of an existuing variable (mean, sum, etc.)

All of them have a similar structure: the first argument in the function is the data frame to which it will be applied, and the rest of arguments specify what to do with this data frame, depending on the verbwe are using.

filter

filter selects those rows of a data frame that accomplish a certain criterion. The first argument is the data frame, and the rest are the criteria, that can be specified in chain, separated by commas.

###Exercice 1

To practice with filter let's try to find those plots of IFN that:

- 1.1 Are located in Barcelona (08) or Girona (17). We have two options:

```
# Option 1
    filter (plots, Provincia =="08" | Provincia =="17")
# Option 2
    filter (plots, Provincia %in% c("08", "17"))
```

We see that both options produce exactly the same result. But the next option wouldn't be valid, since we need to specify explicitly the variable every time we add a new condition:

```
filter(plots, Provincia =="08" | "17")
```

- 1.2 plots that were measured completely in January 2001

To do this we need to find the plots for which the completion date is later than 31 December 2000 and earlier than 1 February 2001. We can do this in two ways: the first one is to use the & operator to indicate we want to get the rows that meet both criteria. The second options would be simply to concatenate both criteria with a comma, since filter assumes all of them must be met.

```
    # Option 1
    filter (plots, FechaFin < "2001-02-01" & FechaFin > "2000-12-31")

    # Option 2
    filter (plots, FechaFin < "2001-02-01", FechaFin > "2000-12-31")
```

- 1.3 Those plots that took more than 2 hours to be measured (7200 seg)

```
    filter(plots, (HoraFin - HoraIni) >7200)
```

As we see, we can do operations within filter conditions. In this case, we want that the difference between EndDate and StartDate be < 7200 s (2 hours).

select

`select` allows to retain only some columns, based on their name. To help us find the columns, there are some specific functions such as `starts_with` or `contains`, that only work within `select`. We can see the list of special functions by typing `help("select")`

###Exercise 2 To practice with `select` let's try to find 4 different ways of selecting the variables that specify the starting and ending date of measurement of the plots (FechaIni y FechaFin)

- For example, we could specifiy the name of the columns we want to keep in an explicit way

```
select(plots, FechaIni, FechaFin)
```

- We can also specify them as a range, so that all columns between the two indicated will be selected

```
select(plots, FechaIni:FechaFin)
```

- Or we could select all the columns that contain the text 'fecha'. In this case, since we are not interested in the pH measurement date, we can decide to delete it from our selection:

```
select(plots, contains ("Fecha"),-FechaPh)
```

- At last, we could also select all the variables that start with 'fecha' (in this case we also need to eliminate FechaPh):

```
select(plots, starts_with("Fecha"), -FechaPh)
```

arrange

`arrange` sorts the data frame based on the values of ine or more variables (columns). The first argument will be, as usual, the data frame we want to sort, and then we must specify the variables that determine the ordering. If we specify more than 1 variable, the succesive variables will be used to decide order when there are ties (i.e. secondary sorting variables). We can also use 'desc(x)' to sort in decreasing order. Let's try with a few exercises:

###Exercise 3

- Ex.3.1 Sort the plots by measurement date and time

```
arrange(plots, FechaFin, HoraFin)
```

- Ex. 3.2 Which plots were started to be measured later in the day?

```
arrange(plots,desc(HoraIni))
```

- Ex. 3.3 Which took longer to be measured?

```
arrange(plots, desc(HoraFin-HoraIni))
```

We see that, as it happens with `filter`, we can also sort data frames based on the result of an arithmetic operation.

mutate

`mutate` allows us to create new variables with a certain value or as combination of existing variables. We just need to specify the data frame, and indicate the new variables name and its value. Let's see some examples:

###Exercise 4 Let's create two new variables:

- Ex.4.1 A variable with individual tree growth (in cm) between IFN2 and IFN3.

```
trees <- mutate (trees, growth= DiamIf3 - DiamIf2)
```

- Ex.4.2 Create two new variables with the basal area per hectare that each tree represents, both in IFN2 and IFN3. Which species was the fastest growing tree in basal area?

```
trees <- mutate(trees, BAIf2= (((pi/4)*(DiamIf2/100)^2)*N),
                BAIf3= (((pi/4)*(DiamIf3/100)^2)*N),
                BA_growth = BAIf3 - BAIf2)

arrange(trees, desc(BA_growth))
```

As we see, we can calculate new variables based on the variables we just created. Also, we can combine `mutate` and `arrange` to know which is the fastest growing tree.

summarise

`summarise` allows us to make calculations with the variables in the data frame, but using *summary functions*, that transform the variability in a given variable into a single value. Functions such as `sum`, `mean`, `max`,`IQR`, etc. are examples of summary functions. However, this function by itself often lacks any interest, cause it would reduce all the data frame to a single value. It is commonly used together with `group_by`, that classifies the data frame in groups based on a categorical variable.

To use `group_by` we just need to indicate the data frame and the variable we want to group it by. To be more efficient, `dplyr` does not create a copy of the data frame, but it creates a hidden variable that indexes the groups, so that when we ask it to perform operations by group, it know to which group belongs each observation.

In the case of our data frame `trees`, there are several groups that could be of interest el caso de nuestra base de datos de pies trees, hay varios grupos que pueden tener inter?s:

```
# Por provincia
by_province <- group_by (trees, Provincia)

# Por parcela
by_plot <- group_by (trees, Codi)

# Por especie
by_species <- group_by (trees, Especie)

#Por clase diam?trica
by_CD <- group_by (trees, CD)

#Por parcela y especie
by_plot_species <- group_by (trees, Codi, Especie)
```

We can see, by typing `glimpse(by_plot)` that the resulting data frame is not different at all from the original, at least apparently. However, if we type `class(by_plot)` we see it has now a new class `grouped_df`.

###Exercise 5 What statistics could be of interest to characterize the diameter values of each plot? We can, for example, calculate te mean, minimum and maximum value, percentile 0.9 and interquartile range for each plot. We can also compute the number of trees measured in each plot and the number of different species, using the functions `n()` y `n_distinct(x)`. In this case, the resulting data frame will have less rows, one per plot, and will only contain the new variables created.

```
summarise(by_plot,
          media = mean(DiamIf3),
          min = min (DiamIf3),
          max = max(DiamIf3),
          q90 = quantile(DiamIf3, 0.9),
          IQ = IQR(DiamIf3),
```

```
          n =n(),
          sps = n_distinct(Especie) )
```

```
## # A tibble: 7,713 x 8
##    Codi   media   min   max   q90    IQ     n   sps
##    <fct>  <dbl> <dbl> <dbl> <dbl> <dbl> <int> <int>
##  1 080001  26.3  13.4  38    34.4  4.15    15     1
##  2 080002  35.2  24.8  44.4  43.0 10.2     13     3
##  3 080003  32.0  14.2  51    46.1 12.2      7     2
##  4 080004  24.3  16.8  31.7  30.2  7.42     2     1
##  5 080005  28.4  16.2  59.8  39.7 15.3     12     3
##  6 080006  35.9  14    55.9  52.3 17.9     23     2
##  7 080007  30.8  15.2  63.6  49.4 12.3     35     2
##  8 080008  16.0   9    21.4  17.8  2.05    11     1
##  9 080009  16.7   9    36.0  24    3.88    16     2
## 10 080010  31.6   9.2  95.5  61.5 18.2     13     3
## # ... with 7,703 more rows
```

Pipelines (%>%)

We will often use several `dplyr` verbs together, creating nested functions. However, when we need to perform several operations, these nested functions can easily get complex and difficult to understand. For example, by having a look at this code, would you be able to say what it will do?

```
diam_medio_especie <- filter(
    summarise(
        group_by(
            filter(
                trees,
                !is.na(DiamIf3)
            ),
        Codi, Especie
        ),
    diam = mean (DiamIf3),
    n = n()
    ),
n > 5)
```

The code gets those observations, from the data frame `trees`, that have a value of diameter (`!is.na(DiamIf3)`), it then groups them by plot and species (`group_by(Codi, Especie)`), calculates for each combination the mean diameter (`diam = mean (DiamIf3)`), and the number of trees per plot (`n = n()`), and finally selects only those cases in which there are at least 5 trees (`filter (n>5)`).

Although this syntaxis is not operationally complex, it is hard to understand. Often a solution is to save each step as a different data frame, but this is an important source of errors.

We can however simplify this code using the *pipe* operator (`%>%`) from the `magrittr` package, which is installed and loaded with `tidyr` and `dplyr`. When we use `%>%`, the result of the left side is processed by the righ side function as first argument. In the case of `dplyr` and `tidyr`, since the first argument is always a data frame, `%>%` makes that a function be applied to the data frame resulting from the previous function. Thus, we can express `filter (df, color =="blue)` as `df %>% filter(color == "blue")`. This allows to concatenate several functions in a logical and understandable way, so that the operator `%>%` could be read as *then*. Let's see how this would be in the previous function

```
diam_medio_especie <- trees %>%              # take the df 'trees' and THEN
filter(!is.na(DiamIf3)) %>%                  # eliminate NA values and THEN
group_by(Codi, Especie) %>%                  # group y plot and species and THEN
summarise(diam=mean(DiamIf3), n = n()) %>%   # calculate mean and number of trees and THEN
filter(n > 5)                                # filter those with n> 5
```

###Exercise 6 Let's do some exercises. Using the pipe operator, let's create pipelines to solve the next exercises:

- Ex.6.1 Which plots have the greatest average growth between IFN2 and IFN3?

We first define the data frame we will work with. *THEN* (`%>%`) we create a new variable with the growth of each tree, *THEN* we group by plot, *THEN* we calculate, for each plot, the mean growth, and *THEN* we arrange the results in decreasing order. The resulting code would be:

```
trees %>%
    mutate(growth=DiamIf3-DiamIf2) %>%
    group_by(Codi) %>%
    summarise(av_growth=mean(growth), n=n()) %>%
    arrange(desc(av_growth))
```

```
## # A tibble: 7,713 x 3
```

```
##    Codi   av_growth     n
##    <fct>      <dbl> <int>
##  1 171089      23.1     3
##  2 170819      21.6     1
##  3 172607      17.6     6
##  4 172216      17.4     6
##  5 172690      16.0    17
##  6 171682      15.4     6
##  7 083267      15.3     1
##  8 431363      15.1     4
##  9 171664      14.8     5
## 10 171976      14.4     1
## # ... with 7,703 more rows
```

- Ex.6.2 Which is the plot with highest species richness?

First, we define thedata frame (`trees`), *THEN* we group by Code, *THEN* we determine the number of species per plot and *THEN* we arrange in decreasing order:

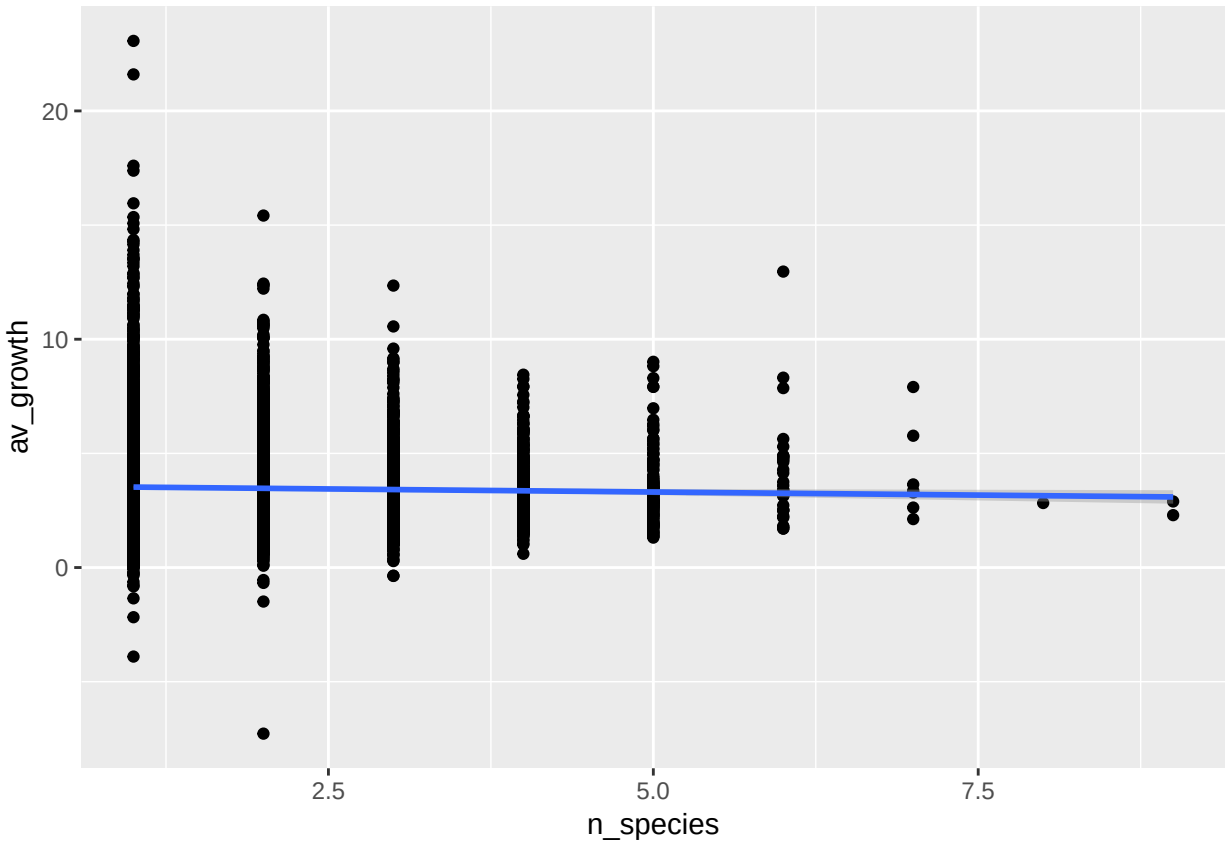
```
trees %>%
    group_by(Codi) %>%
    summarise (n_species=n_distinct(Especie)) %>%
    arrange(desc(n_species))
```

```
## # A tibble: 7,713 x 2
##    Codi   n_species
##    <fct>      <int>
##  1 170195         9
##  2 171036         9
##  3 170218         8
##  4 170121         7
##  5 170596         7
##  6 170635         7
##  7 170799         7
##  8 171398         7
##  9 171481         7
## 10 172650         7
## # ... with 7,703 more rows
```

- Ex.6.3 Are both variables (species richness and average growth) related?

First, we would need to indicate the data frame we will work with, *THEN* we will group by plot, *THEN* we will calculate the variables. To see the relationship between both variables, we will create a plot with `ggplot2`, just to show how all the packages in the tidyverse relate to each other. The aim of this workshop is not to learn `ggplot2`, so we won't go into further details. To know more about ggplot you can visit this website: http://ggplot2.org/.

```
trees %>%
    mutate(growth=DiamIf3-DiamIf2) %>%
    group_by(Codi) %>%
        summarise (n_species=n_distinct(Especie),
                   av_growth=mean(growth)) %>%
    ggplot(aes(n_species, av_growth)) +
    geom_point() +
    geom_smooth(method = "lm")
```



Here we see one of the advantages of the `tidyverse`, the fact that all the packages and functions can communicate one with another. In this way, we just created a plot without the need of creating intermediate objects or data frames, starting directly from the raw data frame, and chianing orders in a logical and intuitive way.

Grouped mutate/grouped filter

Most of the times we use `group_by`, we will do it with the *summary functions*, that is, functions that take n values as input, and give back 1 value as output. Examples of *summary functions* are `mean()`, `sd()`, `min()`, `sum()`, etc.

However, some times we will need to do some operation by group, but we will need to produce one output per input, that is n inputs –> n outputs. This can be done using `mutate` or `filter` in combination with `group_by`.

###Exercise 7

Taking this into account, let's try to:

- Ex.7.1 Identify those trees that grow much faster than the average of the plot

```
trees %>%
    mutate(growth=DiamIf3-DiamIf2) %>%
    group_by(Codi) %>%
    mutate(plot_mean= mean(growth),
           des = (growth - plot_mean)) %>%
    arrange(desc(des))
```

In the previous code we see we first calculate the growth of each tree, and after grouping by plot, we calculate

a new variable, where the plot average is substracted from the growth of each tree, and the result is divided by the standard deviation of the plot. We calculate in this way the standardized growth of each tree with respect to the plot, making it easy to identify those trees that grow suspiciously more than the average for their plot.

- Ex. 7.2 Identify those plots where a species grows much more than the average for the species

```
trees %>%
    mutate(growth=DiamIf3-DiamIf2) %>%
    group_by(Especie) %>%
    mutate(growth_sp = mean(growth)) %>%
    group_by(Codi, Especie) %>%
    mutate(growth_sp_plot = mean(growth),
           inc = (growth_sp_plot /growth_sp))%>%
    arrange(desc(inc))
```

As we did before, we first calculate the growth of eaxh tree, we then group by species, so that we can calculate the mean growth for each species (`growth_sp`). Finally, we group again, now for plot and species, to calculate the mean growth of each species on each plot (`growth_sp_plot`). Once we have this, we can calculate the ratio between the two variables, identifying those plots where the species is performing better)assuming no mistakes, of course).

Let's see one last example:

*Ex.7.3 Select those species of the IFN3 occupied by "pure" Pinus nigra stands (Especie = 025)

Note: a forest is considered as pure stand if more than 80% of their Basal Area corresponds to a single species. Let's see how we would do that:

```
trees %>%
    group_by(Codi,Especie) %>%
    summarise(BA_sp= sum(BAIf3)) %>%
    group_by(Codi) %>%
    mutate(BA_tot = sum(BA_sp),
           ratio= BA_sp/BA_tot) %>%
    filter(Especie=="025", ratio >0.8)
```

```
## # A tibble: 648 x 5
## # Groups:   Codi [648]
##    Codi   Especie BA_sp BA_tot ratio
##    <fct>  <fct>   <dbl>  <dbl> <dbl>
##  1 080132 025      41.4   49.9  0.830
##  2 080307 025      51.2   54.8  0.933
##  3 080313 025      28.1   35.0  0.804
##  4 080318 025      23.7   26.6  0.894
##  5 080322 025      29.7   32.3  0.919
##  6 080323 025       1.61   1.61 1
##  7 080324 025      18.2   19.7  0.924
##  8 080325 025      43.8   44.6  0.982
##  9 080326 025       6.01   6.01 1
## 10 080328 025      62.1   65.7  0.945
## # ... with 638 more rows
```

In this case, we first calculate BA per plot, using `summarise`. We then calculate the sum of BA per plot, but in this case we use `mutate`, because we don't want to aggregate tge data by plot, but calculate them separately for each plot but keeping the rest of the data as it was. Once we have both values, we can filter to select those plots with Pinus nigra, in which percenrage of basal area for that species be $> 80\%$.

Joins: working with to tables

Very often, the information we will work with more than a table. The *join* functions will allow us to work with several data frames, joining them in different ways. Within `dplyr` there are two types of joins:

Mutating joins

They add the columns of a data frame to the other, depending on whether they share some observations or not. There are four types.

- `left_join(x, y)` adds the columns of `y` to the observations of `x` that are also in `y`. Those that are not present in y will receive the value `NA`. With this function we ensure that we will not lose any observation.
- `right_join(x, y)` adds the columns of `x` to those observations in `y` that are also in `x`. Those that are not present will receive `NA`. It is equivalent to `left_join`, but the columns will be ordered differently.
- `full_join(x,y)` includes all observations in `x` and `y`. If they do not coincide, they assign `NA`.
- `inner_join(x, y)` includes only those observations both in `x` and `y` (repeats rows if it is necessary).

Filtering joins

The second type of joins are the **filtering joins**, that affect only to the observation, not to the variables. That is, they never add new columns, but they keep or delete the rows of the original frame as a function of their correspondence or not with a second data frame. There are only two types:

- `semi_join(x, y)` *keeps* the observations in `x` that match observations in `y`.
- `anti_join(x, y)` *deletes* the observations in `x` that match observations in `y`.

You can find more information about the join functions typing `vignette("two-table")`.

###Exercise 8 To try the join functions, let's add the geographic information (X & Y coordinates), contained in the data frame `coordinates` to the data frame `plots`.

```
left_join(plots,coordinates, "Codi")
```

In this case, since we want to keep all the plots in the original `plots`data frame, we use `left_join`. In this case, since the number of observations in `coordinates` and `plots` is the same, the function `inner_join` should give us the same results.

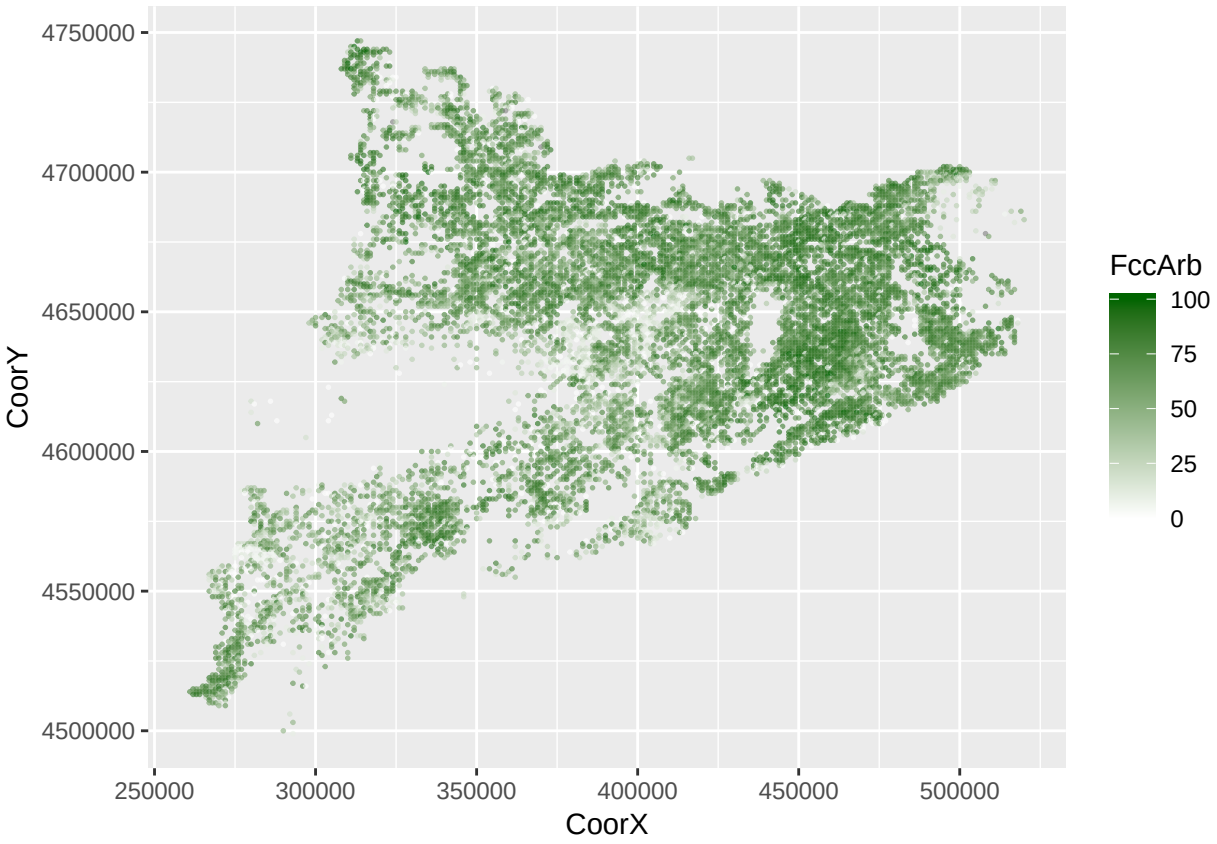
Now we added the coordinates, we can represent in a map any variable in the data frame. We could, for example, represent the values of tree canopy cover (FccArb). We need to load the package "maps". (If we don't have it installed, we can install it typing in the console `install.packages("maps")`).

```
library(maps)
```

```
##
## Attaching package: 'maps'
```

```
## The following object is masked from 'package:purrr':
##
##     map
```

```
left_join(plots,coordinates, "Codi") %>%
ggplot( aes(CoorX, CoorY)) +
    geom_point(aes(color=FccArb), size=0.3, alpha=0.6) +
    scale_color_continuous(low= "white", high="dark green")
```



Once again, we see we don't even need to create a new data frame with the new information, we can chain the functions in `dplyr` and `ggplot2`, producing the results in a very easy and fast way.

`tidyr`: changing the shape of the data frames

One of the main ideas behind the `tidyverse` is the concept of *tidy data*, that we have introduced before. According to Hadley Wickham, we can say that our data are tidy when two conditions are met:

- Each column corresponds to a variable
- Each row is a different observation

Of course, the data are not always organized in this way, sometimes other formats are more efficient (for example, for gathering data). For instance, if we have a look at the table `species`we will see that the number of trees for the different size classes are in different columns. This format is more convenient for entering the data or for some kinds of analyses, but in general the *tidy* format easens the processing and analysis, specially in vectorized languages such as `R`.

```
# View(species)
```

The `tidyr` package allows to change the way in which data are organized, so that we can arrange them in the way we need to our analysis. It has four basic verbs:

- `gather` aggregates variables that are in several columns and converts them into two variables: a factor (*key*) and a numeric variablea (*value*).
- `spread` is the inverse of gather, it takes the levels of a factor and a numeric variable and creates a new variable for each level of the factor.
- `separate` divides the content of a column into several columns
- `unite` inverse of separate, concatenates the values of several columns

`gather` & `separate`

`gather` transforms data in *wide* format into *long* format. `gather` takes a series of columns and transforms them into two variables: a factor (*key*) and a numeric variable (*value*). The first parameter in `gather` is the data frame, the second and third are the names we will give to *key* and *value*, and the rest are the variables to group.

###Exercise 9 Let's use `gather` and `separate` to transform the data frame `species` into a *tidy* format, where each column is a variable and each row, an observation. First, let's have a look at the data frame we want to transform:

```
glimpse(species)
```

To convert it into the 'long' format we specify the data frame, then the new factor to create (*key*), and the numeric variable (*value*), and last, the columns to aggregate. For the last part we have three equivalent options:

- (A) Explicitly define the variables we want to gather:

```
gather(species, CD, n, CD_10,CD_15,CD_20, CD_25,CD_30,
                    CD_35,CD_40, CD_45,CD_50,CD_55,CD_60, CD_65, CD_70)
```

- (B) Define the interval containing the variables we want to gather

```
gather(species, CD, n, CD_10:CD_70)
```

- (C) Define the variables with the helpers functions

```
gather(species, CD, n, starts_with('CD'))
```

- (D) Define the variables we DO NOT want to include (with -). The function will assume we want to gather the rest of variables.

```
    gather(species,CD, n,-Codi, -Especie)
```

The three pieces of code above produce the same result. Once we have converted the data frame into the new format, we can divide the new variable "CD" into two new variables, that we will name "Name" and "CD", using `separate`. If we do not specify where to make the separation, the function takes by defualt the first non alphanumeric character in the string.

```
    species_long <-   gather(species,CD, n,-Codi, -Especie)
    species_long<-separate(species_long, col=CD, into = c("Nombre", "CD"))
    species_long
```

`spread` & `unite`

If we have a data frame in *long* format, we can use `spread` and `unite` to transform it back into the *wide* format. This is what we will do in the next exercise, converting back the data frame with species and diameter classes into its original format. As with `gather`, `spread` takes the data frame as the first argument. The secod parameter is the factor we will use to create the new columns, and the third parameter is the name of the column that contains the values. We can see this with an example:

###Exercise 10 Use `unite` and `spread` to transform back the data to its origianl format.

First we create a new variable, that will be useful to create the new columns:

```
species_unite <- unite(species_long, CD, Nombre, CD)
```

Now we will transform the data frame, specifying the variable that will produce the new columns ("CD") and the variable that contains the values (n)

```
spread(species_unite, CD,n)
```

`purrr` fuctional programming

`purrr` allows for functional programming in R, meaning that functions becomes "first citizens'in the R environment (they can be used as arguments or returned from other functions).

Making loops pipe friendly

You can think of `map`, the main verb of `purrr`, as a *pipe-friendly* version of the apply family. It will accept functions as arguments and they will be applied to all elements of the list/vector provided.

Exercise 11

First thing is grouping by the plot code (`Codi`), but also, and this is important, by `Province` as we will need it later. Then we summarise the height with the mean, join the leaf dataset, split by province an map the lineal model. Optionally, we can use broom or summary to see the results:

```
trees %>%
  group_by(Codi, Provincia) %>%
  summarise(height = mean(HeiIf3, na.rm = TRUE)) %>%
  left_join(leaf, by = 'Codi') %>%
  split(.$Provincia) %>%
  map(~ lm(height ~ leaf_biomass, data = .)) %>%
  map_dfr(broom::tidy)
```

Other interesting functionalities of the tidyverse

Communication between packages

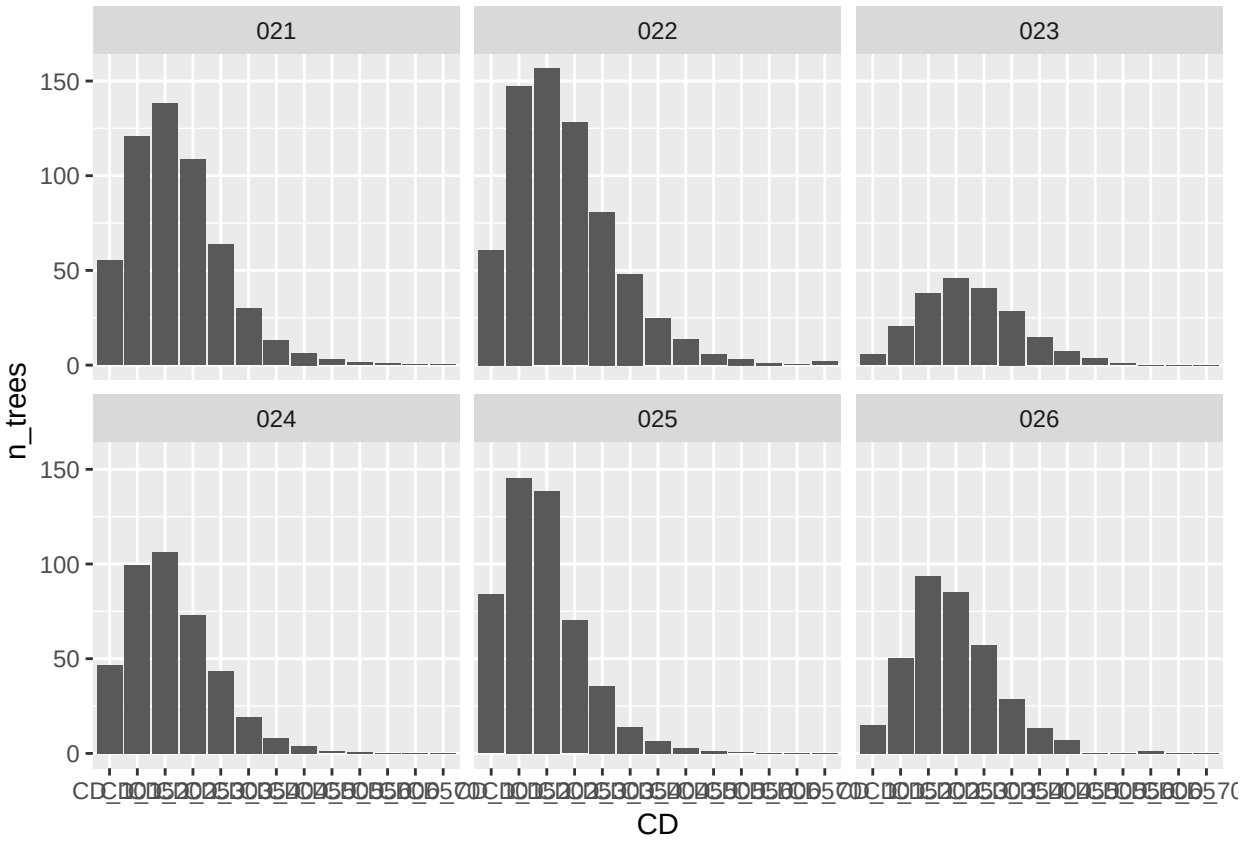
All the packages in the tidyverse are designed to communicate with each other. That means that we can combine `dplyr` and `tidyr` functions, connecting them trhough pipes (`%>%`). For example, if we wanted to know which the mean diameter distribution of all the pine species we could first filter the species we want to study (pines) then transform the data frame, group the data by species and diameter class and finally calculate the mean number of trees:

```
species %>%
    filter(Especie %in% c("021", "022","023", "024", "025", "026")) %>%
    gather(CD, n, CD_10:CD_70) %>%
    group_by(Especie,CD) %>%
    summarise(n_trees=mean(n))
```

```
## # A tibble: 78 x 3
## # Groups:   Especie [?]
##    Especie CD    n_trees
##    <fct>   <chr>   <dbl>
##  1 021     CD_10   55.4
##  2 021     CD_15  121.
##  3 021     CD_20  138.
##  4 021     CD_25  109.
##  5 021     CD_30   63.7
##  6 021     CD_35   29.8
##  7 021     CD_40   13.2
##  8 021     CD_45    6.43
##  9 021     CD_50    3.00
## 10 021     CD_55    1.39
## # ... with 68 more rows
```

But `dplyr` and `tidyr` also can connect to other packages in the `tidyverse`, such as `ggplot2` or `broom`, so we could expand on the previous code to generate a plot by species.

```
species %>%
    filter(Especie %in% c("021", "022","023", "024", "025", "026")) %>%
    gather(CD, n, CD_10:CD_70) %>%
    group_by(Especie,CD) %>%
    summarise(n_trees=mean(n)) %>%
    ggplot(aes(x=CD, y=n_trees)) +
    geom_col() +
    facet_wrap(~Especie)
```



Functional sequences

Another interesting aspect of `dplyr`is that we can save sequences of orders as an object, so they can later be applied to different data frames, as if it was a function. To do this, we must use the pronoun `.` as data frame in the sequence of orders to save. Let's see an example:

```
av_growth <-  . %>%
   mutate(growth=DiamIf3-DiamIf2) %>%
   group_by(Codi) %>%
   summarise(mean=mean(growth), n=n())
```

If we print the object, we will see it has a class `functional sequence`, and it specifies the orders to execute:

```
av_growth
```

```
## Functional sequence with the following components:
##
##  1. mutate(., growth = DiamIf3 - DiamIf2)
##  2. group_by(., Codi)
##  3. summarise(., mean = mean(growth), n = n())
##
## Use 'functions' to extract the individual functions.
```

We can then apply this sequence to a data frame...

```
    trees %>% av_growth()
```

```
## # A tibble: 7,713 x 3
##    Codi    mean     n
##    <fct>  <dbl> <int>
```

```
##  1 080001  3.33    15
##  2 080002  3.63    13
##  3 080003  5.93     7
##  4 080004  6.55     2
##  5 080005  2.08    12
##  6 080006  2.28    23
##  7 080007  2.45    35
##  8 080008  1.79    11
##  9 080009  1.86    16
## 10 080010  3.33    13
## # ... with 7,703 more rows
```

... or combine it with new `dplyr` or `tidyr` functions

```
    trees %>%
        filter(Provincia=="17") %>%
        av_growth()
```

```
## # A tibble: 2,113 x 3
##    Codi    mean     n
##    <fct>  <dbl> <int>
##  1 170004  2.99    39
##  2 170005  1.86    28
##  3 170006  1.75    31
##  4 170007  3.14    29
##  5 170008  1.68     9
##  6 170009  1.48    28
##  7 170010  2.40    26
##  8 170012  2.11    21
##  9 170013  1.55    34
## 10 170014  5.38     8
## # ... with 2,103 more rows
```

Databases

In this workshop we've seen how to work with `dplyr` and `tidyr` using data stored in our computer, but `dplyr`also allows working woth remote databases, admiting most formats and standards: PostgreSQL, MySQL, SQLite, MonetDB, BigQuery, Oracle...

When working with databases we will use the same verbs and coding we've seen so far, but `dplyr`transforms the `R` code into `SQL`sequences, so we don't need to change the language to read and analyze the data. Also, it is much faster than `R`. The details of working with databases are beyond the scope of this workshop, but you can find more information in the resources listed in the following section.

More info

Both the code and the data needed to generate this document and execute the examples can be found in GitHub (https://github.com/amezteguí/dplyr_workshop). You can also find more information about these packages and their functions in the book *R for data science* by Hadley Wickham, or in the vignettes for each function.

```
# Sobre dplyr
vignette("introduction")

# Sobre tidyr
vignette("tidy-data")

# Sobre unir dos tablas mediante join
vignette("two-table")

# Sobre trabajo con databases
vignette("databases")
```