Ejercicios y soluciones del workshop sobre `dplyr` y `tidyr`

Aitor Ameztegui

Enero de 2017

Contents

Introducción **2**

`tidyr`: cambiando la forma de los data frames **4**
- `gather` & `separate` . . . 4
- `spread` & `unite` . . . 5

`dplyr`: transformando los data frames **7**
- `filter` . . . 7
- `select` . . . 8
- `arrange` . . . 9
- `mutate` . . . 9
- `summarise` . . . 10

Pipelines (`%>%`) **12**

Grouped mutate/grouped filter **15**

Do **18**

Joins: trabajando con dos tablas **20**
- Mutating joins . . . 20
- Filtering joins . . . 20

Otras funcionalidades interesantes de `dplyr` **22**
- Comunicación entre paquetes . . . 22
- Secuencias funcionales . . . 23
- Databases . . . 25

Más información **26**

Introducción

Este es un documento `RMarkdown` generado para mostrar los ejercicios y las soluciones del taller/seminario sobre los paquetes `dplyr` y `tidyr` realizado en Solsona el 24 de enero de 2017. El código y los datos necesarios para generar este documento se pueden encontrar en mi GitHub (https://github.com/ameztegui/dplyr_workshop). Para cualquier duda sobre los ejercicios, consultar con Aitor Ameztegui (ameztegui@gmail.com).

Los paquetes `tidyr` y `dplyr` son parte de un conjunto de paquetes que ha dado en llamarse `tidyverse`, creado por Hadley Wickham , científico jefe en RStudio. El `tidyverse` se creó para facilitar el análisis de datos. Consta de paquetes para importar y leer datos, otros para organizarlos y modificarlos, paquetes de análisis y modelización y paquetes de visualización de resultados. En este seminario nos centraremos en `tidyr`, pensado para la organización de datos, y `dplyr`, que se centra en su transformación. Para trabajar con ellos lo primero es instalarlos, si no lo están ya. Podemos instalar de manera conjunta todos los paquetes del `tidyverse` escribiendo `install.packages("tidyverse")`. Posteriormente, los cargamos mediante:

```
library(tidyverse)
```

Además deberemos cargar los datos que utilizaremos durante el seminario, que se pueden descargar de GitHub.

```
load("../data/mayores.Rdata")
mayores <- tbl_df(mayores)

load("../data/parcelas.Rdata")
parcelas <- tbl_df(parcelas)

load("../data/especies.Rdata")
especies <- tbl_df(especies)

load("../data/coordenadas.Rdata")
coordenadas <- tbl_df(coordenadas)
```

Mediante la función `tbl_df` convertimos los data frames 'normales' en *tibbles*. Un *tibble* no es más que un data frame normal, pero presenta algunas ventajas, como el hecho de que imprime en pantalla por defecto las 10 primeras líneas (en vez de todo el data frame) y proporciona información sobre las variables y su tipo. Por lo demás, podemos considerarlas como data frames normales, ya que se comportan como tales a todos los efectos.

En estos ejercicios utilizaremos cuatro data frames con información relativa al segundo y tercer inventario forestal nacional (IFN2 e IFN3) en Cataluña. Los data frames son:

- **parcelas** [11,858 x 15]: todas las parcelas del IFN3 en Catalunya, con información sobre fecha y hora de medición, textura y pH del suelo, FCC total y FCC arbolada, etc.
- **mayores** [111,756 x 12]: todos los pies mayores ((dbh $>$ 7.5 cm) medidos tanto en

IFN2 e IFN3. Contiene la parcela, especie, clase diamétrica (CD), diámetro en el IFN2 y el IFN3, altura...

- **especies** [14,778 x 15]: el número de individuos por hectárea en cada parcela, por especie y clase diamétrica.
- **coordenadas** [11,858 x 6]: coordenadas X e Y de las parcelas del IFN3.

Lo primero que debemos hacer es echar un vistazo a los datos, familiarizarnos con ellos y las variables que contienen. Para ello usaremos la función `glimpse`

```
glimpse(parcelas)
glimpse(mayores)
glimpse(especies)
glimpse(coordenadas)
```

`tidyr`: cambiando la forma de los data frames

El concepto del `tidyverse` está muy relacionado con el de datos organizados o *tidy data*. Según Hadley Wickham, podemos decir que nuestros datos están organizados (son tidy) cuando cumplen dos condiciones:

- Cada columna corresponde a una variable diferente
- Cada fila es una observación diferente

No siempre los datos están organizados de esta manera, ya que a veces interesa otro tipo de formato. Por ejemplo, si echamos un vistazo a la tabla `especies` veremos que los valores de número de individuos de las distintas clases diamétricas están en diferentes columnas. Este formato puede ser más conveniente para introducir los datos o para determinados tipos de análisis, pero en general el formato *tidy* facilita el procesado y análisis, sobre todo en lenguajes vectorizados como `R`.

```
View(especies)
```

```
## Warning: running command ''/usr/bin/otool' -L '/Library/Frameworks/
## R.framework/Resources/modules/R_de.so'' had status 69
```

El paquete `tidyr` permite cambiar la organización de los datos, de manera que los dispongamos en el formato necesario para nuestro análisis. Tiene cuatro verbos (funciones) básicos:

- `gather` reúne variables que están en varias columnas y las convierte en dos variables: un factor (*key*) y una variable numérica (*value*).
- `spread` el inverso de gather, toma los niveles de un factor y una variable numérica y crea una nueva variable para cada nivel del factor.
- `separate` separa los valores de una columna en varias
- `unite` opuesto de separate, concatena los valores de varias columnas

`gather` & `separate`

La función `gather` transforma datos de formato *ancho* a formato *largo*. `gather` toma una serie de columnas y las transforma en dos variables: un factor (*key*) y un valor (*value*). El primer parámetro de `gather` es la base de datos, el segundo y tercero son el nombre que daremos a las variables *key* y *value*, y el resto son las variables a agrupar.

Ejercicio 1

Probemos a usar `gather` y `separate` para transformar el data frame `especies` en un formato *tidy*, donde cada columna sea una variable y cada fila, una observación.

Primero echemos un ojo a la base de datos que debemos transformar

```
glimpse(especies)
```

Para convertirlo a formato 'largo', especificaremos primero el data frame, después, el nuevo factor a crear (*key*), la nueva variable numérica (*value*). Para especificar las columnas a agregar tenemos tres opciones equivalentes:

- (A) Especificar expresamente las variables que queremos reunir

```
gather(especies, 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) Especificar el intervalo de variables que queremos reunir

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

- (C) Especificar las variables que NO queremos reunir (con -). El resto de variables se asume que forma parte del proceso de agrupamiento.

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

Una vez convertido el data frame al nuevo formato, podemos convertir la nueva variable "CD" en dos nuevas variables, que llamaremos "Nombre" y "CD", usando `separate`

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

`spread` & `unite`

Al inverso que antes, si tenemos una base de datos en formato *largo*, podemos usar `spread` y `unite` para volver a transformarla a formato *ancho*. Es lo que haremos en el siguiente ejercicio, convirtiendo la base de datos de distribución diamétrica y especies a su formato original. Al igual que `gather`, `spread` pasa como primer argumento la base de datos. El segundo parámetro es el factor que usaremos para formar las nuevas columnas, mientras que el tercer parámetro es el nombre de la columna que contiene los valores. Podemos verlo con un ejemplo:

Ejercicio 2

Usaremos `unite` y `spread` para volver a transformar los datos de antes al formato original.

Primero crearemos una nueva variable, que servirá para crear las nuevas columnas:

```
especies_unite <- unite(especies_long, CD, Nombre, CD)
```

Ahora transformamos la base de datos, especificando la variable que conformará las nuevas columnas (CD) y la que contiene los valores (n)

```
spread(especies_unite, CD,n)
```

`dplyr`: transformando los data frames

Si `tidyr` sirve para organizar las bases de datos, `dplyr` sirve para transformarlas: crear nuevas variables, seleccionar las que nos interesan, ejecutar filtros, etc. El paquete `dplyr` tiene cinco verbos básicos:

- `filter` selecciona filas en base a un criterio definido
- `select` seleccionar columnas en base a su nombre
- `arrange` ordenar el data frame en base a una o varias variables
- `mutate` crear nuevas variables
- `summarise` crear nuevas variables que resumen los valores de una variable (media, suma, etc)

Todos ellos tienen la misma estructura: el primer argumento de la función es el `data frame`al que se aplica, y el resto de argumentos especifica qué hacer con ese `data frame`.

`filter`

La función `filter` selecciona las filas del data frame que cumplen un determinado criterio. El primer argumento es el data frame, y los posteriores son los criterios, que se pueden concatenar mediante comas.

Ejercicio 3

Para practicar la función `filter` probemos a encontrar las parcelas del IFN que:

- 3.1 Se encuentren en Barcelona (08) o Girona (17). Tenemos dos opciones

```
# Opción 1
filter (parcelas, Provincia =="08" | Provincia =="17")

# Opción 2
filter (parcelas, Provincia %in% c("08", "17"))
```

En cambio, la siguiente opción no sería válida, ya que hay que especificar expresamente la variable en cada nueva condición.

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

- 3.2 Parcelas que se acabaron de medir en enero de 2001

Para esto debemos definir que la fecha de finalización sea posterior al 31 de diciembre de 2000 y anterior al 1 de febrero de 2001. Tenemos dos maneras de hacerlo: la primera implica utilizar el operador `&` para indicar que queremos las parcelas que cumplen un criterio y el

otro. La segunda opción sería simplemente encadenar los criterios mediante una coma, ya que `filter` asume que se deben cumplir todos ellos.

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

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

- 3.3 Se tardó más de dos horas en medirlas (7200 seg)

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

Como vemos, podemos hacer operaciones dentro de las condiciones de `filter`. En este caso, queremos que la diferencia entre la hora de finalización y la de inicio sea menor de 7200 s (2 horas).

select

La función `select` nos permite seleccionar unas columnas determinadas en base a su nombre. Hay algunas funciones específicas, como `starts_with` o `contains`, que sólo funcionan dentro de `select` y que ayudan a identificar las columnas que nos interesen.

Ejercicio 4

Para practicar la función `select` probemos a encontrar cuatro maneras distintas de seleccionar las variables que marcan la fecha de inicio y fin de medicion de las parcelas (FechaIni y FechaFin)

- Por ejemplo, podemos especificar las columnas que queremos

```
select(parcelas, FechaIni, FechaFin)
```

- También podemos especificarlas como un rango, de manera que se seleccionan todas las columnas entre las dos especificadas.

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

- O las columnas que contienen el texto 'fecha'. En este caso, seleccionaremos también la fecha de medición del pH, que en principio no nos interesa, pero podemos eliminarla:

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

- También podríamos seleccionar las variables que empiezan por 'fecha' (en este caso también tendríamos que eliminar FechaPh):

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

arrange

`arrange` ordena la base de datos en base a una o más variables. El primer argumento será, como siempre, el data frame que queremos reordenar, y después especificamos las variables que determinan la ordenación. Si especificamos más de 1 variable, las sucesivas variables servirán para romper los empates de las anteriores. También podemos usar 'desc(x)' para ordenar en orden decreciente. Probemos con unos cuantos ejercicios:

Ejercicio 5

- Ej.5.1 Ordenar las parcelas por fecha y hora de medición

```
arrange(parcelas, FechaFin, HoraFin)
```

- Ej. 5.2 ¿Qué parcelas se empezaron a medir más tarde en el día?

```
arrange(parcelas,desc(HoraIni))
```

- Ej. 5.3 ¿Cuáles tardaron más en medirse?

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

Vemos que, igual que con `filter`, también podemos ordenar un data frame en base al resultado de una operación aritmética.

mutate

`mutate` nos permite crear nuevas variables como combinación de las ya existentes. Simplemente tenemos que especificar el data frame e indicar el nuevo nombre de las variables a crear y su valor. Veamos unos ejemplos:

Ejercicio 6

Probemos a calcular dos nuevas variables

- 6.1 Una variable que calcule el crecimiento en cm en base al diámetro en el IFN2 y el IFN3.

```
mayores <- mutate (mayores, crec= DiamIf3 - DiamIf2)
```

- 6.2 Crear dos nuevas variables con el área basimétrica por hectárea que representa cada árbol, tanto en el IFN2 como en el IFN3. ¿De qué especie es el árbol que creció más rápido en AB?

```
mayores <- mutate(mayores, BAIf2= (((pi/4)*(DiamIf2/100)^2)*Fac),
                  BAIf3= (((pi/4)*(DiamIf3/100)^2)*Fac))
```

```r
arrange(mayores, desc(BAIf3-BAIf2))
```

Como vemos, tras calcular el área basimétrica podemos saber qué árbol creció más rápido combinando `mutate` con `arrange`.

summarise

En el caso de `summarise`, nos permite hacer cálculos con las variables del data frame, pero utiliza funciones de agregación (*summary functions*), que resumen las variables a un sólo valor. Funciones como `sum`, `mean`, `max`,`IQR`, etc. son ejemplos de funciones de agragación. Sin embargo, esta función por si misma no tiene normalmente mayor interés, ya que reduciría toda nuestra base de datos a un solo valor. Por ello se suele usar en conjunción con `group_by`, que clasifica el data frame en grupos en función de una variable categórica.

Para usar `group_by` basta con indicar el data frame y la variable por la que lo queremos agrupar. Para ser más eficiente, `dplyr` no crea una copia del data frame, sino que sólo crea una variable oculta que indexa los grupos, de manera que cuando le pedimos operaciones por grupo, sabe a qué grupo pertenece cada observación.

En el caso de nuestra base de datos de pies mayores, hay varios grupos que pueden tener interés:

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

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

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

#Por clase diamétrica
by_CD <- group_by (mayores, CD)

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

Puedes ver, escribiendo por ejemplo `glimpse(by_plot)` que el data frame resultante no se diferencia en nada del original, aparentemente.

Ejercicio 7

Una vez hecho esto, ¿qué estadísticas podríamos calcular para caracterizar los valores de diámetro para cada parcela? Probemos a calcular la media, valor mínimo y máximo, el

percentil 90 y el rango intercuartil para cada parcela. En este caso vemos que el valor resultante ya tiene menos filas, en concreto, una por parcela, y que sólo contiene las nuevas variables creadas.

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

```
## # A tibble: 7,713 <U+00D7> 6
##      Codi    media   min   max    q90     IQ
##    <fctr>    <dbl> <dbl> <dbl>  <dbl>  <dbl>
## 1  080001 26.25667 13.45 38.00 34.400  4.150
## 2  080002 35.16154 24.85 44.40 43.030 10.200
## 3  080003 31.96429 14.25 51.00 46.080 12.225
## 4  080004 24.27500 16.85 31.70 30.215  7.425
## 5  080005 28.36667 16.15 59.85 39.730 15.300
## 6  080006 35.94565 14.00 55.90 52.310 17.900
## 7  080007 30.76286 15.25 63.55 49.390 12.275
## 8  080008 16.04545  9.00 21.45 17.750  2.050
## 9  080009 16.68750  9.00 35.95 24.000  3.875
## 10 080010 31.55385  9.20 95.50 61.540 18.150
## # ... with 7,703 more rows
```

Pipelines (%>%)

A menudo, los verbos de `dplyr` se utilizan de manera conjunta, creando funciones anidadas. Sin embargo, estas funciones pueden ser complejas si encadenan numerosas órdenes, y al final se pueden hacer difíciles de entender. Por ejemplo, observa el siguiente código. ¿Eres capaz de saber lo que hace?

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

El código toma, de la base de datos 'mayores', aquellos pies con valor existente de diámetro (`!is.na(DiamIf3)`), después los agrupa por parcela y especie (`group_by(Codi, Especie)`), calcula para cada una de estas combinaciones el diámetro medio (`diam = mean (DiamIf3)`), y el número de pies por parcela (`n = n()`), y selecciona finalmente sólo aquellos casos en los que haya más de 5 pies (`filter (n>5)`).

Esta sintaxis, pese a no ser operaciones complejas, se hace complicada de entender, y se suele solucionar guardando cada paso como un data frame diferente, lo cual es una fuente importante de errores.

Sin embargo, es posible simplificar este código mediante el operador *pipe* (`%>%`) del paquete `magrittr`, que se carga directamente con `tidyr` y `dplyr`. Cuando usamos `%>%`, el resultado de la parte izquierda se pasa a la función de la derecha como primer argumento. En el contexto de `dplyr`y `tidyr`, como el primer argumento es siempre un data frame, `%>%` hace que una función se aplique al data frame resultante de la función anterior. Así, podemos expresar `filter (df, color =="blue)` como `df %>% filter(color == "blue")`. Esto permite concatenar funciones de manera más lógica e inteligible, de forma que el operador `%>%` se leería como *luego*. Veamos como quedaría el ejemplo de más arriba.

```
diam_medio_especie <- mayores %>%           # toma el df 'mayores' y LUEGO
filter(!is.na(DiamIf3)) %>%                 # elimina los valores NA y LUEGO
group_by(Codi, Especie) %>%                 # agrupa por parc. y especie y LUEGO
summarise(diam=mean(DiamIf3), n = n()) %>%  # calcula media y n de pies y LUEGO
filter(n > 5)                               # filtra aquellos con n> 5
```

Ejercicio 8

Veamos ahora algunos ejercicios. Usando este operador, probemos a crear pipelines para los siguientes enunciados:

- 8.1 ¿Qué parcelas tienen el mayor crecimiento medio?

Primero definimos el data frame con el que trabajaremos. *LUEGO* (%>%) creamos una nueva variable con el crecimiento de cada árbol, *LUEGO* agrupamos por parcela, *LUEGO* calculamos, para cada parcela, el crecimiento medio, y *LUEGO* ordenamos los resultados de este crecimiento en orden decreciente. El código quedaría de la siguiente manera:

```
mayores %>%
    mutate(crec=DiamIf3-DiamIf2) %>%
    group_by(Codi) %>%
    summarise(media=mean(crec), n=n()) %>%
    arrange(desc(media))
```

```
## # A tibble: 7,713 <U+00D7> 3
##      Codi    media     n
##    <fctr>    <dbl> <int>
## 1  171089 23.06667     3
## 2  170819 21.60000     1
## 3  172607 17.60000     6
## 4  172216 17.38333     6
## 5  172690 15.95294    17
## 6  171682 15.41667     6
## 7  083267 15.35000     1
## 8  431363 15.07500     4
## 9  171664 14.82000     5
## 10 171976 14.35000     1
## # ... with 7,703 more rows
```

- 8.2 ¿Cuál es la parcela con mayor número de especies?

El proceso a seguir es: primero, identificar el data frame (`mayores`), *LUEGO* agruparlos por parcela, *LUEGO* determinar el número de especies distintas por parcela y *LUEGO* ordenar de manera decreciente. Veamos:

```
mayores %>%
    group_by(Codi) %>%
    summarise (especies=n_distinct(Especie)) %>%
    arrange(desc(especies))
```

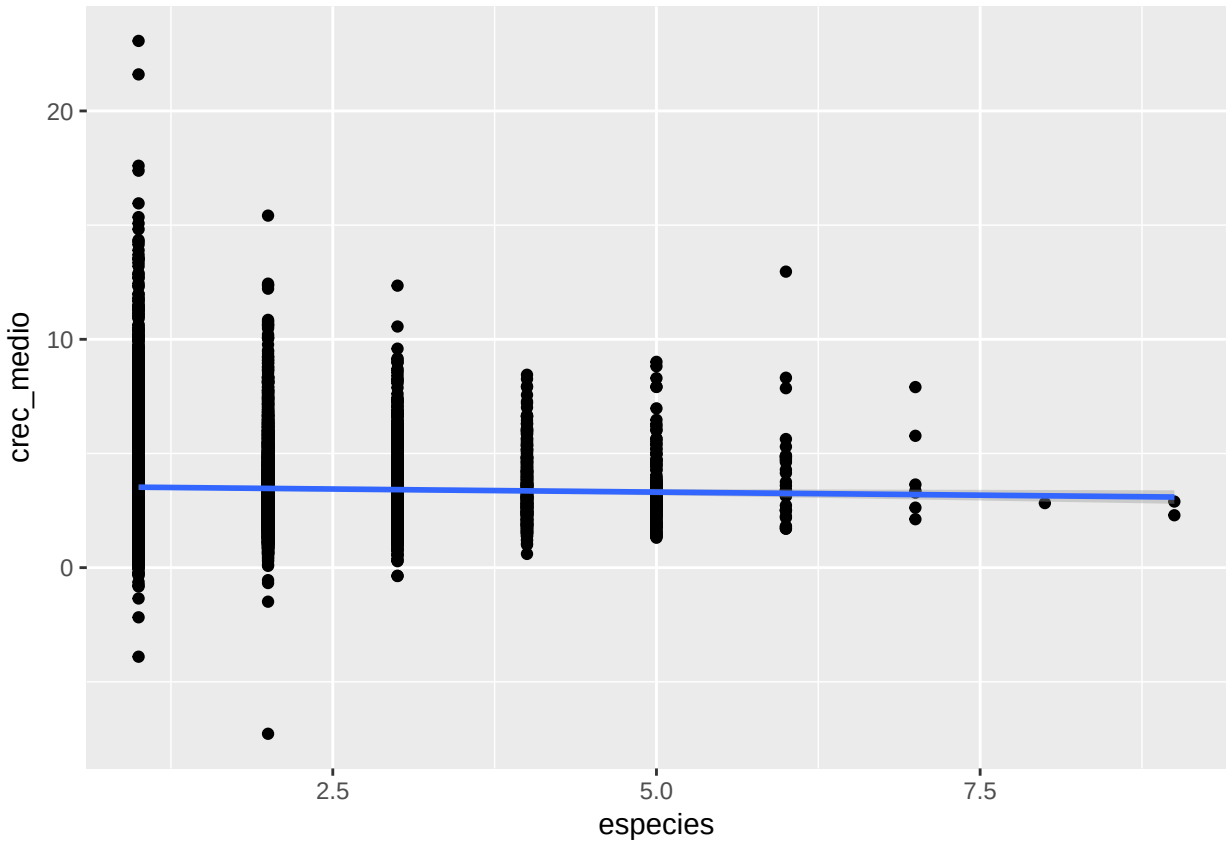
```
## # A tibble: 7,713 <U+00D7> 2
##      Codi especies
##    <fctr>    <int>
## 1  170195        9
```

```
## 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
```

- 8.3 ¿Hay relación entre ambas variables?

Primero, indicamos el data frame con el que trabajaremos, *LUEGO* agrupamos por parcela, *LUEGO* calculamos las nuevas variables. Para ver la relación entre ambas vamos a crear un gráfico con `ggplot2`, simplemente para mostrar cómo `dplyr` y `tidyr` se comunican con `ggplot2`. Como el objetivo de este workshop no es aprender a usar `ggplot`, de momento no daremos más detalles. Para saber más sobre este paquete de visualización, podéis visitar esta web: http://ggplot2.org/.

```
mayores %>%
    mutate(crec=DiamIf3-DiamIf2) %>%
    group_by(Codi) %>%
        summarise (especies=n_distinct(Especie),
               crec_medio=mean(crec)) %>%
    ggplot(aes(especies, crec_medio)) +
    geom_point() +
    geom_smooth(method = "lm")
```



Aquí estamos viendo una de las ventajas del `tidyverse`, el hecho de que los paquetes y funciones se entiendan entre si. De esta manera, acabamos de crear una figura sin necesidad de crear objetos y data frames intermedios, partiendo directamente del data frame original y encadenando ordenes de manera lógica e intuitiva.

Grouped mutate/grouped filter

La mayor parte de las veces que usmos `group_by`, lo haremos con las llamadas *summary functions*, es decir, funciones que toman n valores como input, y devuelven 1 valor como output. Ejemplos de *summary functions* son `mean()`, `sd()`, `min()`, `sum()`, etc.

Otras veces, sin embargo, necesitamos realizar operaciones por grupo, pero que el output sea por caso, es decir n inputs –> n outputs. Esto se puede hacer pefectamente usando `mutate` o `filter`en combinación con `group_by`.

Ejercicio 9

Teniendo esto que acabamos de ver en cuenta, probemos a:

- 9.1 Identificar los árboles que crezcan mucho más que la media en esa parcela

```
mayores %>%
    mutate(crec=DiamIf3-DiamIf2) %>%
    group_by(Codi) %>%
    mutate(des = (crec - mean(crec))/sd(crec)) %>%
    arrange(desc(des))
```

```
## Source: local data frame [111,756 x 16]
## Groups: Codi [7,713]
##
##      Codi Provincia    Cla Subclase Especie Rumbo  Dist    Fac    CD
##    <fctr>     <chr> <fctr>   <fctr>  <fctr> <dbl> <dbl>  <dbl> <dbl>
## 1  431306        43      A        1     025   174  9.89  31.83    30
## 2  083073        08      A        1     023   130 13.00  14.14    35
## 3  250904        25      A        1     022   221  5.50  31.83    30
## 4  170055        17      A        1     031    52  8.10  31.83    35
## 5  170916        17      A        1     024    55 22.00   5.09    70
## 6  250010        25      A        1     071   214 32.90   5.09    70
## 7  172462        17      A        1     026   158  6.50  31.83    35
## 8  083037        08      A        1     026    70  8.60  31.83    30
## 9  251126        25      A        1     025   363  3.79 127.32    30
## 10 172223        17      A        1     046   383 10.39  14.14    25
## # ... with 111,746 more rows, and 7 more variables: DiamIf3 <dbl>,
## #   DiamIf2 <dbl>, HeiIf3 <dbl>, crec <dbl>, BAIf2 <dbl>, BAIf3 <dbl>,
## #   des <dbl>
```

En el código anterior vemos que primero calculamos el crecimiento de cada árbol, y después de agrupar por parcela, calculamos una nueva variable, donde al crecimiento de cada árbol se le resta la media (de la parcela) y se divide por la sd (también de la parcela). De esta manera, hemos calculado el crecimiento estandarizado de cada arbol respecto a la parcela en la que se encuentra, y ahora es muy facil identificar aquellos que crecen sospechosamente más que lo normal para su parcela.

- 9.2 Identificar las parcelas donde una especie crece mucho más que la media de la especie

```
mayores %>%
    mutate(crec=DiamIf3-DiamIf2) %>%
    group_by(Especie) %>%
    mutate(crec_sp = mean(crec)) %>%
    group_by(Codi, Especie) %>%
    mutate(crec_sp_plot = mean(crec),
           inc = (crec_sp_plot /crec_sp))%>%
    arrange(desc(inc))
```

Del mismo modo que antes, primero calculamos el crecimento para cada árbol. Posteriormente, agrupamos por especie, de manera que podemos calcular la media de crecimiento para cada

especie (`crec_sp`). Finalmente, volvemos a agrupar, esta vez por parcela y especie, para calcular el crecimiento medio de una especie en una parcela determinada (`crec_sp_plot`). Una vez tenemos esto, podemos ver el ratio entre crecimiento de la especie en la parcela y crecimiento de la especie en general, identificando aquellas parcelas donde la especie se desarrolla mejor.

Do

En ocasiones, ninguno de los verbos implementados en `dplyr` será suficiente, y necesitaremos implementar otras funciones por grupo. Esto se puede conseguir con la función `do`. Esta función es mucho más lenta que los verbos implementados por defecto, pero permite aplicar cualquier tipo de función. Como la función que incluiremos en `do` no tiene porqué entender los pipelines (`%>%`), la función o verbo `do` requiere un pronombre (`.`) para indicar que la función se aplica al grupo definido antes. Así, podemos por ejemplo usarla para observar las tres primeras líneas de cada parcela y especie, mediante la función `head`:

```
head(mayores, 3)
```

```
## # A tibble: 3 <U+00D7> 15
##     Codi Provincia    Cla Subclase Especie Rumbo  Dist   Fac    CD DiamIf3
##   <fctr>     <chr> <fctr>   <fctr>  <fctr> <dbl> <dbl> <dbl> <dbl>   <dbl>
## 1 080001        08      A        1     022     7   8.3 31.83    20    20.3
## 2 080002        08      A        1     476    38   9.1 31.83    35    34.0
## 3 080003        08      A        1     021    25   7.0 31.83    25    24.8
## # ... with 5 more variables: DiamIf2 <dbl>, HeiIf3 <dbl>, crec <dbl>,
## #   BAIf2 <dbl>, BAIf3 <dbl>
```

```
mayores %>%
    group_by(Codi, Especie) %>%
    do(head(., 3))
```

```
## Source: local data frame [35,472 x 15]
## Groups: Codi, Especie [14,778]
##
##      Codi Provincia    Cla Subclase Especie Rumbo  Dist    Fac    CD
##    <fctr>     <chr> <fctr>   <fctr>  <fctr> <dbl> <dbl>  <dbl> <dbl>
## 1  080001        08      A        1     022     7  8.30  31.83    20
## 2  080001        08      A        1     022    44 16.10   5.09    35
## 3  080001        08      A        1     022    80 14.10  14.14    25
## 4  080002        08      A        1     021   107  6.59  31.83    45
## 5  080002        08      A        1     021   114  5.09  31.83    35
## 6  080002        08      A        1     021   145 14.10  14.14    35
## 7  080002        08      A        1     022    68  5.19  31.83    25
## 8  080002        08      A        1     022   139  3.90 127.32    25
## 9  080002        08      A        1     022   177 14.30  14.14    25
## 10 080002        08      A        1     476    38  9.10  31.83    35
## # ... with 35,462 more rows, and 6 more variables: DiamIf3 <dbl>,
## #   DiamIf2 <dbl>, HeiIf3 <dbl>, crec <dbl>, BAIf2 <dbl>, BAIf3 <dbl>
```

O incluso podemos usarlo para ajustar un modelo para cada grupo. En este caso, ajustaremos un modelo lineal por especie, relacionando altura y diámetro, pero podríamos usar cualquier función creada por nosotros.

```
models <- mayores %>%
 group_by(Especie) %>%
 do(mod=lm(HeiIf3 ~DiamIf3, data=.))

models
```

```
## Source: local data frame [91 x 2]
## Groups: <by row>
##
## # A tibble: 91 <U+00D7> 2
##    Especie      mod
## *   <fctr>   <list>
## 1      004 <S3: lm>
## 2      008 <S3: lm>
## 3      012 <S3: lm>
## 4      013 <S3: lm>
## 5      014 <S3: lm>
## 6      015 <S3: lm>
## 7      016 <S3: lm>
## 8      017 <S3: lm>
## 9      019 <S3: lm>
## 10     021 <S3: lm>
## # ... with 81 more rows
```

Vemos que la salida de datos crea una columna, llamada `mod` (el nombre lo hemos definido nosotros), que en realidad es una lista. Cada uno de los elementos de esa lista contiene la salida del modelo lineal ajustado a esa especie. Si queremos obtener, por ejemplo, los coeficientes del modelo lineal ajustado para el pino silvestre (codigo ==“021”), bastará con hacer:

```
models$mod[models$Especie=="021"]
```

```
## [[1]]
##
## Call:
## lm(formula = HeiIf3 ~ DiamIf3, data = .)
##
## Coefficients:
## (Intercept)      DiamIf3
##      7.2471       0.2099
```

Joins: trabajando con dos tablas

A menudo la información con la que trabajaremos no está almacenada en una sola tabla, sino en varias. Las funciones *join* nos permiten trabajar con varios data frames, uniéndolos según varios criterios. En `dplyr` hay dos tipos de joins:

Mutating joins

Son los que añaden las columnas de un data frame al otro, en función de si comparten o no determinadas observaciones. Hay cuatro tipos.

- `left_join(x, y)` añade las columnas de `y` a aquellas observaciones de `x` que también estén en `y`. Las que no lo estén recibirán `NA`. Con esta función aseguramos no perder observaciones de la lista original.
- `right_join(x, y)` añade las columnas de `x` a aquellas observaciones de `y` que también estén en `x`. Las que no lo estén recibirán `NA`. Es equivalente a left_join, pero las columnas se ordenarán de manera diferente.
- `full_join(x,y)` incluye todas las observaciones en `x` e `y`. Si no coinciden, asigna `NA`.
- `inner_join(x, y)` incluye sólo las observaciones que coinciden en `x` e `y` (repite filas si se da el caso).

Filtering joins

El segundo tipo de joins son los **filtering joins**, que sólo afectan a las observaciones, no a las variables. Es decir, no añaden nuevas columnas, sino que conservan o eliminan las filas del data frame original en función de si coinciden o no con las filas de un segundo data frame. Hay dos tipos:

- `semi_join(x, y)` *mantiene* las observaciones en `x` que coinciden con observaciones en `y`.
- `anti_join(x, y)` *elimina* las observaciones en `x` que coinciden con observaciones en `y`.

Podéis encontrar más información sobre los joins tecleando el siguiente código: `vignette("two-table")`.

Ejercicio 10

Para probar a utilizar las funciones join, probemos a añadir la información geográfica (coordenadas X e Y) contenida en el data frame `coordenadas`al data frame de las `parcelas`.

```
left_join(parcelas,coordenadas, "Codi")
```

En este caso, como queremos mantener todas las parcelas del data frame original, la función a usar sería `left_join`. En cualquier caso, como el número de observaciones `coordenadas`y en `parcelas`es el mismo, la función `inner_join`debería darnos el mismo resultado.

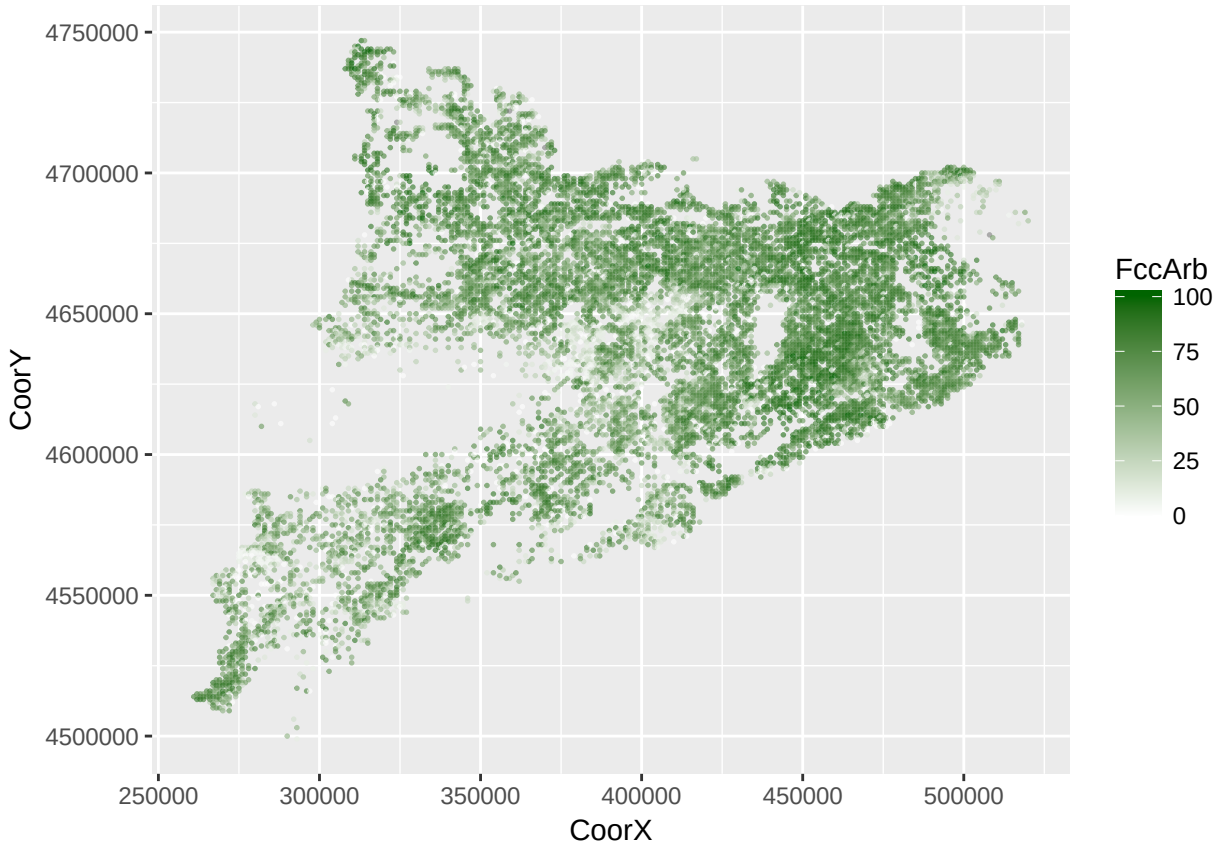
Ahora que tenemos las coordenadas añadidas, podemos representar en un mapa cualquier variable del data frame `coor`. En este caso, vamos a representar los valores de FCC arbolada (FccArb). Para ello necesitamos tener instalado el paquete "maps"

```
library(maps)
```

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

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

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



Una vez más, vemos que ni siquiera es necesario crear un nuevo data frame con la nueva información, sino que podemos encadenar las funciones y órdenes de `dplyr` y `ggplot2`, y el resultado se genera de forma casi instantánea.

Otras funcionalidades interesantes de `dplyr`

Comunicación entre paquetes

Además de todas las funciones y posibilidades que ya hemos visto, `dplyr` presenta otras ventajas dignas de mencionar. Por ejemplo, como ya hemos mencionado, están diseñados para comunicarse entre si, de manera que podemos concatenar funciones de `dplyr` y `tidyr` indistintamente, conectándolas con el operador `%>%`. Por ejemplo, si quisieramos saber cuál es la distribución diamétrica media de las especies de pino podríamos primero filtrar las especies que nos interesan, después transformar el data frame para ponerlo en formato *tidy*, agrupar los datos por especie y calcular el número de pies medio.

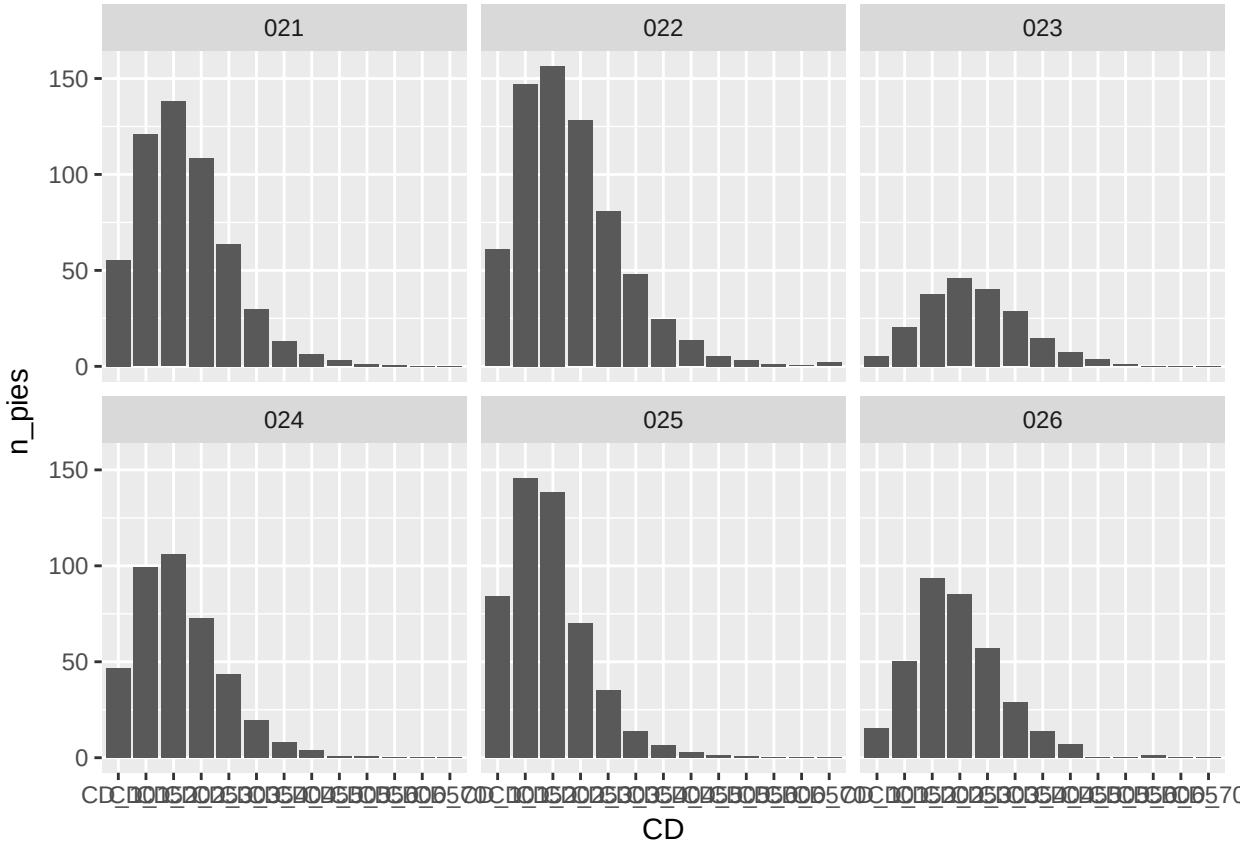
```
especies %>%
    filter(Especie %in% c("021", "022","023", "024", "025", "026")) %>%
    gather(CD, n, CD_10:CD_70) %>%
    group_by(Especie,CD) %>%
    summarise(n_pies=mean(n))
```

```
## Source: local data frame [78 x 3]
## Groups: Especie [?]
##
##    Especie    CD     n_pies
##     <fctr> <chr>      <dbl>
## 1      021 CD_10  55.409367
## 2      021 CD_15 120.890058
## 3      021 CD_20 138.019205
## 4      021 CD_25 108.629969
## 5      021 CD_30  63.725341
## 6      021 CD_35  29.755332
## 7      021 CD_40  13.175198
## 8      021 CD_45   6.432053
## 9      021 CD_50   2.997165
## 10     021 CD_55   1.387929
## # ... with 68 more rows
```

Pero además de entenderse entre si, `dplyr`y `tidyr` también se entienden con otros paquetes del `tidyverse`, como `ggplot2` o `broom`, de manera que para generar un plot no necesitaríamos crear ningun objeto intermedio, podríamos pasar directamente de los datos brutos a la figura final.

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

```
    geom_col() +
    facet_wrap(~Especie)
```



Secuencias funcionales

Otro aspecto interesante de `dplyr` es que permite guardar las secuencias de órdenes como un objeto, de manera que después ese objeto se puede aplicar a diferentes data frames, como si se tratara de una función. Para ello, debemos usar el pronombre `.`como data frame de la secuencia de órdenes a almacenar. Veamos un ejemplo.

```
crec_medio <-  . %>%
   mutate(crec=DiamIf3-DiamIf2) %>%
   group_by(Codi) %>%
   summarise(media=mean(crec), n=n())
```

Si imprimimos el objeto, vemos que tiene un formato `functional sequence`, y nos especifica las oórdenes que ejecuta:

```
crec_medio
```

```
## Functional sequence with the following components:
##
```

```
##  1. mutate(., crec = DiamIf3 - DiamIf2)
##  2. group_by(., Codi)
##  3. summarise(., media = mean(crec), n = n())
##
## Use 'functions' to extract the individual functions.
```

Después podremos aplicar esta secuencia a un data frame...

```
    mayores %>% crec_medio()
```

```
## # A tibble: 7,713 <U+00D7> 3
##      Codi    media     n
##    <fctr>    <dbl> <int>
## 1  080001 3.326667    15
## 2  080002 3.634615    13
## 3  080003 5.928571     7
## 4  080004 6.550000     2
## 5  080005 2.083333    12
## 6  080006 2.278261    23
## 7  080007 2.447143    35
## 8  080008 1.786364    11
## 9  080009 1.856250    16
## 10 080010 3.326923    13
## # ... with 7,703 more rows
```

... o combinarlo con nuevas órdenes de `dplyr`o `tidyr`

```
    mayores %>%
        filter(Provincia=="17") %>%
        crec_medio()
```

```
## # A tibble: 2,113 <U+00D7> 3
##      Codi    media     n
##    <fctr>    <dbl> <int>
## 1  170004 2.991026    39
## 2  170005 1.855357    28
## 3  170006 1.746774    31
## 4  170007 3.136207    29
## 5  170008 1.677778     9
## 6  170009 1.478571    28
## 7  170010 2.398077    26
## 8  170012 2.107143    21
## 9  170013 1.550000    34
## 10 170014 5.375000     8
## # ... with 2,103 more rows
```

Databases

En este tutorial hemos visto como trabajar con `dplyr` y `tidyr`usando datos almacenados en el ordenador. Sin embargo, `dplyr` también permite trabajar con bases de datos remotas, admitiendo los principales formatos y estándares: PostgreSQL, MySQL, SQLite, MonetDB, BigQuery, Oracle...

En realidad, usaremos los mismos verbos que hemos trabajado hasta ahora, y `dplyr` se encarga de transformar las órdenes en secuencias de SQL, de manera que no es necesario cambiar de lenguaje mientras analizamos los datos. Los detalles del trabajo con databases quedan fuera del ámbito de este seminario, pero se puede encontrar información detallada en el siguiente apartado.

Más información

Tanto el código como los datos necesarios para generar este documento y ejecutar los ejemplos se pueden encontrar en mi GitHub (https://github.com/ameztegui/dplyr_workshop). Puedes encontrar más información sobre estos paquetes y sus funciones en el libro *R for data science* de Hadley Wickham, o escribiendo el siguiente código.

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

# Sobre tidyr
vignette("tidy-data")

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

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