Explorar, transformar y visualizar

PID_00268328

Jordi Mas Elias

Tiempo mínimo de dedicación recomendado: 3 horas

Universitat Oberta de Catalunya

Jordi Mas Elias

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Jordi Mas Elias (2019)

Primera edición: septiembre 2019

Avda. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Índice

Introducción

El objetivo de este módulo es familiarizarnos con el entorno R de una manera ágil y sin demasiada carga teórica para poder aplicar rápidamente los conocimientos aprendidos al uso del software. En las páginas siguientes aprenderemos en pocos pasos a transformar un marco de datos de dimensiones considerables en información útil y visualmente atractiva. Para este propósito tendremos que aprender las funciones básicas de dos de los paquetes esenciales de R: *dplyr* y *ggplot2*. La librería *dplyr* incluye principalmente funciones orientadas a manipular marcos de datos. Cuando decimos *manipular*, no nos referimos a ello en mal sentido, sino en el sentido de adaptar y transformar los datos en información útil que nos ayude a responder a preguntas concretas que nos queramos formular. Manipular quiere decir, por ejemplo, cambiar el orden de las filas de un marco de datos sobre la base de un criterio determinado, seleccionar una parte de las filas o de las columnas o bien crear columnas con información nueva a partir de datos ya existentes. Una vez hayamos transformado los datos a nuestro gusto con *dplyr*, la librería *ggplot2* nos permitirá crear visualizaciones gráficas para poder comunicar de manera atractiva nuestros resultados.

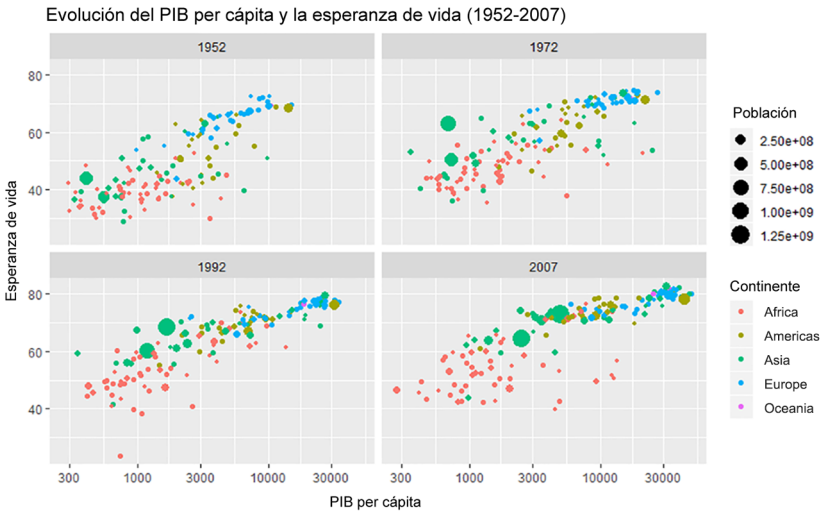
Para este proceso, tendremos que tener instalados y cargados en R *dplyr* y *ggplot2* y también el paquete *gapminder*, que utilizaremos para hacer los ejercicios en este módulo. Este paquete incluye un marco de datos con indicadores socioeconómicos como el PIB per cápita o la esperanza de vida en 142 países diferentes. Una vez tengáis los paquetes instalados y cargados, tecleando la función `search()` podéis comprobar que efectivamente *dplyr*, *ggplot2* y *gapminder* están cargados en el Global Environment de R. Instalaremos los paquetes con la función `install.packages()` y los cargaremos, por separado, con `library()`.

Para saber más

No os perdáis la página web de *gapminder* (www.gapminder.org), un portal educativo donde podréis analizar en línea una gran cantidad de datos. Tened en cuenta que cuando cargáis el paquete en R no lo visualizaréis en la ventana principal de Environment como si fuera un objeto que hayamos creado nosotros. No es necesario, pero si lo queréis visualizar en Environment, lo tendréis que crear mediante `gapminder <- gapminder`.

Este es probablemente el módulo que cambiará definitivamente nuestra percepción sobre R. Una vez finalizamos su lectura y hayamos trabajado los ejercicios, dominaremos los procesos básicos del análisis de datos. En las páginas siguientes utilizaremos el lenguaje de R de manera aplicada, dominaremos los instrumentos básicos para transformar y visualizar datos y seremos capaces de comunicarnos con el programa de manera fluida. Sin ir más lejos, al final del módulo habremos aprendido a construir visualizaciones como la figura 1, donde seremos capaces de agrupar en un solo gráfico hasta cinco variables. ¿Sois capaces de identificar cuáles son y cómo está representada cada una?

Figura 1. Visualización de un marco de datos tratado



Las cinco variables del gráfico. El PIB per cápita de cada país es la variable que está representada en el eje horizontal. En el eje vertical, encontramos la esperanza de vida. La tercera variable es el continente, donde cada categoría tiene asignado un color diferente. La población de cada país está representada por el tamaño de cada uno de los puntos, de forma que países con poca población tendrán un punto pequeño y países grandes tendrán un punto más grande. La quinta variable es el año. Hemos construido una parrilla de cuatro gráficos que representan cuatro años diferentes.

Fijaos en cuántas preguntas podemos llegar a responder en esta figura:

1) Hay una relación positiva entre el PIB per cápita y la esperanza de vida?
2) ¿En qué continente son más bajos la esperanza de vida y el PIB per cápita?
3) ¿En qué continente están los países con más población?
4) ¿A qué continente pertenecen la mayoría de los países con PIB per cápita y esperanza de vida alta?

Si os fijáis, estas preguntas las podemos contestar con una visualización rápida. También podemos apreciar otros detalles, como que en 1992 tenemos un caso extremo: un país africano con una esperanza de vida extremadamente baja. También vemos algunos casos, especialmente en África, pero también en Asia en 1952 y en 1972, de países relativamente ricos pero con una esperanza de vida muy baja. En cambio, no observamos que haya casos en el sentido contrario: países muy pobres pero con una esperanza de vida alta.

En este módulo, aprenderemos a hacer este proceso: observar y explorar un marco de datos, hacernos preguntas sobre los datos y cómo se relacionan entre ellos, tratarlos para poder responder a estas preguntas, elegir la mejor visualización para poder responder a estas preguntas y finalmente comunicarlas. La visualización nos puede ayudar a formular nuevas preguntas, que nos pueden llevar a investigar nuevos aspectos de los datos.

Interpretar números como e+08

Es bastante habitual en R encontrar números con notación científica que acostumbran a tener el formato e+. Interpretarlos es más fácil de lo que parece, puesto que simplemente tenemos que mover los decimales tantas veces a la derecha como nos indique el último número, si es positivo, o tantas veces a la izquierda si es negativo. Por ejemplo, 2.50+e08 se traducirá como 250.000.000.

1. Explorar

La exploración inicial de los datos es el primer paso que tenemos que hacer cuando tenemos delante un marco de datos (Babbie, 2013, pág. 90; King y otros, 1994). El objetivo principal consiste en entender, a grandes rasgos, cuáles son la composición y la estructura de nuestros datos. Nos interesa saber las dimensiones del marco de datos, el número de variables que hay, la tipología de estas variables y también nos puede interesar observar una pequeña muestra de los datos. Una vez hayamos hecho esta exploración general, el segundo paso consistirá en ver las características más sustantivas de cada una de las variables de interés de manera más específica.

Consultar el manual de ayuda

En la exploración inicial siempre es conveniente dar un vistazo a las instrucciones, el libro de códigos o el material asociado a las bases de datos. En el caso de los marcos de datos que provienen de paquetes de R podemos consultar la ayuda por medio, indistintamente, de `?gapminder` o bien de `help(gapminder)`. En este pequeño manual, *gapminder*, hay una descripción de las variables que nos permite, por ejemplo, ver una descripción de qué es y cuánto mide cada variable.

1.1. Exploración general

Como ya hemos incorporado a R el paquete *gapminder*, lo primero que haremos es una exploración inicial de los datos. Directamente, podemos imprimir el marco de datos `gapminder` para ver en la consola una idea inicial.

```
> gapminder
# A tibble: 1,704 x 6
   country     continent  year lifeExp      pop gdpPercap
   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.
 2 Afghanistan Asia       1957    30.3  9240934      821.
 3 Afghanistan Asia       1962    32.0 10267083      853.
 4 Afghanistan Asia       1967    34.0 11537966      836.
 5 Afghanistan Asia       1972    36.1 13079460      740.
 6 Afghanistan Asia       1977    38.4 14880372      786.
 7 Afghanistan Asia       1982    39.9 12881816      978.
 8 Afghanistan Asia       1987    40.8 13867957      852.
 9 Afghanistan Asia       1992    41.7 16317921      649.
10 Afghanistan Asia       1997    41.8 22227415      635.
# ... with 1,694 more rows
```

¿Qué observamos en esta tabla? En primer lugar, vemos que es un marco de datos en formato *tibble* que contiene 1.704 observaciones y 6 variables. Las variables, situadas en las columnas, son el país, el continente, el año, la esperanza de vida, la población y el PIB per cápita. Estas variables están formadas por tipos de vectores diferentes. Podemos comprobar que las variables *país* y *continente* están registradas como factores <*fctr*>, las variables *año* y *población* están registradas como vector entero <*int*> mientras que las dos restantes son un vector numérico o doble <*dbl*>.

En la dimensión vertical están las observaciones. En lo que nos tenemos que fijar en esta dimensión es en la **unidad de análisis** del marco de datos. Es decir, tenemos que comprobar qué mide cada una de las filas. Fijémonos en que, en este marco de datos, cada fila representa una combinación de país y año, de forma que todo el resto de variables nos muestran datos en relación con un país determinado en un año determinado.

Ver los nombres de cada columna

La función `names()` nos permite ver los nombres que toman las variables de un marco de datos. Solo hace falta que introduzcamos el nombre del marco de datos dentro del paréntesis. En marcos de datos muy anchos, con muchas columnas, podemos utilizar los corchetes para seleccionar una parte de las columnas que queramos visualizar. Por ejemplo, `names(gapminder)[2:4]` nos devolverá los nombres de la columna 2 a la 4.

También podemos explorar el marco de datos de una manera más ajustada con las funciones `head()` o `tail()`. La primera función muestra por defecto las seis primeras filas del marco de datos, mientras que la última nos muestra las seis últimas. Si, en lugar de seis, queremos ver una cantidad diferente de observaciones, podemos indicarlo añadiendo dentro de la función un nuevo argumento con el número de observaciones que queremos visualizar.

Ejemplo

Si queremos explorar las primeras 15 observaciones, introduciremos el código `head(gapminder, 15)`. Si queremos explorar las 10 últimas, escribiremos `tail(gapminder, 10)`.

Otras funciones que podemos utilizar para explorar de manera general los datos son `dim()`, para ver el número de filas y de columnas, o `str()` y `glimpse()`, que hacen tareas exploratorias parecidas.[1] Estas dos funciones son muy útiles para explorar marcos de datos con muchas variables, puesto que nos muestran las columnas verticalmente. En lugar de ver, como es habitual, las variables en el eje horizontal y las observaciones en el eje vertical, con estas funciones encontraremos las variables distribuidas en vertical y encabezadas por el símbolo del dólar (`$`). La razón por la cual se nos muestra así es porque, en la exploración inicial, nos interesará más bien observar todas las variables, pero solo algunas observaciones. Esta operación nos será más fácil si nos desplazamos en vertical por la consola, que no tiene límites para mostrarnos la información en formato vertical pero sí que nos tendría que cortar los datos si nos los muestra en horizontal.

[1] La función `glimpse()` pertenece al paquete *dplyr*, mientras que `str()` es una función de base de R. Podríamos considerar `glimpse()` como una versión avanzada de `str()`, puesto que nos devuelve una tabla más limpia y que se ajusta automáticamente a la anchura de nuestra consola. Intentad imprimir `gapminder` con las dos funciones para ver las diferencias visuales.

¡Los datos en la cabeza, no en la pantalla!

La manera de explorar marcos de datos con R funciona de modo diferente en comparación con otros programas como Excel o SPSS. En lugar de visualizar siempre el marco de datos en pantalla y desplazarnos de manera horizontal y vertical, R asume que cuando trabajamos con grandes bases de datos no necesitamos visualizar la tabla entera, sino tener una idea general de su contenido. Es por eso que nos tendremos que acostumbrar a tener la estructura de los datos ¡en la cabeza, no en la pantalla!

A continuación, hemos llamado con la función `str()` a la estructura de *gapminder*. Si el marco de datos tuviera 20 variables más sería más cómodo explorar los datos de este modo que en horizontal. Podemos observar, por ejemplo, cuántas categorías hay en las variables definidas como factores. La variable *country* tiene 142 países diferentes (nos indica que es un factor con 142 niveles), mientras que la variable *continent* tiene cinco continentes diferentes (cinco niveles).

```
> str(gapminder)
Clases 'tbl_df', 'tbl' and 'data.frame': 1704 obs. of  6 variables:
 $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 ...
 $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 ...
 $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
 $ pop      : int  8425333 9240934 10267083 11537966 ...
```

```
 $ gdpPercap: num  779 821 853 836 740 ...
```

Ahora ya tenemos una idea general de qué contiene el marco de datos *gapminder*. Durante la exploración general habrá variables que nos habrán llamado más la atención que otras y que querremos trabajar. O bien pensemos que puede haber observaciones (países y años concretos) que nos interesará más analizar.

1.2. Exploración específica

La exploración específica permite obtener más información del contenido de las variables del marco de datos que más nos han llamado la atención. Como ya sabéis, para llamar a una variable que se encuentra dentro de un marco de datos tenemos que escribir el nombre del marco de datos, seguido del símbolo `$` y a continuación el nombre del vector correspondiente. Si lo hacemos así, R nos devolverá todo el vector. Cuando trabajamos con grandes bases de datos los vectores acostumbran a ser larguísimos, de forma que si observamos todo el vector no podremos sacar información relevante. Probad, por ejemplo, el código siguiente:

```
gapminder$country
```

En lugar de llamar a la variable directamente, lo que haremos es pedir información más concreta mediante algunas funciones. En la exploración general hemos visto que la variable *country* tenía 142 países. Podríamos, por ejemplo, preguntarnos cuáles son estos países. O bien cuáles son los diferentes años en que los datos han sido capturados. La función `unique()` nos ayuda a obtener los valores únicos de una variable, de forma que nos será muy útil para variables categóricas e incluso para numéricas discretas. Así, `unique(gapminder$country)` nos devolverá un vector con los países únicos y `unique(gapminder$year)` nos devolverá todos los años de los que tenemos datos:

```
> unique(gapminder$year)
1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007
```

R nos devuelve un vector en el cual observamos que el primer año del cual tenemos datos es 1952 y el último año 2007. Entre estos dos periodos, tenemos datos cada cinco años. Podríamos combinar esta función con otras funciones que ya hemos visto anteriormente, como por ejemplo `head(unique(gapminder$year), 5)` para ver solo los primeros cinco años que aparecen en la base de datos. En el supuesto de que queramos saber solo los años más actuales, una opción sería añadir la función `sort()` y pedir que nos ordene los años en orden descendente y a continuación que nos enseñe

Contar casos

Si nos da pereza contar cuántos años únicos tenemos, podemos pedirle a R que nos los cuente por nosotros con la función `length()`. Si introducimos `length(unique(gapminder$year))` vemos que tenemos 12 años diferentes.

los cinco valores más altos: `head(sort(unique(gapminder$year), decreasing = TRUE), 5)`. Si eliminamos el argumento *decreasing* o bien lo pasamos a FALSE veremos los cinco valores más bajos de la variable.

Una función muy utilizada y que nos resultará muy útil para la exploración específica es `summary()`. Esta orden nos mostrará un resumen del objeto de interés. Esto quiere decir que podemos tanto pedir el sumario entero del marco de datos *gapminder* (nos devolverá un sumario de cada una de sus variables) como pedir el sumario de una variable específica. R nos ofrece un sumario diferente según el tipo de vector. En el código siguiente hemos pedido un sumario de *continent*, *lifeExp* y *gdpPercap*.

```
> summary(gapminder$continent)
Africa Americas     Asia   Europe  Oceania
   624      300      396      360       24

> summary(gapminder$lifeExp)
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
23.60   48.20   60.71   59.47   70.85   82.60

> summary(gapminder$gdpPercap)
  Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
241.2   1202.1   3531.8   7215.3   9325.5 113523.1
```

El `summary()` nos permite hacernos una idea de cómo están distribuidos los valores de la variable según el tipo de vector con que lo hayamos codificado. Fijaos en que de este modo podemos responder a preguntas más específicas como por ejemplo:.

«¿En qué continente tenemos más datos registrados?»
«¿Cuál es la esperanza de vida máxima de un país?»
«¿Y el PIB per cápita mínimo?»

También podemos hacer algunas apreciaciones interesantes con los datos que tenemos, como que prácticamente no hay diferencia entre la mediana y la media en la esperanza de vida mientras que sí que la hay en el PIB per cápita.

Sumario diferente según el tipo de vector

Cuando pedimos un sumario, R nos devolverá un resumen u otro según el tipo de vector. Si es un factor nos devolverá el número de frecuencias que tiene cada categoría. Si es un vector de carácter nos devolverá la longitud total de la columna y el tipo de vector. Si es una variable numérica o entera nos devolverá estadísticos descriptivos, como el valor mínimo, la mediana, la media o el valor máximo. En el supuesto de que haya datos perdidos, la función `summary()` nos devolverá una última columna con el número de datos perdidos que tiene el vector. En este caso, vemos que no hay datos perdidos.

En último lugar, en la exploración específica también podemos pedir la visualización gráfica de los datos mediante algunas funciones de los paquetes de base de R. Según el tipo de variable que queramos visualizar, utilizaremos una función diferente. Para una variable numérica, utilizaremos la función `hist()` para visualizar un histograma. Para las variables categóricas utilizaremos `plot()`, función que también nos irá bien para visualizar la interacción entre dos datos numéricos. Para visualizar la interacción entre una variable categórica y una numérica usaremos `boxplot()`. Probad de reproducir los códigos siguientes:

```
hist(gapminder$lifeExp)
```

```
hist(gapminder$gdpPercap)
plot(gapminder$continent)
plot(gapminder$gdpPercap, gapminder$lifeExp)
boxplot(gapminder$continent, gapminder$lifeExp)
```

En los dos primeros códigos hemos pedido un histograma porque es la manera de visualizar cómo están distribuidos los valores de variables numéricas, como la esperanza de vida y el PIB per cápita. Fijémonos en la columna más alta de la figura 2, que indica el intervalo que tiene más observaciones. Esta columna está situada entre 70 y 75 años en la esperanza de vida y por debajo los 10.000 dólares en el PIB per cápita (hay que tener en cuenta que las observaciones empiezan a partir de 1952 y, por lo tanto, nos encontraremos muchos países con un PIB per cápita muy bajo).

Figura 2. Visualización simple de un histograma

También observamos que la distribución de los valores en un histograma y el otro es completamente diferente. Mientras que los casos están distribuidos de manera suficientemente uniforme a lo largo del histograma de esperanza de vida, la mayoría de casos del PIB per cápita están situados en un extremo. Esto explica el motivo por el cual la media y la mediana son tan diferentes en el PIB per cápita y tan similares en la esperanza de vida. Veremos más detalles sobre estos estadísticos descriptivos más adelante, lo cual nos ayudará a entender mejor la diferencia entre la media y la mediana. Podéis probar de hacer una exploración específica de los otros gráficos que tenéis en el código anterior.

2. Transformar

A estas alturas, gracias a la exploración inicial, ya tenemos una imagen mental bastante clara de la estructura de nuestros datos. Ahora que sabemos qué variables tenemos, de qué tipo son y cómo están distribuidos los valores en cada una de ellas, es el momento de hacernos preguntas todavía más concretas sobre estos datos. Y por eso tendremos que aprender a transformar y manipular nuestros datos. Una pregunta podría ser, por ejemplo, cómo ha evolucionado la esperanza de vida de España a lo largo de los años. También nos podríamos preguntar qué diferencias hay entre la evolución del PIB per cápita entre España y Francia. O bien incluso podríamos crear datos nuevos a partir de los datos existentes.

En este apartado aprenderemos a transformar los datos con las funciones del paquete *dplyr* a partir de una sintaxis basada principalmente en seis verbos: filtrar, ordenar, seleccionar, mutar, resumir y agrupar, que resumimos en la tabla 1.

Tabla 1. Seis funciones básicas del paquete *dplyr*

Objetivo	Función	Acción
Manipular filas	`filter()`	Elimina filas.
	`arrange()`	Reordena las filas.
Manipular columnas	`select()`	Elimina o reordena las columnas.
	`mutate()`	Crea nuevas columnas o transforma existentes con valores construidos a partir de datos.
Manipular grupos de observaciones	`group_by()`	Agrupa observaciones.
	`summarize()`	Resume varias observaciones en una sola mediante una operación.

En las próximas páginas explicamos, una por una, cada una de estas funciones. Pero también tenemos que saber que sacaremos mucho mayor partido de ello si las aplicamos de manera simultánea. Combinar funciones es una tarea visualmente poco agradable, puesto que tenemos que poner funciones dentro de otras funciones, y es complicado leer un código entre tantos paréntesis. Si, siguiendo la tabla anterior, queremos filtrar unos datos, después crear una nueva columna y finalmente ordenar los datos, podemos o bien crear un código diferente para cada verbo o bien intentar añadir capas y capas de paréntesis en un solo código.

Obtener los datos de PIB total

Si nos fijamos bien en los datos que tenemos, veremos que podemos obtener el PIB total de cada país, puesto que disponemos de información del PIB per cápita y la población. Como sabéis, el PIB de un país es el conjunto de bienes y servicios producidos por todos los habitantes que viven en un país durante un año. Como tenemos los datos del número de habitantes que vive en un país en un año determinado (`pop`) y lo que producen de media (`gdpPercap`), multiplicando estos dos datos podremos crear una nueva columna donde tendremos el PIB total.

Más sobre el paquete *dplyr*

Podéis ampliar los conocimientos sobre estas funciones consultando la sección *5, Data Transformation*, que encontraréis en el libro en línea (https://r4ds.had.co.nz/transform.html) *R for Data Science*.

Por suerte, hay otra opción para ahorrarnos todos estos procedimientos. Dentro del paquete *dplyr* tenemos la *pipe* (símbolo `%>%`), herramienta que nos permite aplicar sobre el mismo objeto varias funciones de una manera mucho más intuitiva y ordenada. Lo que hace *la pipe* es cambiar la gramática de las funciones, de forma que nos transforma una función de varios argumentos `f(x, y)` en la estructura `x %>% f(y)`. A primera vista, cuesta comprender su utilidad, pero a través de diferentes ejemplos iremos comprobando la funcionalidad. Para ilustrar una primera idea, en la tabla 2 observamos cómo las funciones `arrange()` y `filter()` se agrupan a la manera clásica y con la *pipe*. A la manera clásica, empezamos por la función y vamos poniendo los argumentos dentro del paréntesis. Con la *pipe*, ponemos el primer argumento antes de indicar la función. Esto nos permite encadenar funciones siempre que tengan el primer argumento en común. De este modo, todas las funciones de dentro de la *pipe* empezarán por el segundo argumento.

Más sobre *la pipe*

La *pipe* está diseñada originalmente en el paquete *magrittr*. Podéis encontrar más información en este enlace (https://magrittr.tidyverse.org/).

Tabla 2. La función *pipe* (`%>%`)

Clásica y *pipe*	`f(x, y) = x %>% f(y)`
Agrupación clásica	`arrange(filter(gapminder, continent == "Europe"))`

Agrupación con *pipe*	`gapminder %>%`
	`filter(continent == "Europe") %>%`
	`arrange(lifeExp)`

Podemos añadir tantas funciones como queramos a la *pipe*. Si en el ejemplo quisiéramos añadir una función nueva, colocaríamos otra *pipe* después de la función *arrange* y abriríamos una línea nueva con la función nueva correspondiente. Durante este módulo nos iremos dando cuenta poco a poco de la utilidad de la *pipe*. Por ahora, solo intentad retener que es una nueva manera de estructurar gramaticalmente un código en R.

Crear objetos para guardar las transformaciones

Es importante saber que, cuando transformamos un objeto con una función, esta no guarda los cambios en cuestión a menos que lo indiquemos expresamente. En la consola veremos cómo quedarían estos cambios, pero R en ningún caso nos modifica el objeto. Si queremos guardar las operaciones, tendremos que sobrescribir el objeto o crear uno nuevo con el símbolo `<-`. Por ejemplo, si queremos solo los datos de Alemania, podemos introducir `gapminder_germany <- filter(gapminder, country == "Germany")`. Por ahora, recomendamos no sobrescribir los cambios sobre el mismo marco de datos.

2.1. Filtrar

La función `filter()` nos permite filtrar un subconjunto de las filas de un marco de datos sobre la base de una o varias condiciones lógicas.[2] Supongamos que solo queremos ver las observaciones de un solo país, en este caso de Alemania. Escogemos "Germany", filtramos por este país y pedimos observar solo las primeras seis filas con `head()`.

[(2)]Hay que diferenciar esta función, que reduce o filtra las filas, de la función `select()`, que reduce o selecciona las columnas.

```
> gapminder %>%
    filter(country == "Germany") %>%
    head()
 # A tibble: 6 × 6
   country continent  year lifeExp      pop gdpPercap
    <fctr>    <fctr> <int>   <dbl>    <int>     <dbl>
 1 Germany    Europe  1952    67.5 69145952  7144.114
 2 Germany    Europe  1957    69.1 71019069 10187.827
 3 Germany    Europe  1962    70.3 73739117 12902.463
```

```
   4 Germany    Europe  1967    70.8 76368453 14745.626
   5 Germany    Europe  1972    71.0 78717088 18016.180
   6 Germany    Europe  1977    72.5 78160773 20512.921
```

En este ejemplo hemos filtrado solo por una condición lógica, pero la función `filter()` permite hacer filtros con varias condiciones. En la mayoría de los casos solo hará falta que pongamos una coma e introduzcamos un nuevo argumento con otra condición lógica. La coma hace de condición lógica AND, en que pedimos que nos devuelva solo los casos que cumplen las condiciones que hemos señalado. En la primera función del código siguiente pedimos a R que nos devuelva los casos que cumplan a la vez dos requisitos: que sean del año 1987 y que el PIB per cápita sea superior a los 25.000 dólares (conseguiremos el mismo resultado si en lugar de una coma ponemos el símbolo `&`).

Igual sencillo (=) e igual doble (==)

Como veis, estamos trabajando con dos tipos de igualdad. Para diferenciarlos, tenemos que pensar que el igual doble (==) solo lo utilizaremos cuando estemos comprobando las condiciones de un objeto, de forma que nos creará, implícita o explícitamente, un vector lógico. En cambio, el igual sencillo (=) nos hace asignaciones de valores o resultados, como veremos más adelante.

También podemos filtrar los datos de otras maneras, como con la condición lógica OR (símbolo `|`). Como vemos, en la segunda función del código estamos filtrando por los datos que cumplan cualquiera de los dos requisitos: solo con que los datos sean igual o superiores a 1982 o bien que el PIB per cápita sea superior a 25.000 dólares, ya se incluirá en la selección. Este filtro devolverá más casos que el anterior. En el tercer código hemos separado los requisitos por el símbolo `&`, pero esta vez pedimos que nos enseñe todos los años excepto 1982 (con el símbolo de negación `!=`) para aquellos países con un PIB per cápita inferior o igual a 25.000 dólares. En la última función hemos incluido el símbolo `%in%` para indicar que queremos filtrar por un determinado número de categorías, que especificamos dentro de un vector.

Filtrar los datos perdidos

R nos filtra automáticamente los datos perdidos que hay en el vector que indicamos dentro de la función `filter()`. En el supuesto de que queramos conservar los datos perdidos, lo podemos indicar adentro mismo de la función con `is.na()`, por ejemplo `filter(gdpPercap > 30000 | is.na(gdpPercap)`.

```
gapminder %>%
  filter(year == 1982, gdpPercap > 25000)


gapminder %>%
  filter(year >= 1982 | gdpPercap > 25000)


gapminder %>%
  filter(year != 1982 & gdpPercap <= 25000)


gapminder %>%
  filter(year %in% c(1982, 1987, 1992))
```

Tenemos que tener en cuenta, a la hora de filtrar, que los factores requieren una atención especial. Esto es debido al hecho de que cuando filtramos los datos por medio de una variable que es un factor, R elimina los valores que hemos indicado, aunque no elimina ningún nivel de la variable. Es decir, si en *gapminder* filtramos por el continente África, nos quedará un marco de datos con solo una sola categoría en la variable continente, pero en cambio si pedimos los niveles de la variable, veremos que continúa teniendo cinco niveles.

```
gap_afr <- gapminder %>%
```

```
              filter(continent == "Africa")


> unique(gap_afr$continent)
Africa
Levels: Africa Americas Asia Europe Oceania
```

Para eliminar los niveles vacíos tendremos que crear otra *pipe* después de la función `filter()` y añadir la función `droplevels()`. Hay que remarcar, finalmente, que podemos añadir tantas condiciones como queramos siempre y cuando no filtremos más de la cuenta, puesto que en este caso R nos devolverá un marco de datos sin ninguna observación.

2.2. Ordenar

La función `arrange()` simplemente ordena las filas de manera ascendente o descendente a partir de los valores de una columna determinada. Es, pues, probablemente, la función que tiene menos misterio de todos los verbos que estudiaremos de la librería *dplyr*. Antes que nada, ¿sabríais intuir cómo está ordenado el marco de datos *gapminder*?

Ejercicio 1. *Gapminder*

Imprimid el marco de datos *gapminder* e intentad deducir por qué columna está ordenado en primer lugar. ¿Los valores están ordenados en orden ascendente o descendente? ¿Y cuál es la columna que está ordenada en segundo lugar?

Si hemos hecho el intento, habremos podido comprobar que *gapminder* está ordenado primero por países por orden alfabético descendente, puesto que empieza por todas las observaciones que tiene de Afganistán, sigue por Albania, etc. En segundo lugar, el marco de datos está ordenado por año de manera ascendente, puesto que empieza con datos de 1952 y va subiendo hasta 2007. Si queremos cambiar cómo está ordenado *gapminder* y, por ejemplo, nos gustaría ver en primer lugar los países con la esperanza de vida más baja, teclearemos el código siguiente:

```
gapminder %>%
  arrange(lifeExp)
```

Vemos que la mayoría de casos registrados con la esperanza de vida más baja son del inicio de la recogida de los datos en 1952 y 1957. Los datos, sin embargo, de esperanza de vida más bajos que encontramos en toda la base de datos son los de Ruanda en 1992, en la época de la guerra civil y el genocidio. ¿Cómo sería si miramos los datos de esperanza de vida por el lado de los países que la tienen más alta? Pues tenemos que especificar que queremos que nos devuelva la tabla en orden descendente con la función `desc()`:

Ordenar los datos perdidos

Tanto sea en orden ascendente como descendente, la función `arrange()` siempre situará los valores perdidos de la variable que queremos ordenar al final del marco de datos.

```
gapminder %>%
  arrange(desc(lifeExp))
```

El país que ha registrado una esperanza de vida más alta es Japón, con 82,6 años en 2007, seguido de Hong Kong. Como vemos, el tercer país de la tabla vuelve a ser Japón, de forma que el hecho de que tengamos mezcladas esperanzas de vida de años diferentes hace que tengamos duplicados países en la tabla.

En este punto, llega la parte más creativa de *dplyr* y donde podremos empezar a explotar realmente las virtudes de la *pipe*, puesto que gracias al símbolo `%>%` podremos trabajar con varias funciones a la vez. Supongamos que solo nos interesa ver los datos más recientes de países que tienen la esperanza de vida alta:

1) Primero, filtramos las observaciones que cumplan los dos requisitos siguientes: que sean del año 2007 y que tengan una esperanza de vida superior a los 78 años.

2) A continuación, creamos una nueva *pipe* y pedimos que nos ordene los resultados por esperanza de vida en orden descendente.

3) Como podemos establecer un orden por tantas columnas como queramos, pediremos que, en caso de tener dos países con la misma esperanza de vida, primero nos muestre el que tiene un PIB per cápita superior.

```
gapminder %>%
  filter(year == 2007, lifeExp > 78) %>%
  arrange(desc(lifeExp), desc(gdpPercap))
```

Como veis, mediante la *pipe* hemos encadenado dos funciones:

1) Primero, hemos indicado el argumento *x*, en este caso el objeto *gapminder*, seguido de la primera *pipe*.

2) En segundo lugar, hemos introducido la primera función con los argumentos correspondientes seguido de otra *pipe* y la segunda función con sus argumentos correspondientes.

2.3. Seleccionar

Hasta ahora, hemos visto dos funciones del paquete *dplyr* que transforman las filas (las filtran o las ordenan). La próxima función es `select()`, una función parecida a `filter()` pero que, en lugar de reducir el número de filas, lo que hace es reducir el número de columnas. Esta función es especialmente útil cuando tenemos marcos de datos con un número enorme de variables. Para poder trabajar mejor el marco de datos, nos interesará reducir las columnas a una cantidad más manejable.

Distinguir entre filtrar y seleccionar

Para distinguir mejor una función de la otra, diremos que filtramos por filas mientras que seleccionamos por columnas. Cuando transformamos datos, veremos que utilizaremos mucho más a menudo la función `filter()` que `select()`.

El marco de datos *gapminder* no tiene un número excesivo de columnas, pero su número es suficiente para poder practicar los diferentes usos de la función `select()`.

1) El primer código que vemos a continuación nos devuelve un marco de datos con solo tres columnas: *country*, *year* y *lifeExp*.

2) En el segundo código, el símbolo negativo nos sirve para excluir una variable y, por lo tanto, nos devolverá todo el marco de datos excepto la columna *pop*.

3) En el tercer caso seleccionamos desde la variable *country* hasta la variable *year* y, además, también seleccionamos la variable *gdpPercap*.

4) En el último caso, hemos dado una utilidad un poco diferente a `select()`, puesto que lo único que hemos hecho es indicar un orden de columnas diferente al orden original.[3]

[3] También podemos hacer la selección por números en lugar de por nombres de variables. Por ejemplo, `select(1:3)` nos seleccionará de la primera a la tercera columna.

```
gapminder %>%
  select(country, year, lifeExp)

gapminder %>%
  select(-pop)

gapminder %>%
  select(country:year, gdpPercap)

gapminder %>%
  select(country, year, continent, gdpPercap, lifeExp, pop)
```

Dentro de `select()` podemos incluir funciones, algunas resumidas en la tabla 3, que nos ayudarán en la selección de las variables que queremos retener. Por ejemplo, si queremos cambiar de orden algunas columnas y situarlas al principio, pero mantener el resto, podemos utilizar `everything()` (por ejemplo, `select(gapminder, continent, year, everything())`).

Tabla 3. Funciones adicionales con `select()`

`everything()`	Empieza con unos caracteres determinados.
`starts_with()`	Empieza con unos caracteres determinados.
`ends_with()`	Acaba con unos caracteres determinados.
`contains()`	Contiene unos caracteres determinados.

2.4. Mutar

El paquete *dplyr* también incorpora la función `mutate()`, que permite transformar los valores de una variable o crear nuevas variables a partir de datos ya existentes. Cuando mutamos una columna, podemos escoger dos opciones como primer argumento:

1) si lo primero que especificamos dentro de la función es el nombre de la columna que queremos transformar, R nos sobrescribirá los valores en la misma columna;

2) si, en cambio, indicamos el nombre de una columna nueva que aún no existe en el marco de datos, R nos mantendrá la columna original y nos creará una columna nueva con la transformación que hemos indicado.

Veamos un ejemplo de cada una y empecemos por la primera opción, que implica transformar los datos de una columna. En la variable *pop* de *gapminder*, los datos contienen muchos números. Supongamos que nos gustaría visualizar los datos de una manera más simple y tener las cifras de la variable en millones. En otras palabras, que si un país tiene 7.500.000 de habitantes pasáramos a ver este valor en 7,5 (en R el separador será un punto, no una coma). Lo que tendremos que hacer es lo siguiente:

```
gapminder %>%
  mutate(pop = pop / 1000000)
```

Como, en este primer caso, indicamos en la función que *pop* = *pop*, R entiende que tiene que sustituir los datos originales de la columna *pop* por los nuevos datos que hemos creado con la operación. Con la tabla que genera el código anterior veréis cómo R ha dividido entre un millón todos los valores de la columna *pop*, de forma que ahora vemos los datos en millones. En este proceso, por lo tanto, R no ha creado ninguna variable.

Si lo que queremos es mantener la columna original y crear una variable nueva a partir de la transformación mediante `mutate()`, indicaremos como primer argumento un nombre diferente en cualquiera de las columnas que conforman el marco de datos. En este ejemplo, crearemos la variable `gdp`, que indica el PIB total de cada país. Como en el marco de datos tenemos los datos de población y los de PIB per cápita, para saber el PIB solo hará falta que multipliquemos las variables *gdpPercap* y *pop*. Como ninguna otra columna del marco de datos tiene el nombre de *gdp*, R interpretará que tiene que incorporar una nueva variable en el marco de datos que hasta ahora no existía.

```
> gapminder %>%
    mutate(gdp = gdpPercap * pop) %>%
    head()
# A tibble: 6 x 7
```

```
  country     continent  year lifeExp      pop gdpPercap          gdp
  <chr>       <fct>     <int>   <dbl>    <int>     <dbl>        <dbl>
1 Afghanistan Asia       1952    28.8  8425333      779.  6567086330.
2 Afghanistan Asia       1957    30.3  9240934      821.  7585448670.
3 Afghanistan Asia       1962    32.0 10267083      853.  8758855797.
4 Afghanistan Asia       1967    34.0 11537966      836.  9648014150.
5 Afghanistan Asia       1972    36.1 13079460      740.  9678553274.
6 Afghanistan Asia       1977    38.4 14880372      786. 11697659231.
```

Calcular porcentajes

Anotad bien la fórmula `x / sum(x)`, puesto que nos puede ser de mucha utilidad más adelante para calcular proporciones con nuestros datos.

Exprimimos algo más las funciones de *dplyr* y las *pipe* con todo lo que hemos aprendido hasta ahora. En el código siguiente, hemos filtrado por el año 1952 y el continente asiático. Como solo nos quedará una sola categoría en las variables *año* y *continent*, hemos eliminado estas columnas con la función `select()`. Seguidamente, hemos mutado los datos, primero creando la variable `gdp`, y después hemos calculado el porcentaje sobre el PIB total de las filas filtradas con `gdp / sum(gdp)`. Finalmente, ordenaremos los resultados por la columna *gdp*, acabada de crear, en orden descendente, de forma que los valores más altos nos queden en lo alto de la tabla. En este código hemos creado el objeto `gap_asia`. Una vez lo hayamos creado, imprimiremos el objeto para ver el resultado.

```
gap_asia <- gapminder %>%
    filter(year == 1952, continent == "Asia") %>%
    select(-continent, -year) %>%
    mutate(gdp = gdpPercap * pop,
           perc_gdp = gdp / sum(gdp)) %>%
    arrange(desc(gdp))
```

Tened en cuenta que el orden en que establezcamos las diferentes funciones puede determinar la tabla resultante. Si hubiéramos primero mutado y después filtrado los datos, la nueva columna de porcentajes nos hubiera calculado los porcentajes sobre el total de la base de datos, no sobre el total de las columnas filtradas. En cambio, como hemos mutado después de filtrar, la mutación nos lo ha hecho solo para los países del continente asiático en 1952. También podéis comprobar cómo podemos llamar a objetos acabados de crear. Por ejemplo, podemos ordenar por el PIB porque, justo antes, hemos creado esta columna con `mutate()`.

2.5. Resumir

La función `summarize()` nos resume en un solo valor los datos de una columna a partir de una determinada operación matemática. Su funcionalidad es parecida a `mutate()`, pero en lugar de trabajar en horizontal creando nuevas columnas, trabaja en vertical, transformando varios datos de una misma columna en una sola. En el objeto `gap_asia` tenemos una tabla con la esperanza de vida, la población, el PIB per cápita, el PIB y el porcentaje sobre el PIB del continente para los países del continente asiático en 1952. Ahora intentaremos pedir resúmenes de este marco de datos. Calcularemos:

Calcular proporciones y casos

Las funciones `mean()` y `sum()`, aparte de sacar la media y sumar, también son muy útiles para otras cosas. Si pedimos un vector lógico dentro de `mean()`, la función nos hará la media de los valores que son verdad y los que son falsos, devolviéndonos, al fin y al cabo, una proporción de qué porcentaje de observaciones son verdad. Si pedimos un vector lógico dentro de `sum()`, sumaremos los casos en que la operación lógica sea verdad.

1) el total de población en el continente,
2) la media del PIB per cápita,
3) la proporción de observaciones con un PIB per cápita superior a la media,
4) el número de países con una esperanza de vida superior a los 60 años y
5) el número de observaciones que tenemos en la muestra.

Dentro de la función `summarize()`, separados por comas, hemos creado cinco sumarios:

1) *tpop* nos suma la población de todos los países filtrados,
2) *mean_gdpcap* nos hace la media,
3) *cerca* nos calcula la proporción de países que se encuentran por encima de la media,
4) *esp* nos devuelve el número de países con esperanza de vida superior a los 60 años y
5) *count* recuenta las observaciones filtradas.

```
> gap_asia %>%
    summarize(tpop = sum(pop),
              mean_gdpcap = mean(gdpPercap, na.rm = TRUE),
              prop = mean(gdpPercap > mean_gdpcap),
              esp = sum(lifeExp > 60),
              count = n())
# A tibble: 1 x 5
        tpop mean_gdpcap   prop   esp count
       <int>       <dbl>  <dbl> <int> <int>
1 1395357351       5195. 0.0909     4    33
```

R nos ha devuelto un sumario para cada operación que hemos pedido. La población total era de unos 1.400 millones de habitantes en 1952 en Asia. La riqueza por habitante mediana era de unos 5.200 dólares y sabemos que solo un 9 por ciento de los países del continente estaban por encima de esta media. La función `n()`, sin nada dentro del paréntesis, ha hecho el recuento de 33 observaciones (en este caso, países asiáticos en 1952), de los cuales solo cuatro tenían una esperanza de vida superior a los 60 años.

Sacar datos perdidos del sumario

Si nos fijamos en el código de la media, veremos que hemos introducido el argumento `na.rm = TRUE` (significa *NA remove*, eliminar valores perdidos). Es importante conocer este argumento y utilizarlo cuando pedimos estadísticos descriptivos, puesto que, en cualquier operación, excluirá los datos perdidos cuando queramos obtener un sumario. Al contrario, por defecto, este argumento es falso (`na.rm = FALSE`) de forma que, en el caso de haber un solo dato perdido en la columna, R nos hubiera devuelto en la tabla el valor *NaN (Not a Number)*.

2.6. Agrupar

Los datos que nos ofrece `summarize()` son muy genéricos y para nosotros solo tienen un valor bastante significativo si podemos compararlos con otros datos. Supongamos que queremos resumir los datos del código anterior para cada continente. Una opción, lenta y farragosa, sería aplicar la función `filter()` para cada continente y generar un sumario diferente cada vez. Primero, filtraríamos por Asia y haríamos un sumario, después por África, después por Europa, etc. Así, después de un buen rato, acabaríamos teniendo los resúmenes por cada categoría. Una opción más rápida es mediante `group_by()`, que agrupa los resúmenes de `summarize()` según los valores de una variable categórica o una numérica discreta. Si agrupamos, por ejemplo, por la variable *año*, R nos devolverá un sumario para cada valor de la variable *año*.

La función `group_by()` por ella misma no tiene ningún efecto sobre la tabla, sino que se tiene que aplicar en combinación con otra función. Normalmente, la combinaremos con `summarize()`, pero también se puede combinar con las otras funciones de *dplyr* .

En el código siguiente aprovecharemos partes de los códigos que hemos pedido anteriormente, pero en lugar de filtrar por un continente determinado, pediremos a R que nos agrupe los sumarios por continente. Lo que haremos será poner la función `group_by()` con el nombre de la variable que queremos que nos agrupe los sumarios. R nos devuelve un sumario para cada continente.

Ejercicio 2. Agrupar por varias variables

Podemos agrupar con la función `group_by()` por varias variables. Intentad sacar el filtro del año y añadidlo al agrupamiento con `group_by(continent, year)` para que también agrupe la tabla a la vez por continente y por año. Así podréis ver una evolución temporal para cada continente. Probad también a poner los datos a la inversa para ver cómo cambia la distribución de la tabla: `group_by(year, continent)`.

```
> gapminder %>%
    filter(year == 1952) %>%
    group_by(continent) %>%
    summarize(tpop = sum(pop),
              mean_gdpcap = mean(gdpPercap, na.rm = TRUE),
              prop = mean(gdpPercap > mean_gdpcap),
              esp = sum(lifeExp > 60),
              count = n())
# A tibble: 5 x 6
```

```
  continent       tpop mean_gdpcap   prop   esp count
  <fct>          <int>       <dbl>  <dbl> <int> <int>
1 Africa     237640501       1253. 0.327      0    52
2 Americas   345152446       4079. 0.24       6    25
3 Asia      1395357351       5195. 0.0909     4    33
4 Europe     418120846       5661. 0.433     23    30
5 Oceania     10686006      10298. 0.5        2     2
```

La función `group_by()` nos hace una agrupación previa de los datos antes de que `summarize()` aplique los cálculos pertinentes. R nos ha devuelto un marco de datos en que cada observación, tal como hemos indicado, es un continente. Vemos que Asia era el continente con más población en 1952, mientras que el PIB per cápita en Oceanía era el más alto (aunque solo hay dos países en el recuento). Europa era el continente con más países con una esperanza de vida superior a los 60 años.

Tenemos que tener en cuenta que la función `group_by()` se aplica solo en variables discretas que contengan un número suficiente de observaciones en cada grupo (como país o continente, que son categóricas, pero también año, que también es discreta y tiene suficientes observaciones para cada valor de la variable). Si aplicamos la función a variables numéricas continuas, lo más probable es que nos acabe devolviendo el mismo marco de datos.

2.7. Recapitulando *dplyr*

Si tenemos una pregunta sobre unos determinados datos, lo más probable es que con R tengamos las herramientas para contestarla. Lo más complicado es saber qué herramientas tenemos que utilizar de entre la enorme cantidad de posibilidades que ofrece R. En esta sección hemos aprendido las funciones más esenciales del paquete *dplyr*, que nos ayudan a transformar los datos de múltiples maneras para responder a la mayoría de preguntas que nos formulamos. También hemos conocido cómo agrupar estas funciones mediante la *pipe*, una manera más intuitiva de hacer operaciones complejas con varias capas de funciones.

Para cerrar esta sección, os ofrecemos un ejemplo de cómo las seis funciones diferentes de *dplyr* que hemos estudiado se pueden combinar en un solo código. En primer lugar, informamos a R de cuál es nuestra *x* y abrimos una *pipe*. A continuación, filtramos por los datos de 2007 y mutamos los datos para crear la nueva columna *gdp*. Seguidamente, agrupamos los datos por continente y pedimos un sumario del país con más población, el país con menos población, la diferencia entre el país con más población y el país con menos población y un recuento. Para acabar, ordenamos los datos en orden descendente por la nueva columna *diff* que nos aparecerá con el sumario y seleccionamos un orden diferente de las columnas, de forma que nos aparezca primero el continente, después la variable *diff* y a continuación el resto de variables.

```
gapminder %>%
  filter(year == 2007) %>%
  mutate(gdp = gdpPercap * pop) %>%
  group_by(continent) %>%
  summarize(max_pop = max(pop),
            min_pop = min(pop),
            diff = max_pop - min_pop,
            count = n()) %>%
  arrange(desc(diff)) %>%
  select(continent, diff, everything())
```

Evidentemente, un código con más operaciones no es necesariamente un código mejor. Un buen código es simplemente aquel que nos ayuda a transformar los datos de la manera que los resultados respondan a nuestras preguntas. Y para hacerlo, normalmente no nos hará falta que utilicemos cada vez las seis funciones de *dplyr*.

Para saber más sobre *dplyr*

De ahora en adelante nos será de gran utilidad el resumen (https://github.com/rstudio/cheatsheets/raw/master/data-transformation.pdf) para transformar datos con *dplyr* que han preparado los programadores de RStudio. También nos será muy útil la función de ayuda que podemos activar en cualquier momento a través de la consola (por ejemplo, `?dplyr`, `?filter()`, `?mutate()`, etc.).

3. Visualizar

La visualización es una parte importantísima en el trabajo de un analista de datos, puesto que todo el trabajo llevado a cabo en el proceso de análisis también se tiene que poder comunicar por medio de gráficos atractivos y fáciles de entender. Esto quiere decir que parte de la información que hemos conseguido generar mediante varias funciones del paquete *dplyr* se puede quedar en nada si no somos capaces de transmitirla al gran público. Demasiada información en un gráfico, o un gráfico que no explica lo bastante bien los datos con que estamos trabajando, puede dificultar la legibilidad y la comprensión de todo nuestro trabajo. Es por eso que tenemos que ser especialmente cuidadosos a la hora de mostrar el producto final.

Seguramente, ya conocemos algunas funciones de base de R como `plot()` o `boxplot()`, que nos permiten visualizar los datos de una manera rápida y exploratoria. En esta sección aprenderemos a crear visualizaciones de los datos más atractivas, que nos sirvan no solamente para explorar sino también para comunicar y presentar nuestros hallazgos. El paquete que utilizaremos es *ggplot2*, un paquete de gráficos muy potente pensado para poder representar estéticamente los datos a partir de una gramática implícita formada por varias capas de sintaxis.[4]

[(4)]Esta gramática de capas está construida a partir de los fundamentos teóricos del libro de Leland Wilkinson (1999) *The Grammar of Graphics*, que describe y sistematiza las estructuras que operan en la construcción de gráficos en el análisis cuantitativo.

La estructura basada en diferentes capas del paquete *ggplot2* conlleva una gran ventaja para trabajar con código, puesto que nos permitirá aplicar siempre la misma estructura independientemente del gráfico que queramos visualizar. A continuación describiremos siete capas, aunque hay algunas más. Para crear un gráfico mediante *ggplot2* necesitamos siempre, como mínimo, las tres primeras capas. Lo primero que tendremos que hacer es indicar la función `ggplot()`. Dentro de esta función indicaremos las dos primeras capas.

1) En la primera capa indicaremos el marco de datos.

2) En la segunda capa los estéticos (cómo se dispondrán las variables en el gráfico).

3) Seguido del signo + indicaremos la tercera capa, que es la geometría (qué figura geométrica representará los datos).

La estructura básica de cualquier visualización con *ggplot2* la encontramos resumida en la tabla 4. Las primeras tres capas (base de datos, estéticos y geometría) serán siempre necesarias para producir una visualización, mientras que las otras cuatro capas que explicaremos en este módulo (*facet*, escalas, coordenadas y temas) son opcionales. Tenemos que tener en cuenta que, a medida que queramos crear gráficos más sofisticados, nos será más necesario dominar todas las capas de *ggplot2*.

Tabla 4. Estructura de capas *de ggplot2*

`ggplot(marco de datos, aes(x, y, otros estéticos)) +`
`geometría() +`
`facet() +`
`escalas() +`
`coordenadas() +`
`temas()`

Para saber más sobre *ggplot2*

Podéis ver un ejemplo de las posibilidades de *ggplot2* en este enlace (http://www.ggplot2-exts.org/gallery/) o bien en la página web oficial de Tidyverse (https://ggplot2.tidyverse.org/reference/). También podéis consultar los libros de Chang (2012) y Grolemund y Wickham (2016). Para trabajar con el paquete, nos será muy útil este resumen (https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf) preparado por los creadores de RStudio.

Errores comunes con *ggplot2*

Cuando trabajamos con *ggplot2*, también es muy frecuente el error de describir la *x* y la *y* del gráfico sin ponerlas al final de la función de estéticos (por ejemplo, `aes(x = var1, y=var2)`), como también el de no cerrar el segundo paréntesis después de estéticos. Recordad el hecho de empezar con `aes()` el argumento de los estéticos y que siempre habrá dos paréntesis que hay que cerrar al final de esta línea, el de `ggplot()` y el de `aes()`.

Antes de empezar a repasar la estructura de capas, es muy importante remarcar cuatro aspectos fundamentales de *ggplot2*:

1) Primero, fijaos en que el paquete se llama *ggplot2*, pero la función para crear gráficos es `ggplot()` sin el '2' final.

2) Segundo, la función `aes()` abre siempre la segunda capa y los estéticos se sitúan dentro del paréntesis.

3) Tercero, a partir de la segunda capa, *ggplot2* une las otras capas mediante el símbolo +. Olvidarnos de él es bastante frecuente y nos llevará a mensajes de error en la consola.

4) Finalmente, tened también en cuenta que, en muchas ocasiones, llamaremos a una geometría determinada pero no tendremos que poner nada dentro del paréntesis, como `geom_point()` o `geom_abline()`, puesto que estaremos utilizando implícitamente los argumentos por defecto asociados a la geometría.

3.1. Las primeras capas

3.1.1. Marco de datos

En la primera capa simplemente introducimos el nombre del marco de datos que queremos utilizar. Ningún misterio. Solo hay que tener en cuenta dos consideraciones:

1) La primera es que, antes de visualizar datos con *ggplot2*, normalmente trataremos previamente el marco de datos con paquetes como *dplyr*.

2) La segunda es que *ggplot2* se puede enlazar con la lógica de *pipes*. Por lo tanto, tal como mostramos al inicio de la sección anterior, podemos poner el marco de datos al comienzo del código, hacer las transformaciones pertinentes con *dplyr* y a continuación enlazar una *pipe* con la función `ggplot()`. Como vemos en el ejemplo siguiente, como ya hemos puesto el marco de datos al inicio del código, situaremos los estéticos como primer argumento. Recordad que a partir de aquí las capas irán conectadas mediante el símbolo +.

```
gapminder %>%
  filter(country == "Germany") %>%
  ggplot(aes(x = year, y = lifeExp)) +
  geom_line()
```

3.1.2. Estéticos

La segunda capa de sintaxis son los estéticos, representados por la función `aes()`, que indica la manera en que *ggplot2* dispone estéticamente cada una de las variables que queremos representar. La variable que en este apartado definimos como *x* estará representada en el eje horizontal y la que definimos como *y* en el eje vertical. También tenemos la opción de incluir variables adicionales con otros tipos de estéticos, como el color, la forma o el tamaño. Por ejemplo, fijaos en el código siguiente:

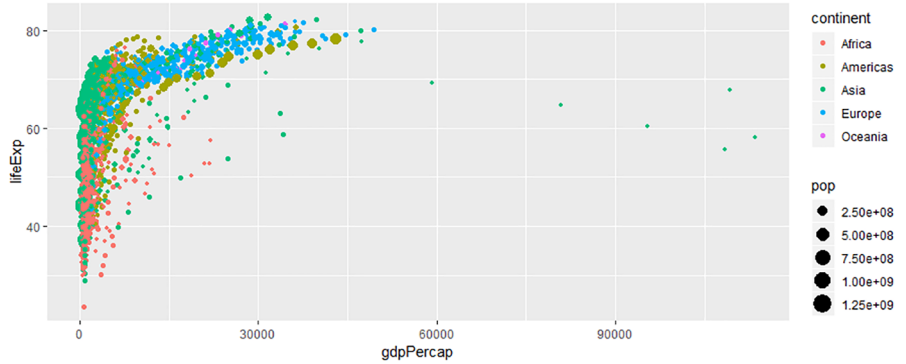
```
gapminder %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, color = continent, size = pop)) +
  geom_point()
```

Hemos representado este código en la figura 3. Como veis, hemos representado el PIB per cápita en el eje horizontal de las *x* y la esperanza de vida en el eje vertical de las *y*. Además de esta relación entre *x* e *y*, también especificamos que queremos visualizar dos dimensiones más: el color nos representará la variable continente (cada continente será un color), mientras que el tamaño nos representará la variable de población (el tamaño de la geometría variará según el valor de la población en cada caso). Cerraremos con un doble paréntesis e introduciremos el símbolo + para dar paso a la geometría, donde especificaremos a R que utilice un diagrama de dispersión para representar estos datos con la función `geom_point()`.

Variables dependiente e independiente

Normalmente, pondremos la variable que consideramos independiente o explicativa en el eje horizontal y la variable que consideramos dependiente o explicada en el eje vertical.

Figura 3. Visualización con cuatro estéticos



En esta figura hemos utilizado un tipo de gráfico llamado diagrama de dispersión, que es una buena manera de visualizar una relación cuando tenemos variables numéricas en los ejes horizontal y vertical. Si bien los estéticos *x* e *y* serán siempre los que nos marcarán el tipo de gráfico que utilizaremos para representar los datos, tenemos que saber que la utilización de otros estéticos como el color o la forma estarán muy determinados por el tipo de variable que queramos representar. El color, por ejemplo, funciona muy bien con variables categóricas, como es el caso de la variable *continent*. En el gráfico podemos ver que el estético *color* asigna automáticamente un color diferente a cada continente y dispone una leyenda en la parte derecha del gráfico. El tamaño (*size*) es más útil para representar variables continuas, de forma que un punto más grande nos representará un valor más alto de la variable. En la tabla 5 tenéis un resumen de los diferentes estéticos que podemos utilizar según el tipo de variable.

Tabla 5. Estéticos en función del tipo de variable

Estético	Código	Descripción
X	`x`	Eje horizontal.
Y	`y`	Eje vertical.
Color	`col / color`	Si la variable es categórica, nos dará un color diferente para cada categoría. Si es numérica, nos hará una gradación de color según el valor de la variable.
Tamaño	`size`	En variables numéricas, nos hará la geometría más grande o más pequeña (diámetros de los puntos, tamaño del texto o grosor de las líneas) según cada valor asociado.
Forma	`shape`	Restringido a variables categóricas. La representación por defecto es un punto, pero se puede representar con otras figuras como un triángulo, un cuadrado, una redonda, etc. Se tiene que introducir un valor del 1 al 25 (ved enlace (https://ggplot2.tidyverse.org/reference/scale_shape.html)). A los valores del 1 al 20 solo se les puede cambiar el color. Del 21 al 25 se les puede cambiar el color y llenarlos.
Llenar	`fill`	Llena las geometrías de forma 21 a 25, polígonos u otras figuras geométricas.
Transparencia	`alpha`	Modifica la transparencia de la geometría en un valor que varía de 0 a 1.

Qué estético utilizar

En general, los estéticos que funcionan mejor para variables categóricas son el color (puede representar cada punto o línea geométrica con un color diferente), la forma o el tipo de línea. En el caso de las variables numéricas, el tamaño y también el color en escala cromática (por ejemplo, azul intenso indica un valor más grande y azul menos intenso un valor más pequeño) nos representarán bien los diferentes valores.

Estético	Código	Descripción
Tipo de línea	`linetype / lty`	Modifica el tipo de línea. La que corresponde al valor 1 es continua y hasta 12 son tipos de líneas discontinuas.
Etiquetas	`labels`	Restringido a variables categóricas. Nos permite poner etiquetas en el gráfico.

Para utilizar los estéticos, pondremos el código del estético seguido de un símbolo igual y el nombre de la variable. Añadid, para hacer la prueba, `shape = continent` dentro de los estéticos del código anterior.

3.1.3. Geometría

La geometría es la tercera capa de la función `ggplot()`, separada de las dos primeras capas con el símbolo +. Aquí indicaremos el elemento visual que queremos usar para representar los datos: un histograma, un diagrama de cajas o, como en el ejemplo anterior donde tenemos dos variables numéricas, un diagrama de dispersión. La geometría está normalmente representada por *geom* seguido de una barra baja y del objeto geométrico en cuestión: `geom_point()` por el diagrama de dispersión, `geom_histogram()` por el histograma, `geom_bar()` por el diagrama de barras, etc.

Geometrías diferentes en *ggplot2*

El paquete *ggplot2* tiene hasta 37 geometrías diferentes. En este módulo, para introducir *ggplot2*, solo veremos más a fondo la geometría del diagrama de dispersión. En otros módulos de estadística descriptiva, podremos aprender otras representaciones gráficas como el diagrama de línea, el diagrama de barras o el histograma.

Todas las geometrías están asociadas con unas características por defecto. Por ejemplo, si queremos representar un punto en el gráfico, por defecto será de color negro y tendrá un tamaño determinado. Es por eso que muchas veces no nos habrá que especificar nada dentro del paréntesis de una geometría, puesto que las características por defecto asociadas a la geometría ya nos serán buenas. Si, en cambio, queremos cambiar las características por defecto de un determinado gráfico, como el color de una línea o el grueso de un punto, lo tendremos que indicar expresamente dentro del paréntesis. Si queremos, por ejemplo, un diagrama de dispersión con cuadrados azules, más grandes que su tamaño por defecto, y con un toque transparente, tendremos que cambiar la forma (*shape*), el color, el tamaño (*size*) y la transparencia (*alpha*).

Ejercicio 3. Probar combinaciones

Reproducid el código siguiente probando varias combinaciones de color ("red", "light blue", "green", etc.), tamaño (0.1, 1, 4 o 10), forma (0, 8, 15 o 24) y transparencia (0.1, 0.3 o 0.8).

```
gapminder %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, fill = continent)) +
  geom_point(color = "blue", size = 1, shape = 22, alpha = 0.6)
```

Tenemos que saber diferenciar entre los estéticos que representan una variable, que indicaremos normalmente en la segunda capa dentro de la función `aes()` y los estéticos que representan un atributo del gráfico, que indicaremos en la tercera capa dentro de la geometría. En el ejemplo anterior, *fill* es un estético que nos da información sobre la variable *continent*. Podemos hacer que *fill* pase a ser un atributo de la geometría si lo ponemos dentro de la geometría e indicamos, por ejemplo, con `fill = "yellow"`. En este caso, *fill* ya no nos da información de ninguna variable, sino que pasa a ser una manera de representar la geometría.

Nombres de colores

Podemos denominar los colores de centenares de maneras diferentes. En este web (http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf) tenemos centenares de nombres de colores que podemos asignar a *col o fill* .

También es importante retener que los estéticos pueden ir tanto en la función de `ggplot()` como dentro de la geometría. Poner los estéticos dentro de la geometría nos será útil cuando tengamos varias geometrías que tienen todos los estéticos coincidentes. Para intentar aclarar un poco esta distinción entre la capa de estéticos y la de geometría, y entre estéticos y atributos, fijaos en el código siguiente.

```
gapminder %>%
  ggplot(aes(x = log(gdpPercap), y = lifeExp)) +
  geom_point(aes(col = continent), alpha = 0.5) +
  geom_smooth(col = "red")
```

Dentro de `ggplot()` hemos asignado los estéticos generales del gráfico: todas las geometrías representarán en el eje de las *x* el logaritmo del PIB per cápita y en el eje de las *y* la esperanza de vida. Queremos visualizar este gráfico con dos geometrías: un diagrama de dispersión `geom_point()` y una línea de regresión local `geom_smooth()`. Hay, sin embargo, un estético, el color, que solo estará representado en la primera geometría y no afectará a la segunda geometría. Esto quiere decir que veremos puntos de color diferente según el continente que representen, pero no veremos cinco líneas de color diferente según cada continente. El estético del color, pues, lo tendremos que poner dentro de la función `geom_point()` con la función `aes()`. Fuera de los estéticos pediremos un atributo, que no representa ninguna variable, sino que es una característica de la geometría: todos los puntos tendrán un 50 por ciento de transparencia. Seguidamente, con el símbolo `+`, añadimos `geom_smooth()`, que cogerá los estéticos generales *x* e *y*, pero no el específico de color para cada continente. Sí que pediremos, en cambio, ver una línea de color rojo en lugar de la azul que nos mostraría por defecto. En la figura 4 vemos el resultado.

Figura 4. Diagrama de dispersión y línea de regresión local

Hasta ahora, solo hemos utilizado las tres primeras capas de *ggplot2*, suficientes para crear gráficos de todo tipo. Hay, sin embargo, otras capas que nos pueden ayudar a ampliar el número de variables que queremos representar, definir los títulos que daremos a cada una de las variables o cambiar otros aspectos visuales, como el tipo de letra o el color de fondo.

3.2. Otras capas

3.2.1. Facet

El nombre de la cuarta capa, *facet*, se podría traducir como aspecto o dimensión de un objeto. Es precisamente lo que hace esta función. La capa *facet* permite construir múltiples gráficos a partir de las diversas dimensiones de una variable categórica (Tufte, 1983). Por ejemplo, podemos ver la relación entre el PIB per cápita y la esperanza de vida según el continente.

```
gapminder %>%
  ggplot(aes(x = gdpPercap, y = lifeExp)) +
  geom_point(aes(col = continent), alpha = 0.3, show.legend = FALSE) +
  geom_smooth(col = "dark red") +
  facet_wrap(. ~ continent, scales = "free")
```

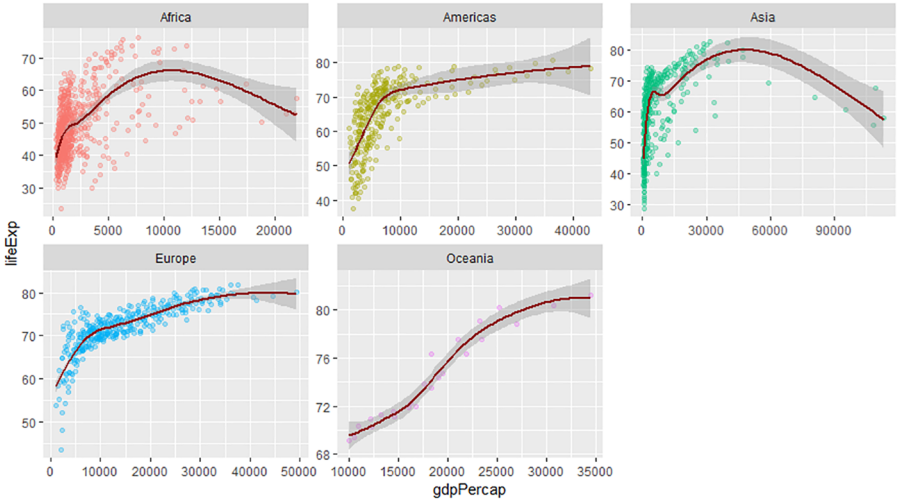
Dentro de la función de *facet* siempre utilizaremos la virgulilla, el símbolo ~, donde indicaremos si dividimos los gráficos por columnas o por filas. Si, como en el ejemplo, la variable categórica está situada a la derecha de la virgulilla, construiremos los *facets* por columnas. Si está a la izquierda, construiremos los *facets* por filas. En el lado contrario, pondremos un punto. Fijaos en que hemos mantenido el estético de color para el diagrama de dispersión, de forma que los puntos de cada *facet* salen pintados de un color diferente. Hemos decidido esconder la leyenda de los estéticos del diagrama con `show.legend = FALSE` porque era redundante. Por defecto, los ejes horizontal y vertical con la función *facet* nos aparecen con la misma escala, pero en nuestro caso hemos

La virgulilla (símbolo ~)

El símbolo ~ se denomina virgulilla y es una marca gráfica que en R significa: «explicado por». Encontraremos este símbolo en otras funciones de R como `lm()` o `case_when()`.

indicado que las escalas sean libres con `scales = "free"` de forma que cada gráfico tiene una escala diferente según sus valores (por ejemplo, en Oceanía el PIB per cápita llega a 35.000, mientras que en Asia sobrepasa los 90.000).

Figura 5. PIB per cápita y esperanza de vida en *facets* por continentes



Otra variante de *facet* es `facet_grid()`. A diferencia de `facet_wrap()`, en este caso tenemos la opción de construir una parrilla de *facets* que compartan etiquetas y ejes de coordenadas. Esta función es muy útil cuando queremos añadir una variable categórica al *facet* de las filas y una variable categórica al *facet* de columnas. Como veis en el gráfico siguiente, hemos creado un `facet_grid()` y hemos construido una parrilla con los años en las filas y el continente en las columnas.

```
gapminder %>%
  filter(year %in% c(2002, 2007)) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp)) +
  geom_point(aes(col = continent), alpha = 0.3, show.legend = FALSE) +
  geom_smooth(col = "dark red") +
  facet_grid(year ~ continent)
```

Si comparáis las dos *facets*, veréis que `facet_wrap()` utiliza etiquetas y coordenadas en cada uno de los gráficos, mientras que `facet_grid()` construye una parrilla con un número muy reducido de etiquetas y coordenadas.

Estilos de *facet*

En este enlace (https://ggplot2.tidyverse.org/reference/facet_grid.html) podéis ver otros ejemplos de su funcionamiento.

3.2.2. Escalas

Por escala (*scale*) nos referimos a la manera de representar, modular y mostrar información de los diferentes estéticos del gráfico, en particular de los ejes horizontal y vertical. Siempre introduciremos las escalas con el mismo orden: primero, pondremos el nombre *scale*, seguido de guion bajo, el estético, otro guion bajo y el nombre de la escala. Por ejemplo, en el código siguiente modulamos y mostramos nueva información sobre los ejes *x* e *y*. En primer lugar,

indicamos el código con el eje y el tipo de variable (*continuous* si es numérica y *discrete* si es categórica). Dentro de la función, tenemos varios argumentos para introducir información o modular el gráfico.

```
gapminder %>%
  ggplot(aes(x = gdpPercap, y = lifeExp)) +
  geom_point(aes(col = continent), alpha = 0.3) +
  geom_smooth(col = "dark red", se = FALSE) +
  scale_x_continuous(name = "PIB per cápita",
                     expand = c(0,0)) +
  scale_y_continuous(name = "Esperanza de vida",
                       limits = c(20, 80),
                       breaks = c(20, 40, 60, 80),
                       label = c("Jóvenes", "Adultos",
                               "Mayores", "Viejos")) +
  ggtitle("PIB per cápita y esperanza de vida")
```

En este código hemos modificado primero las escalas del eje de las *x* indicando que es una variable numérica con `scale_x_continuous()`. Dentro de la función, hemos especificado los límites inferior y superior del gráfico, así como "PIB per cápita" como título de eje con *name*. También con *name*, hemos denominado el eje vertical "Esperanza de vida" y hemos establecido los límites superior e inferior en 20 y 80 años con el argumento límites. A continuación, hemos cortado el eje en cuatro partes con *breaks*, y con *label* hemos asignado el nombre de las etiquetas a cada corte. Finalmente, hemos asignado un título general al gráfico con la función `ggtitle()`.

Hay muchas maneras de modificar las escalas del gráfico. Aquí mostramos cómo establecer los límites del gráfico y cómo indicar las etiquetas:

1) Establecer los límites del gráfico: con el argumento `expand = c(0,0)` conseguimos que los límites de un eje se ajusten el máximo posible a los datos. También dentro de las funciones *scale* podemos utilizar el argumento *límites*, donde indicaremos los límites inferior y superior. Fuera de las funciones *scale*, también podemos establecer los límites con `xlim()` e `ylim()`, por ejemplo `xlim(20, 85)` para la esperanza de vida. Estas dos funciones son una manera más rápida de marcar los límites del gráfico, pero solo funcionan cuando no utilizamos las funciones *scale*. Tenemos que ir con cuidado con las consecuencias de cambiar los límites del gráfico, puesto que R nos puede eliminar parte de la información que mostrará visualmente.

Para saber más

Encontraréis más información en este enlace para variables numéricas (https://ggplot2.tidyverse.org/reference/scale_continuous.html) y en este enlace (https://ggplot2.tidyverse.org/reference/scale_discrete.html) para variables categóricas. También podemos definir las escalas del resto de estéticos como el color, el tamaño o la forma.

2) Establecer las etiquetas del gráfico: podemos indicar las etiquetas del gráfico tanto dentro como fuera de las funciones *scale*. Una manera más rápida de definir el nombre de las escalas es con la función `labs()`. Por ejemplo, `labs(x`

= "Nombre x", y = "Nombre y", col = "Nombre Color"). Del mismo modo, también podemos eliminar los nombres de las escalas con labs(x = NULL, y = NULL).

Paletas de colores

En la escala es donde también indicamos los colores que nos representarán los estéticos. Probad a añadir la escala siguiente en el gráfico anterior:

```
scale_color_brewer(palette = 1, direction = -1, type = "div")
```

En esta función estamos diciendo que muestre los colores de la paleta 1. Podéis ir cambiando el número y veréis varias paletas, que la dirección de los colores sea a la inversa (podéis indicar 1 o -1) y que muestre colores divergentes. Si las categorías son ordinales es mejor utilizar diferentes tonos de un mismo color, que conseguiremos con type = "div". Es posible que queramos colores completamente diferentes, que indicaremos con type = "seq". Otra opción que tenemos es eliminar *direction* y *type* e indicar en la paleta una de las combinaciones que R tiene disponibles, como "PiYG", "Spectral", "Set2" o "Purples". Podéis consultar más colores en ?scale_fill_brewer o ?brewer.pal, o bien en el web de Color Brewer (http://colorbrewer2.org/#type=sequential&scheme=BuGn&n=3).

En las escalas también podemos modular las unidades con las cuales visualizamos la información. Esto nos puede ser útil en el caso de tener variables numéricas con casos extremos o valores sesgados hacia un lado de la distribución. Una deformación típica de los datos es la escala logarítmica, que transforma un eje de forma que cada cambio de unidad en el eje representa un cambio de *x* veces la unidad inicial. Por ejemplo, si añadimos scale_x_log10() estaremos multiplicando por 10 cada cambio de unidad en los datos originales.

Otras opciones de escala

Aparte de la escala logarítmica, también podemos convertir la escala en la raíz cuadrada de la variable con scale_x_sqrt() o cambiar la variable con scale_x_reverse().

3.2.3. Coordenadas

Esta capa controla las dimensiones del gráfico. Hasta ahora, todos los gráficos que hemos visualizado tienen dos dimensiones y se denominan coordenadas cartesianas. Una dimensión es horizontal, representada por el eje de las *x*, y la otra la vertical, representada por el eje de las *y*. Hay, sin embargo, otros tipos de coordenadas como por ejemplo las coordenadas polares, que nos posibilitan visualizar diagramas circulares como el del código siguiente:

Girar las coordenadas

Esta capa la usaremos muy poco. Quizás la función más útil es coord_flip(), que intercambia las dimensiones cartesianas del gráfico de forma que *x* pasa a ser representada en el eje vertical e *y* pasa a ser representada en el eje horizontal.

```
gapminder %>%
  filter(year == 2007) %>%
  group_by(continent) %>%
  summarize(population = sum(pop, rm.na = TRUE)) %>%
  ggplot(aes(x = 1, y = population, fill = continent)) +
  geom_col() + coord_polar(theta = "y")
```

Hay que decir que el diagrama circular no es muy recomendable, puesto que es difícil evaluar y comparar el tamaño relativo de cada segmento. Es decir, entre dos segmentos de tamaño parecido pero diferente será complicado identificar cuál es más grande. Si miramos la visualización del código anterior, no podemos saber si hay más población en África o América. En cambio, con un diagrama de barras son más fáciles de apreciar estas pequeñas diferencias.

3.2.4. Tema

Toda la parte visual que no tiene que ver con los datos se introduce en la capa de tema. Aquí es donde podemos cambiar el color de fondo, el tipo de letra de cada elemento, el tamaño de las letras o cualquier otro elemento del gráfico.[5] En este manual os sugerimos posibles temas que R tiene prediseñados. Por defecto, R siempre os mostrará `theme_gray()`, pero podéis cambiarlo si añadís un nueva capa a la función `ggplot()`, por ejemplo, `theme_classic()`, `theme_bw()`, o `theme_dark()`. Podéis bajaros nuevos temas como el que utilizan *The Economist* o *The Wall Street Journal* mediante el paquete *ggthemes* cargando el paquete `library(ggthemes)`.

(5)Encontraréis más información de cómo cambiar cada uno de los elementos en la página de la función theme (https://www.rdocumentation.org/packages/ggplot2/versions/3.1.0/topics/theme).

Resumen

Con todo lo que hemos aprendido en este módulo ya estamos preparados para podernos enfrentar a cualquier base de datos. Primero, hemos aprendido las herramientas que tenemos para hacer una exploración inicial en un marco de datos, que nos han ayudado a prescindir de la visualización clásica de las bases de datos. Ahora ya no necesitaremos tener los datos visualmente disponibles en la pantalla, sino que con la exploración general hemos aprendido a hacernos una idea en mente de la estructura de los datos y las variables que son de nuestro interés. También nos hemos hecho una idea de las características de las variables más relevantes del marco de datos por medio de una exploración más específica y una visualización rápida de los datos.

Lo más importante, sin embargo, para un analista de datos es dominar la transformación de los datos con paquetes como *dplyr*. Mediante las seis funciones principales de este paquete hemos aprendido a hacer varias manipulaciones en el marco de datos *gapminder* para preparar los datos para una visualización posterior. Finalmente, también hemos visto una pequeña parte de las enormes posibilidades que nos ofrece el paquete *ggplot2*, que nos permite crear una gran cantidad de gráficos basados en un sistema de capas.

Con todo, ahora ya estamos preparados para crear el código que nos permite construir el gráfico que hemos visto en la figura 1 de este módulo.

```
gapminder %>%
  filter(country != "Kuwait",
         year %in% c(1952, 1972, 1992, 2007)) %>%
  ggplot(aes(x = gdpPercap, y = lifeExp, col = continent,
             size = pop)) +
  geom_point() +
  scale_x_log10() +
  facet_wrap(~ year) +
  labs(x = "PIB per cápita", y = "Esperanza de vida",
       col = "Continente", size = "Población") +
  ggtitle("Evolución del PIB per cápita y la esperanza de vida (1952-2007)")
```

Fijaos en que aquí tenemos hasta cinco dimensiones de los datos:

1) el PIB per cápita,
2) la esperanza de vida,
3) el continente,
4) la población y
5) el año.

Fijaos en que hemos sacado el país Kuwait porque se trataba de un caso extremo que nos distorsionaba la visualización de los datos. Con el símbolo `%in%` hemos filtrado cuatro valores de la variable año, que nos servirán para hacer un *facet* con *ggplot2*. Para una mejor visualización de los datos hemos escalado la *x* a logaritmo. Una buena manera de practicar es crear gráficos nuevos a partir de modificaciones de código que podéis hacer en los ejemplos de este módulo.

Ejercicios de autoevaluación

Para un mejor aprendizaje, intentad hacer mentalmente el máximo de ejercicios posible, sin utilizar R.

1. Teclea el código para imprimir el marco de datos siguiente.

```
worldbankdata
```

2. Queremos ver las últimas siete filas y las primeras siete columnas del marco de datos siguiente.

```
imfdata
```

3. ¿Cuál es la función de *dplyr* que devuelve un resultado parecido a esta función?

```
str()
```

4. Teclea el código para visualizar la variable siguiente del marco de datos `imfdata`.

```
regions
```

5. Queremos ver los nombres únicos, ordenados de la Z a la A, de la variable siguiente del marco de datos `imfdata`.

```
regions
```

6. Visualiza un histograma de la variable siguiente del marco de datos `eurostat`.

```
social_expenditure
```

7. Transforma la función siguiente en una *pipe*.

```
length(unique(vector_categoric))
```

8. Detecta el error de la función siguiente.

```
filter(wbdata, country = "Japan")
```

9. Filtra la variable siguiente para los años 1980, 1985 y 1990.

```
wbdata$year
```

10. Filtra por países que tengan bombas nucleares (vector lógico) o gasto militar superior al 4 por ciento del PIB (vector numérico, escala de 0 a 1).

```
militarydata$nuclear / militarydata$gasto_pib
```

11. Ordena el vector siguiente en orden descendente.

```
militarydata$gasto_pib
```

12. Selecciona las primeras tres columnas del marco de datos siguiente y todas las columnas que acaben con la palabra "europe".

```
tradedatabase
```

13. La columna siguiente está en escala de 0 a 1. Mútala para ver los datos en tanto por ciento.

```
militarydata$gasto_pib
```

14. ¿Qué funciones hacen falta para resumir el marco de datos siguiente según los valores de una variable categórica?

```
eurostat
```

15. Agrupa los datos para la primera variable y resume la suma de los valores de la segunda.

```
militarydata$nuclear / militarydata$gasto_pib
```

16. Pide un diagrama de dispersión con las variables siguientes como *x* e *y*.

```
militarydata$num_guerras / militarydata$gasto_pib
```

17. Cambia el color a rojo de la geometría siguiente.

```
geom_smooth()
```

18. Pon transparencia del 50 por ciento en la geometría siguiente.

```
geom_point()
```

19. Indica cuál sería el argumento para llenar la geometría siguiente de color amarillo.

```
geom_bar()
```

20. Construye un `facet_wrap` por filas con la variable categórica siguiente.

```
militarydata$nuclear
```

Solucionario

1. `worldbankdata`

2. `tail(imfdata, 7)[1:7]`

3. `glimpse()`

4. `imfdata$regions`

5. `sort(unique(imfdata$regions), decreasing = TRUE)`

6. `hist(eurostat$social_expenditure)`

7. `vector_categoric %>% unique() %>% length()`

8. El igual (=) tendría que ser doble (==)

9. `wbdata %>% filter(year %in% c(1980, 1985, 1990))`

10. `militarydata %>% filter(nuclear == TRUE | gasto_pib > 0.04)`

11. `militarydata %>% arrange(desc(gasto_pib))`

12. `tradedatabase %>% select(1:3, ends_with("europe"))`

13. `militarydata %>% mutate(gasto_pib = gasto_pib * 100)`

14. `group_by()` y `summarize()`

15. `militarydata %>% group_by(nuclear) %>% summarize(mean(gasto_pib))`

16. `militarydata %>% ggplot(aes(x = num_guerras, y = gasto_pib) + geom_point()`

17. `geom_smooth(col = "red")`

18. `geom_point(alpha = 0.5)`

19. `geom_bar(fill = "yellow")`

20. `facet_wrap(nuclear ~ .)`

Glosario

`%>%` Símbolo *pipe* que replica el primer argumento de una función como argumento de las funciones siguientes (*dplyr*).

`%in%` Condición lógica que indica una selección de categorías.

`arrange()` Reordena las filas de un marco de datos (*dplyr*).

`boxplot()` Visualiza en un diagrama de cajas la relación entre una variable categórica y una variable numérica.

`coord_polar()` Cambia las coordenadas cartesianas por coordenadas polares (*ggplot2*).

`dim()` Vemos las dimensiones del marco de datos, primero las filas y después las columnas.

`droplevels()` Elimina los niveles vacíos de un factor.

`ggtitle()` Muestra el título principal del gráfico (*ggplot2*).

`glimpse()` Muestra una estructura de los datos más limpia que `str()` y adaptada a las dimensiones de la consola (*dplyr*).

`facet_grid()` Crea una parrilla de *facets* (*ggplot2*).

`facet_wrap()` Crea una parrilla de *facets* reduciendo los ejes y las coordenadas (*ggplot2*).

`filter()` Elimina las filas de un marco de datos (*dplyr*).

`geom_****()` Capa *de ggplot2* que crea una geometría (*ggplot2*).

`ggplot()` Crea una visualización gráfica (*ggplot2*).

`group_by()` Agrupa observaciones de un marco de datos (*dplyr*).

`head()` Devuelve las seis primeras filas del marco de datos.

`hist()` Permite visualizar la distribución de una variable numérica.

`install.packages()` Instala librerías en R.

`labs()` Muestra las etiquetas del gráfico (*ggplot2*).

`library()` Muestra las librerías o carga una librería determinada.

`mutate()` Crea nuevas columnas o transforma existentes con valores construidos a partir de datos (*dplyr*).

`names()` Muestra el nombre de las columnas de un marco de datos.

`plot()` Permite visualizar las frecuencias de una variable categórica o bien visualizar la relación entre dos variables numéricas.

`unique()` Devuelve valores únicos de un vector.

`scale_*_****()` Prepara las escalas de un eje (*ggplot2*).

`search()` Muestra los paquetes cargados en R.

`select()` Elimina o reordena las columnas de un marco de datos (*dplyr*).

`summarize()` Resume varias observaciones en una sola mediante una operación (*dplyr*).

`summary()` Resume una variable y devuelve un sumario específico según cada tipo de vector.

`str()` Muestra la estructura de los datos, con las variables en las filas, el tipo de variable y las primeras observaciones de cada variable.

`tail()` Devuelve las seis últimas filas del marco de datos.

Bibliografía

Babbie, E. R. (2013). *The practice of social research*. Wadsworth: Cengage Learning.

Chang, W. (2012). *R Graphics Cookbook*. Canadá: O'Reilly. http://www.cookbook-r.com/graphs/

Grolemund, G.; Wikcham, H. (2016). *R for Data Science*. Canadá: O'Reilly. https://r4ds.had.co.nz/

King, G.; Keohane, R. O.; Verba, S. (1994). *Designing Social Inquiry: Scientific Inference in Qualitative Research*. Princeton: Princeton University Press.

Tufte, E. (1983). *Visualization of Quantitative Information*. Míchigan: Graphics Press.

Wilkinson, L. (1999). *The Grammar of Graphics*. Nueva York: Springer.