

A Minimal Book Example

Yihui Xie

2018-05-01

Contents

1	Prerequisites	5
2	Transformación-I	7
2.1	¿Qué aprendimos la clase pasada?	7
2.2	Esta clase:	7
2.3	Esta sección del curso se compone por dos clases. En la siguiente:	7
2.4	Conceptos preliminares:	7
2.5	El paquete dplyr (instalado con el tidyverse)	12
3	Transformación-II	19
3.1	¿Qué aprendimos la clase pasada?	19
3.2	El operador pipeline %>%	19
3.3	Miscelánea de funcionalidades avanzadas de transformación de datos	22
3.4	Datos faltantes	28
4	Funciones	33
4.1	¿Qué aprendimos la clase pasada?	33
4.2	Funciones en R	33
4.3	<i>Las funciones pueden llamarse a si mismas.</i>	39
4.4	<i>Las funciones pueden regresar funciones:</i>	40
5	Vectores	43
5.1	¿Qué aprendimos la clase pasada?	43
5.2	Vectores, listas y arreglos	43
5.3	Iteración en R	46
6	Modelado	59
6.1	¿Qué aprendimos la clase pasada?	59
6.2	Modelos	59

Chapter 1

Prerequisites

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```
install.packages("bookdown")  
# or the development version  
# devtools::install_github("rstudio/bookdown")
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading #.

To compile this example to PDF, you need XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): <https://yihui.name/tinytex/>.

Chapter 2

Transformación-I

2.1 ¿Qué aprendimos la clase pasada?

- Principios de visualización de datos utilizando el paquete ggplot2.
- Un primer ejemplo de manipulación de datos (la función filter).

2.2 Esta clase:

- Utilizar R como calculadora.
- Asignar objetos a variables para poder utilizarlos después.
- Estudiaremos la lógica de las funciones en R.
- Utilizar scripts como archivos de texto donde apuntamos nuestro código.
- Utilizaremos un paquete de R para manipulación de tablas de datos (**data frames**).

2.3 Esta sección del curso se compone por dos clases. En la siguiente:

- Aprenderemos a leer archivos de datos en nuestro espacio de trabajo de R.
- Seguiremos trabajando con diversas técnicas para transformar y manipular datos.

2.4 Conceptos preliminares:

2.4.1 R como una calculadora

En la clase pasada omitimos algunos elementos básicos de R para lograr que pudiésemos comenzar a graficar lo más pronto posible.

Uno de esos elementos es que R puede llevar a cabo operaciones como una calculadora. La consola de R entiende expresiones matemáticas y las puede operar para obtener sus resultados.

```
2+2
```

```
## [1] 4
```

```
2000*99^2-444
```

```
## [1] 19601556
```

Estas operaciones no se guardan en el espacio de trabajo. Al igual que cuando se usa una calculadora sin guardar un resultado en su memoria, al momento de producir el resultado se pierde la operación.

2.4.2 Asignación de objetos en R

Para almacenar un resultado en el espacio de trabajo de R es necesario asignarlo a un objeto. Las asignaciones en R tienen siempre la misma forma:

```
nombre_del_objeto <- valor
```

Por ejemplo puedes asignar una de las operaciones anteriores a un objeto en R.

```
operacion_1 <- 2+2
```

Escribir el nombre del objeto en la consola es equivalente a preguntarle a R ¿Qué hay en ese objeto?

```
operacion_1
```

```
## [1] 4
```

Es buena práctica ponerle nombres informativos a los objetos que estamos generando. Recomendamos usar la nomenclatura llamada **snake_case** que se refiere a separar las palabras del nombre de tu objeto con un “_”.

Construyamos otro objeto que guarda el resultado de una operación.

```
esta_operacion_es_sumamente_larga <- 2+2^21*2183176321/(22+2.56^121)+11111
```

RStudio tiene una herramienta de autocompletado de nombres, si escribes en la consola únicamente “esta” y aprietas la tecla Tab debe completar el nombre del objeto anterior.

2.4.3 Funciones en R

Los comandos que se utilizaron la clase pasada para graficar se llaman funciones. Las funciones en R tienen la forma:

```
nombre_de_la_función(argumento1=valor1,argumento2=valor2,...)
```

El nombre de la función indica a R qué función se debe ejecutar. Cada función tiene asociada una lista de argumentos que le permiten saber a R qué le estás pidiendo que haga.

Como un primer ejemplo, la instalación de paquetes en R se puede llevar a cabo usando una función: `install.packages()`. Lo mínimo que necesita R para poder instalar un paquete es cuál debe instalar. Por esto el uso de esta función es simplemente `install.packages(pkgs=“nombre_del_paquete”)`.

También debemos recordar que el símbolo “?” junto a una función nos permite acceder a la ayuda de R sobre la misma.

Estudiemos ahora la función `seq()`, pueden escribir `?seq` para ver la ayuda sobre esta función. Esta genera secuencias de números, por ejemplo para generar la secuencia de números enteros del 1 al 100 basta con escribir:

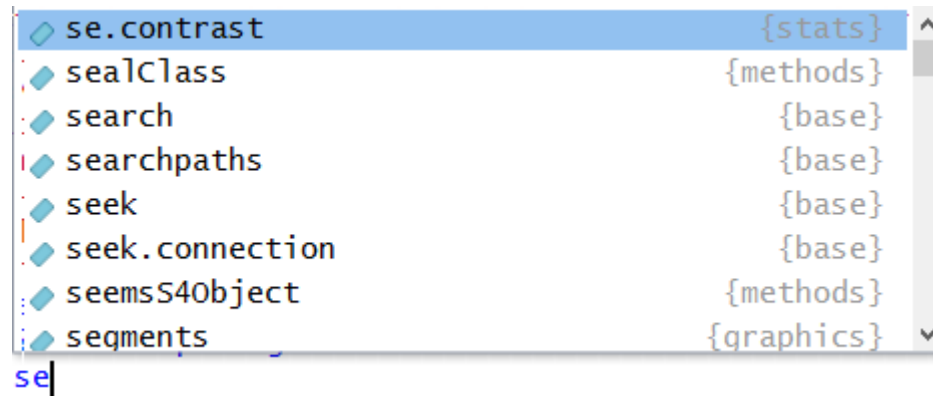
```
seq(from=1,to=100)
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
## [18] 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
```



```
## [35] 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
## [52] 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
## [69] 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85
## [86] 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

Una observación útil es que el autocompletado de RStudio ¡sirve también para funciones! Intenta escribir “se” en la consola y luego oprimir la tecla Tab. Deberá aparecer un recuadro con un listado de opciones.

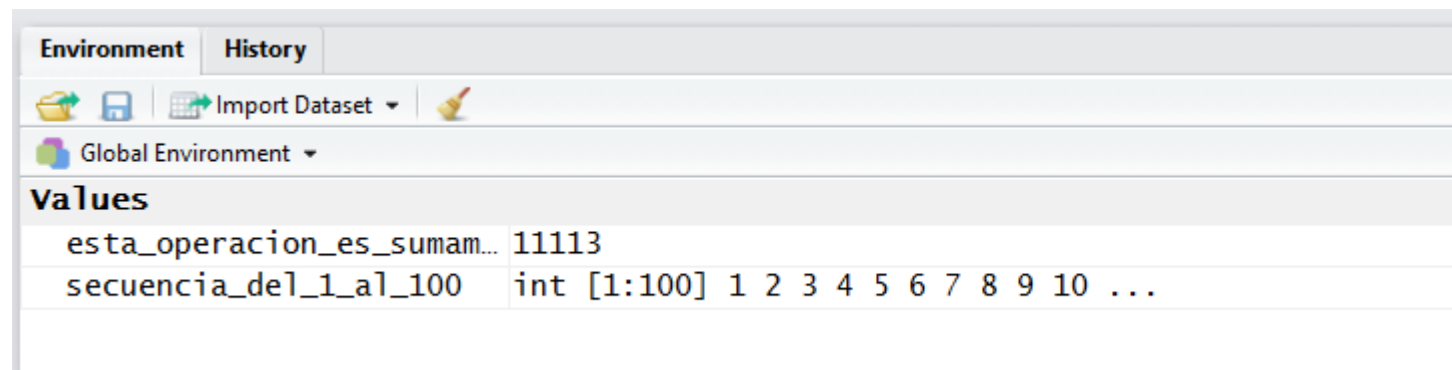


Escribir una letra más (e.g. “q”) reduce la lista de opciones y la función seq deberá ya ser visible en ellas. Luego basta con oprimir la tecla Enter para elegir la función seq.

El resultado de utilizar una función se puede también asignar a un objeto. Esto será de gran utilidad a la hora de trabajar en R.

```
secuencia_del_1_al_100 <- seq(from=1,to=100)
```

En la sección superior derecha de nuestro ambiente de RStudio ya deben existir dos objetos cargados en el espacio de trabajo: “esta_operacion_es_sumamente_larga” y “secuencia_del_1_al_100”. Estos ya están cargados en la memoria RAM de la computadora y R puede acceder a ellos cuando se les solicite utilizando su nombre.



¿Por qué no funciona el siguiente código? Modifícalo hasta que cada instrucción funcione.

```
library(tidyverse)
```

```
ggplot(dota = mpg) + geom_point(mapping = aes(x = displ, y = hwy))
```

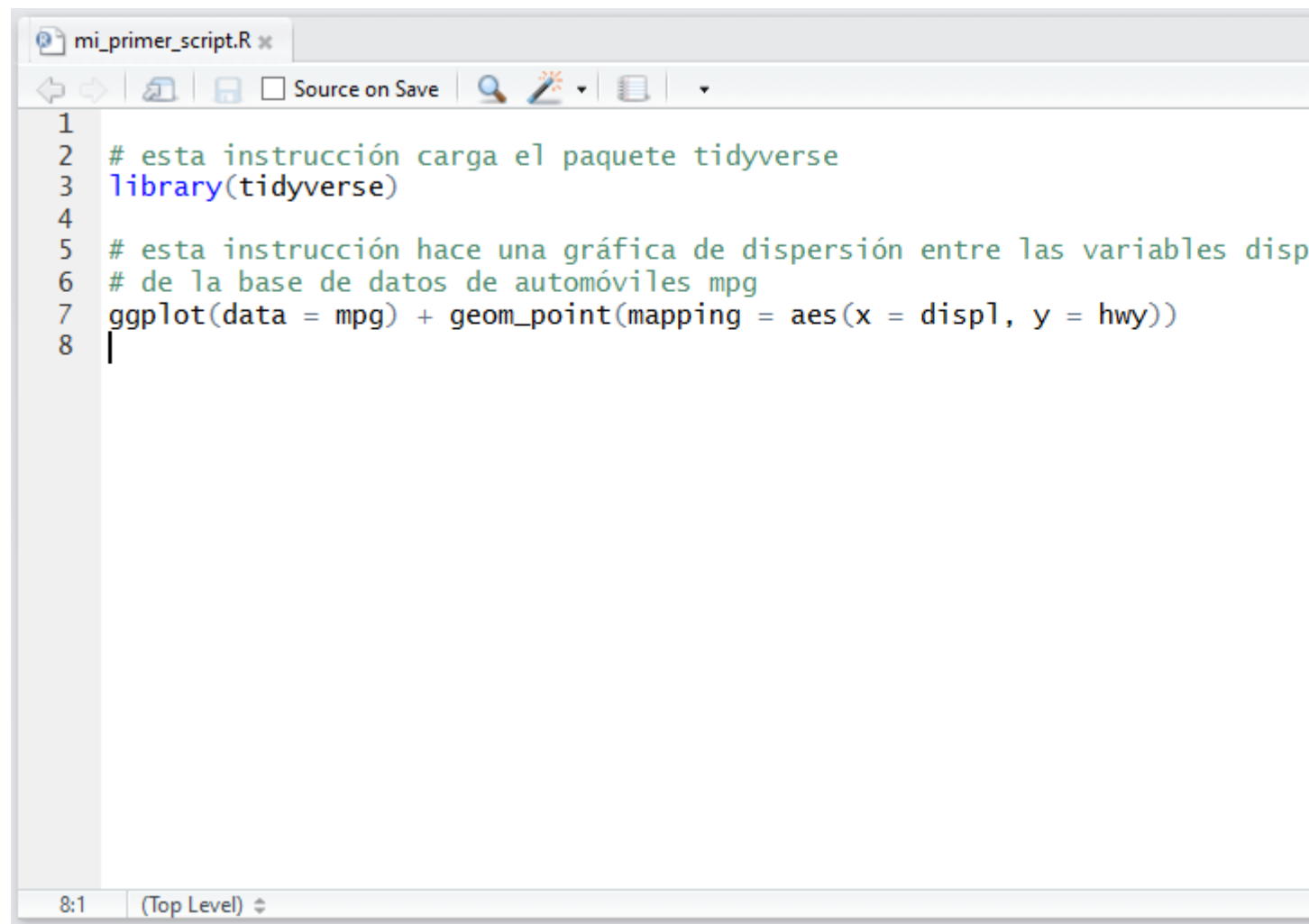
```
fliter(mpg, cyl = 8)
```

```
filter(diamond, carat > 3)
```

2.4.4 Scripts en R

Se puede escribir código de R en cualquier procesador de texto, por ejemplo en word. La conveniencia de tener un procesador de texto dentro de tu ambiente de trabajo (RStudio) es que te permite mandar las instrucciones directamente a la consola para su ejecución.

En un script el símbolo `#` le indica a R que esa línea es un comentario. Los comentarios no se ejecutan y sirven para tener notas para nosotros mismos sobre el código que estamos desarrollando. Cuando una línea de texto es un comentario, será de color verde.



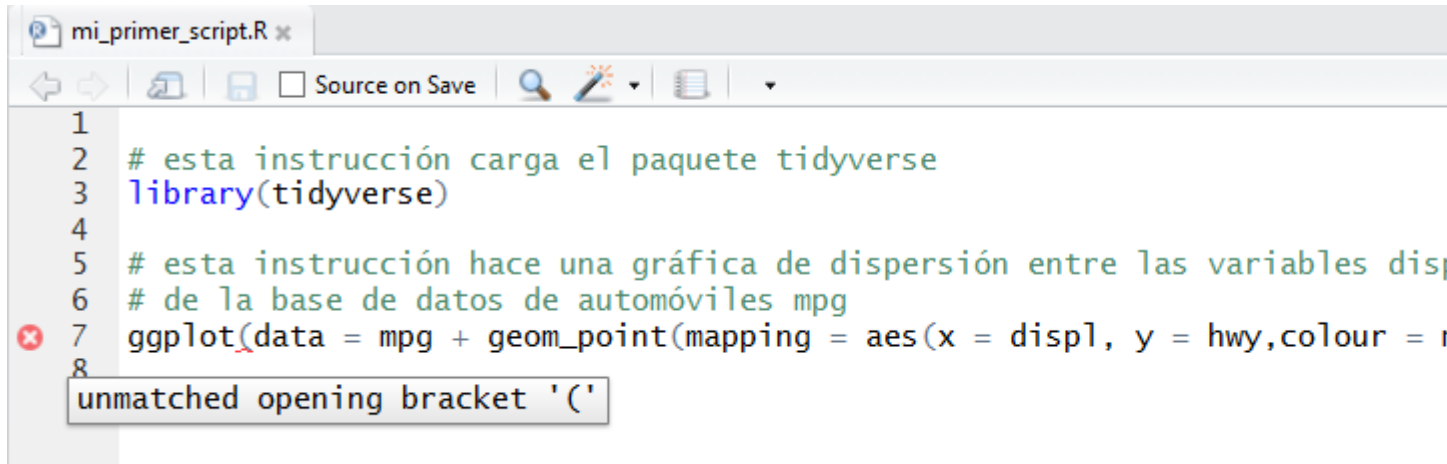
```
1
2 # esta instrucción carga el paquete tidyverse
3 library(tidyverse)
4
5 # esta instrucción hace una gráfica de dispersión entre las variables disp
6 # de la base de datos de automóviles mpg
7 ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy))
8 |
```

Para ejecutar el código que se encuentra en un script se pueden utilizar las teclas: `Ctrl/Cmd (mac) + Enter` o el botón “Run” que se encuentra en la esquina superior derecha de la ventana de scripts. En un script puedes correr una línea en particular (primero la eliges con el mouse, botón izquierdo). También puedes correr una selección de tu script, para esto sólo debes elegir una sección de tu código dejando apretado el botón derecho del mouse o con la tecla `shift` y las flechas del teclado; análogo a como se hace en word.

2.4.5 Diagnósticos de RStudio

Como se sabe que R es tan moléstamente quisquilloso, RStudio tiene integrado un detector de errores de sintaxis.

Un error tremendamente común es que siempre que se abran paréntesis, se deben cerrar `()`. Por ejemplo a la hora de usar alguna función. Si RStudio detecta que en una línea hay paréntesis sin pareja la marcará con una cruz roja.



Por supuesto existen muchísimos posibles errores en R. Si se comete alguno RStudio tratará de informarte sobre la naturaleza del error, basta con colocar el puntero del mouse sobre la cruz roja para ver esta nota.

2.4.6 Operadores relacionales

Sirven para comparar dos cantidades. Regresan **TRUE** si la comparación es cierta y **FALSE** en otro caso.

Ejemplos:

```
5 == 5 # Notar que se pone doble igualdad para comparar dos cantidades
```

```
## [1] TRUE
```

```
5 > 6
```

```
## [1] FALSE
```

```
5 < 6
```

```
## [1] TRUE
```

```
6 >= 3
```

```
## [1] TRUE
```

```
6 <= 3.4
```

```
## [1] FALSE
```

```
# Otro operador muy útil para ver si un elemento se encuentra en un vector
```

```
c(1, 2, 3) # Formando un vector que contiene los números 1, 2, 3
```

```
## [1] 1 2 3
```

```
5 %in% c(1, 2, 3)
```

```
## [1] FALSE
```

```
2 %in% c(1, 2, 3)
```

```
## [1] TRUE
```

2.4.7 Operadores booleanos

Sirven para comparar dos expresiones como las anteriores:

El operador **y** (&) regresa verdadero si las dos expresiones que recibe son verdaderas y falso en otro caso.

```
(5 > 6) & (7 < 8) # Y: notar que regresa false porque 5 > 6 es falso
```

```
## [1] FALSE
```

```
(5 < 6) & (7 < 8)
```

```
## [1] TRUE
```

```
(5 > 6) & (7 > 8)
```

```
## [1] FALSE
```

El operador **o** (|) regresa verdadero si **alguna** de las expresiones que recibe es verdadera y falso en otro caso.

```
(5 > 6) | (7 < 8) # O: notar que regresa true conque alguna de las expresiones sea verdadera
```

```
## [1] TRUE
```

```
(5 < 6) | (7 < 8)
```

```
## [1] TRUE
```

```
(5 > 6) | (7 > 8)
```

```
## [1] FALSE
```

El operador lógico **no** (!) regresa verdadero si la expresión que recibe es falsa y viceversa)

```
5 > 6
```

```
## [1] FALSE
```

```
!(5 > 6)
```

```
## [1] TRUE
```



Evalúa una a una las siguientes expresiones y explica por qué da TRUE o FALSE. Sugerencia: compréndelas una a una y en orden.

```
5 < 7
```

```
!(5 < 7)
```

```
5 > 6
```

```
6 <= 6
```

```
6 >= 6
```

```
!(5 < 7) & (5 > 6)
```

```
(!(5 < 7) & (5 > 6)) | (6 <= 6)
```

```
!(5 < 7) & ((5 > 6) | (6 <= 6))
```

2.5 El paquete dplyr (instalado con el tidyverse)

```
# Cargando el paquete
library("tidyverse")
```

Un data frame se compone de **registros** (renglones) y **campos o variables** (columnas):

Campos

Registros

Nombre Proyecto	Contacto	Correo	Teléfonos
Eagle	anell y adrian	eagleog@cantv.net	5212-2366602 - 4162800
TLC	Ambar Texeira	boutiquepventas2003@gmail.com	543-07-85
Tedexis	Luis Poggi	lpoggi@gmail.com	4141838624
FYC	Carlos Marrero	carlos.marrero@fyccorp.com	2327811
NGS camcaroni	Alfonso Mora	amora@ngs.com.ve	4148885289
Tecnoquim	Lenin Marques	grupotecnquim@gmail.com	7816287, 7828583, 0416 808-0738
Tuscomprasporinternet	Rafael Monserrat	bufetemontserrat@hotmail.com	4147368304

Figure 2.1:

Con **dplyr** se podrán realizar acciones muy útiles como las siguientes:

1. **Seleccionar** campos de un data frame
2. **Filtrar** registros de un data frame que cumplan cierta condición.
3. **Ordenar** registros de acuerdo a su valor en ciertos campos.
4. **Crear** nuevas columnas a partir de los valores de las preexistentes
5. **Calcular resúmenes** de **agregados** de datos (como las tablas dinámicas en Excel).
6. **Unir** tablas de acuerdo a sus valores en ciertos campos (se verá más adelante).

2.5.1 1. Seleccionar campos de un data frame: `select(df, columnas_a_seleccionar)`

```
# Usaremos el data frame diamonds cargado con "tidyverse"
glimpse(diamonds)
```

```
## Observations: 53,940
## Variables: 10
## $ carat   <dbl> 0.23, 0.21, 0.23, 0.29, 0.31, 0.24, 0.24, 0.26, 0.22, ...
## $ cut     <ord> Ideal, Premium, Good, Premium, Good, Very Good, Very G...
## $ color   <ord> E, E, E, I, J, J, I, H, E, H, J, J, F, J, E, E, I, J, ...
## $ clarity <ord> SI2, SI1, VS1, VS2, SI2, VVS2, VVS1, SI1, VS2, VS1, SI...
## $ depth   <dbl> 61.5, 59.8, 56.9, 62.4, 63.3, 62.8, 62.3, 61.9, 65.1, ...
## $ table   <dbl> 55, 61, 65, 58, 58, 57, 57, 55, 61, 61, 55, 56, 61, 54...
## $ price   <int> 326, 326, 327, 334, 335, 336, 336, 337, 337, 338, 339,...
## $ x       <dbl> 3.95, 3.89, 4.05, 4.20, 4.34, 3.94, 3.95, 4.07, 3.87, ...
## $ y       <dbl> 3.98, 3.84, 4.07, 4.23, 4.35, 3.96, 3.98, 4.11, 3.78, ...
## $ z       <dbl> 2.43, 2.31, 2.31, 2.63, 2.75, 2.48, 2.47, 2.53, 2.49, ...
```

```
# View(diamonds) # Para ver el data frame en formato de Excel.
```

```
diamonds_columnas_selectas <- select(diamonds, carat, cut)
diamonds_columnas_selectas
```

```
## # A tibble: 53,940 x 2
##   carat cut
##   <dbl> <ord>
## 1 0.23 Ideal
```

```
## 2 0.21 Premium
## 3 0.23 Good
## 4 0.290 Premium
## 5 0.31 Good
## 6 0.24 Very Good
## 7 0.24 Very Good
## 8 0.26 Very Good
## 9 0.22 Fair
## 10 0.23 Very Good
## # ... with 53,930 more rows
```

```
# Otras maneras de seleccionar columnas:
select(diamonds, starts_with("c"))
```

```
## # A tibble: 53,940 x 4
##   carat cut      color clarity
##   <dbl> <ord>    <ord> <ord>
## 1 0.23 Ideal E      SI2
## 2 0.21 Premium E      SI1
## 3 0.23 Good E      VS1
## 4 0.290 Premium I      VS2
## 5 0.31 Good J      SI2
## 6 0.24 Very Good J      VVS2
## 7 0.24 Very Good I      VVS1
## 8 0.26 Very Good H      SI1
## 9 0.22 Fair E      VS2
## 10 0.23 Very Good H      VS1
## # ... with 53,930 more rows
```

```
select(diamonds, contains("able"))
```

```
## # A tibble: 53,940 x 1
##   table
##   <dbl>
## 1 55
## 2 61
## 3 65
## 4 58
## 5 58
## 6 57
## 7 57
## 8 55
## 9 61
## 10 61
## # ... with 53,930 more rows
```

```
select(diamonds, -carat, -cut, -color)
```

```
## # A tibble: 53,940 x 7
##   clarity depth table price      x      y      z
##   <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1 SI2      61.5   55   326  3.95  3.98  2.43
## 2 SI1      59.8   61   326  3.89  3.84  2.31
## 3 VS1      56.9   65   327  4.05  4.07  2.31
## 4 VS2      62.4   58   334  4.2   4.23  2.63
## 5 SI2      63.3   58   335  4.34  4.35  2.75
```

```
## 6 VVS2      62.8    57   336  3.94  3.96  2.48
## 7 VVS1      62.3    57   336  3.95  3.98  2.47
## 8 SI1       61.9    55   337  4.07  4.11  2.53
## 9 VS2       65.1    61   337  3.87  3.78  2.49
## 10 VS1      59.4    61   338  4      4.05  2.39
## # ... with 53,930 more rows
```



Crea un data frame nuevo a partir de diamonds con las columnas carat, x, y, z únicamente.

2.5.2 2. Filtrar registros de un data frame que cumplen cierta condición: filter(df, condiciones)

```
diamonds_registros_selectos <- filter(diamonds, cut == "Ideal", x > 4)
diamonds_registros_selectos
```

```
## # A tibble: 21,449 x 10
##   carat cut    color clarity depth table price     x     y     z
##   <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.31 Ideal J      SI2     62.2    54   344  4.35  4.37  2.71
## 2  0.3   Ideal I      SI2     62     54   348  4.31  4.34  2.68
## 3  0.33 Ideal I      SI2     61.8    55   403  4.49  4.51  2.78
## 4  0.33 Ideal I      SI2     61.2    56   403  4.49  4.5   2.75
## 5  0.33 Ideal J      SI1     61.1    56   403  4.49  4.55  2.76
## 6  0.32 Ideal I      SI1     60.9    55   404  4.45  4.48  2.72
## 7  0.3   Ideal I      SI2     61     59   405  4.3   4.33  2.63
## 8  0.35 Ideal I      VS1     60.9    57   552  4.54  4.59  2.78
## 9  0.3   Ideal D      SI1     62.5    57   552  4.29  4.32  2.69
## 10 0.3   Ideal D      SI1     62.1    56   552  4.3   4.33  2.68
## # ... with 21,439 more rows
```

```
# Otras maneras de filtrar registros
filter(diamonds, cut == "Ideal" & x > 4)
```

```
## # A tibble: 21,449 x 10
##   carat cut    color clarity depth table price     x     y     z
##   <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.31 Ideal J      SI2     62.2    54   344  4.35  4.37  2.71
## 2  0.3   Ideal I      SI2     62     54   348  4.31  4.34  2.68
## 3  0.33 Ideal I      SI2     61.8    55   403  4.49  4.51  2.78
## 4  0.33 Ideal I      SI2     61.2    56   403  4.49  4.5   2.75
## 5  0.33 Ideal J      SI1     61.1    56   403  4.49  4.55  2.76
## 6  0.32 Ideal I      SI1     60.9    55   404  4.45  4.48  2.72
## 7  0.3   Ideal I      SI2     61     59   405  4.3   4.33  2.63
## 8  0.35 Ideal I      VS1     60.9    57   552  4.54  4.59  2.78
## 9  0.3   Ideal D      SI1     62.5    57   552  4.29  4.32  2.69
## 10 0.3   Ideal D      SI1     62.1    56   552  4.3   4.33  2.68
## # ... with 21,439 more rows
```

```
filter(diamonds, cut == "Ideal" | x > 4)
```

```
## # A tibble: 53,546 x 10
##   carat cut    color clarity depth table price     x     y     z
##   <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal  E      SI2     61.5    55   326  3.95  3.98  2.43
```

```
## 2 0.23 Good E VS1 56.9 65 327 4.05 4.07 2.31
## 3 0.290 Premium I VS2 62.4 58 334 4.2 4.23 2.63
## 4 0.31 Good J SI2 63.3 58 335 4.34 4.35 2.75
## 5 0.26 Very Good H SI1 61.9 55 337 4.07 4.11 2.53
## 6 0.3 Good J SI1 64 55 339 4.25 4.28 2.73
## 7 0.23 Ideal J VS1 62.8 56 340 3.93 3.9 2.46
## 8 0.31 Ideal J SI2 62.2 54 344 4.35 4.37 2.71
## 9 0.32 Premium E I1 60.9 58 345 4.38 4.42 2.68
## 10 0.3 Ideal I SI2 62 54 348 4.31 4.34 2.68
## # ... with 53,536 more rows
```



Crea un data frame nuevo a partir de diamonds que contenga los registros que cumplen la condición: ‘alguna de x, y, z es mayor a 3.5’

2.5.3 3. Ordenar registros de un data frame por los valores en una o más variables: `arrange(df, variables_de_ordenamiento)`

```
diamonds_ordenado <- arrange(diamonds, carat, depth)
diamonds_ordenado
```

```
## # A tibble: 53,940 x 10
##   carat cut      color clarity depth table price      x      y      z
##   <dbl> <ord>    <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.2 Premium E      VS2      59      60    367  3.81  3.78  2.24
## 2  0.2 Premium E      VS2     59.7     62    367  3.84  3.8   2.28
## 3  0.2 Ideal E      VS2     59.7     55    367  3.86  3.84  2.3
## 4  0.2 Premium E      VS2     59.8     62    367  3.79  3.77  2.26
## 5  0.2 Premium E      SI2     60.2     62    345  3.79  3.75  2.27
## 6  0.2 Premium E      VS2     61.1     59    367  3.81  3.78  2.32
## 7  0.2 Ideal D      VS2     61.5     57    367  3.81  3.77  2.33
## 8  0.2 Premium D      VS2     61.7     60    367  3.77  3.72  2.31
## 9  0.2 Ideal E      VS2     62.2     57    367  3.76  3.73  2.33
## 10 0.2 Premium D      VS2     62.3     60    367  3.73  3.68  2.31
## # ... with 53,930 more rows
```

```
# desc(variable): ordena en sentido decreciente (Z-A), (mayor a menor, etc)
arrange(diamonds, carat, desc(depth))
```

```
## # A tibble: 53,940 x 10
##   carat cut      color clarity depth table price      x      y      z
##   <dbl> <ord>    <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.2 Very Good E      VS2     63.4     59    367  3.74  3.71  2.36
## 2  0.2 Premium F      VS2     62.6     59    367  3.73  3.71  2.33
## 3  0.2 Premium D      VS2     62.3     60    367  3.73  3.68  2.31
## 4  0.2 Ideal E      VS2     62.2     57    367  3.76  3.73  2.33
## 5  0.2 Premium D      VS2     61.7     60    367  3.77  3.72  2.31
## 6  0.2 Ideal D      VS2     61.5     57    367  3.81  3.77  2.33
## 7  0.2 Premium E      VS2     61.1     59    367  3.81  3.78  2.32
## 8  0.2 Premium E      SI2     60.2     62    345  3.79  3.75  2.27
## 9  0.2 Premium E      VS2     59.8     62    367  3.79  3.77  2.26
## 10 0.2 Premium E      VS2     59.7     62    367  3.84  3.8   2.28
## # ... with 53,930 more rows
```



```
arrange(diamonds, desc(carat), depth)
```

```
## # A tibble: 53,940 x 10
##   carat cut      color clarity depth table price      x      y      z
##   <dbl> <ord>    <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  5.01 Fair      J      I1      65.5   59 18018 10.7 10.5  6.98
## 2  4.5  Fair      J      I1      65.8   58 18531 10.2 10.2  6.72
## 3  4.13 Fair      H      I1      64.8   61 17329 10   9.85  6.43
## 4  4.01 Premium I      I1      61     61 15223 10.1 10.1  6.17
## 5  4.01 Premium J      I1      62.5   62 15223 10.0 9.94  6.24
## 6  4     Very Good I      I1      63.3   58 15984 10.0 9.94  6.31
## 7  3.67 Premium I      I1      62.4   56 16193 9.86 9.81  6.13
## 8  3.65 Fair      H      I1      67.1   53 11668 9.53 9.48  6.38
## 9  3.51 Premium J      VS2     62.5   59 18701 9.66 9.63  6.03
## 10 3.5  Ideal      H      I1      62.8   57 12587 9.65 9.59  6.03
## # ... with 53,930 more rows
```



Crea un data frame nuevo a partir de diamonds que contenga los registros organizados alfabéticamente por “color”, y por “carat” de manera descendente.

2.5.4 4. Crear nuevas variables: mutate(df, formulas)

```
diamonds_nueva_variable <- mutate(diamonds, dollars_per_carat = price / carat)
```



Crea un data frame nuevo a partir de diamonds que contenga la variable “dollars_per_carat” anterior, y la variable “product” calculada como $x \times y \times z$.

2.5.5 5. Agrupar por ciertas variables: group_by(data_frame, variables) y crear resúmenes por grupo: summarise(data_frame, formulas)

- Para crear resúmenes de un data frame se utiliza la función **summarise**.
- Si se agrupan los datos antes (utilizando la función **group_by**), se pueden crear resúmenes por nivel de las variables de agregación (definidas en el **group_by**). Esto es análogo a las tablas dinámicas en Excel.

```
# Calcular el promedio de depth y la mediana de price y asignarlos a las
# variables "promedio_depth" y "mediana_price"
diamonds_resumen <- summarise(diamonds, promedio_depth = mean(depth), mediana_price = median(price))
diamonds_resumen
```

```
## # A tibble: 1 x 2
##   promedio_depth mediana_price
##           <dbl>         <dbl>
## 1          61.7          2401
```

```
# Calcular el promedio de depth y la mediana de price y asignarlos a las variables
# "promedio_depth" y "mediana_price", por nivel de "cut"
# Primero agrupo por la variable de interés
diamonds_agrupado <- group_by(diamonds, cut)
diamonds_agrupado
```

```
## # A tibble: 53,940 x 10
## # Groups:   cut [5]
```

```
##      carat cut      color clarity depth table price      x      y      z
##      <dbl> <ord>      <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1 0.23 Ideal      E      SI2      61.5   55   326   3.95   3.98   2.43
##  2 0.21 Premium    E      SI1      59.8   61   326   3.89   3.84   2.31
##  3 0.23 Good      E      VS1      56.9   65   327   4.05   4.07   2.31
##  4 0.290 Premium  I      VS2      62.4   58   334   4.2    4.23   2.63
##  5 0.31 Good      J      SI2      63.3   58   335   4.34   4.35   2.75
##  6 0.24 Very Good J      VVS2      62.8   57   336   3.94   3.96   2.48
##  7 0.24 Very Good I      VVS1      62.3   57   336   3.95   3.98   2.47
##  8 0.26 Very Good H      SI1      61.9   55   337   4.07   4.11   2.53
##  9 0.22 Fair      E      VS2      65.1   61   337   3.87   3.78   2.49
## 10 0.23 Very Good H      VS1      59.4   61   338   4      4.05   2.39
## # ... with 53,930 more rows
```

Y calculo resúmenes para cada nivel de dicha variable

```
diamonds_resumen_por_grupo <- summarise(diamonds_agrupado, promedio_depth = mean(depth), mediana_price = median(price))
diamonds_resumen_por_grupo
```

```
## # A tibble: 5 x 3
##   cut      promedio_depth mediana_price
##   <ord>          <dbl>          <dbl>
## 1 Fair            64.0            3282
## 2 Good            62.4            3050.
## 3 Very Good       61.8            2648
## 4 Premium         61.3            3185
## 5 Ideal           61.7            1810
```

Algunos resúmenes útiles con *summarise* son:

- El mínimo de un campo x: **min(x)**
- La mediana de un campo x: **median(x)**
- El máximo de un campo x: **max(x)**
- El número de registros: **n()**
- La suma de un campo x: **sum(x)**
- La desviación estándar de un campo x: **sd(x)**.



Tarea: Crea un data frame nuevo a partir de diamonds que contenga el número de registros para cada combinación de valores en las variables “cut” y “color”.



Tarea: Explica las diferencias que notas entre los data frames obtenidos de:

```
diamonds_agrupado <- group_by(diamonds, cut)
resultado_1 <- summarise(diamonds_agrupado, promedio_depth = mean(depth))
resultado_2 <- mutate(diamonds_agrupado, promedio_depth = mean(depth))
```



Tarea: Partiendo del data frame diamonds, crea un data frame nuevo con las siguientes características:

1. Contenga la variable `dollars_per_carat = price / carat`
2. Contenga sólo aquellos registros que cumplen `dollars_per_carat < 4000`
3. Esté ordenado en orden descendente por la variable `dollars_per_carat`

Chapter 3

Transformación-II

3.1 ¿Qué aprendimos la clase pasada?

- Utilizamos R para saber si comparaciones entre dos cantidades son ciertas o no ($5 > 6$ #FALSE, $6 == 6$ #TRUE)
- Aprendimos el uso de los operadores y (&), o (|) y no (!): $(5 > 6) | (3 < 4)$ #TRUE
- Aprendimos las funciones básicas para manipular **tablas de datos** (paquete **dplyr** contenido en el tidyverse):

Funcionalidad	Función
Seleccionar campos	<code>select(diamonds, carat, cut)</code>
Seleccionar registros de acuerdo a un criterio	<code>filter(diamonds, cut == "Ideal" x > 4)</code>
Ordenar registros de acuerdo a uno o más campos	<code>arrange(diamonds, carat, depth)</code>
Crear nuevas variables	<code>mutate(diamonds, dollars_per_carat = price / carat)</code>
Preparar un data frame para calcular resúmenes por grupo	<code>diamonds_agrupado <- group_by(diamonds, cut)</code>
Calcular resúmenes por grupo	<code>summarise(diamonds_agrupado, promedio_depth = mean(d</code>

En esta clase aprenderemos a:

- Utilizar el operador **pipeline** para simplificar la aplicación de funciones de transformación de datos, una tras otra (recordar la tarea, ejercicio 3).
- Miscelánea de funcionalidades avanzadas de transformación de datos:
 - **joins** (uniones de dos o más tablas).
 - El paquete **tidyr** para transformar la **estructura** de los datos en una tabla.
- Un poco acerca de datos faltantes.
- Leer datos en R.

3.2 El operador pipeline %>%

El ejercicio 3 de la tarea nos introduce a lo tedioso que es aplicar varias funciones para transformar datos una tras otra sin ayuda. Aquí es cuando el operador pipeline entra en acción:

Nos permite encadenar operaciones de manera sencilla, comenzando por el data frame original (diamonds), luego aplicar una transformación, al resultado aplicar otra y así sucesivamente.

Retomemos el ejemplo de la tarea 3. En lugar de:

```
diamonds_dollars_per_carat <- mutate(diamonds, dollars_per_carat = price / carat)
diamonds_dollars_per_carat
```

```
## # A tibble: 53,940 x 11
##   carat cut      color clarity depth table price      x      y      z
##   <dbl> <ord>    <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1 0.23 Ideal     E      SI2     61.5   55   326   3.95   3.98   2.43
## 2 0.21 Premium  E      SI1     59.8   61   326   3.89   3.84   2.31
## 3 0.23 Good     E      VS1     56.9   65   327   4.05   4.07   2.31
## 4 0.290 Premium I      VS2     62.4   58   334   4.2    4.23   2.63
## 5 0.31 Good     J      SI2     63.3   58   335   4.34   4.35   2.75
## 6 0.24 Very Good J      VVS2     62.8   57   336   3.94   3.96   2.48
## 7 0.24 Very Good I      VVS1     62.3   57   336   3.95   3.98   2.47
## 8 0.26 Very Good H      SI1     61.9   55   337   4.07   4.11   2.53
## 9 0.22 Fair     E      VS2     65.1   61   337   3.87   3.78   2.49
## 10 0.23 Very Good H      VS1     59.4   61   338   4      4.05   2.39
## # ... with 53,930 more rows, and 1 more variable: dollars_per_carat <dbl>

diamonds_dollars_per_carat_filtrado <- filter(diamonds_dollars_per_carat, dollars_per_carat < 4000)
diamonds_dollars_per_carat_filtrado

## # A tibble: 32,083 x 11
##   carat cut      color clarity depth table price      x      y      z
##   <dbl> <ord>    <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1 0.23 Ideal     E      SI2     61.5   55   326   3.95   3.98   2.43
## 2 0.21 Premium  E      SI1     59.8   61   326   3.89   3.84   2.31
## 3 0.23 Good     E      VS1     56.9   65   327   4.05   4.07   2.31
## 4 0.290 Premium I      VS2     62.4   58   334   4.2    4.23   2.63
## 5 0.31 Good     J      SI2     63.3   58   335   4.34   4.35   2.75
## 6 0.24 Very Good J      VVS2     62.8   57   336   3.94   3.96   2.48
## 7 0.24 Very Good I      VVS1     62.3   57   336   3.95   3.98   2.47
## 8 0.26 Very Good H      SI1     61.9   55   337   4.07   4.11   2.53
## 9 0.22 Fair     E      VS2     65.1   61   337   3.87   3.78   2.49
## 10 0.23 Very Good H      VS1     59.4   61   338   4      4.05   2.39
## # ... with 32,073 more rows, and 1 more variable: dollars_per_carat <dbl>

diamonds_dollars_per_carat_filtrado_ordenado <- arrange(diamonds_dollars_per_carat_filtrado, desc(dollars_per_carat))
diamonds_dollars_per_carat_filtrado_ordenado

## # A tibble: 32,083 x 11
##   carat cut      color clarity depth table price      x      y      z
##   <dbl> <ord>    <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1 1.05 Very Good J      SI1     62.5   58  4199   6.47   6.52   4.06
## 2 0.92 Very Good E      SI2     63.2   54  3679   6.29   6.25   3.96
## 3 0.92 Premium  E      SI2     61.8   59  3679   6.19   6.11   3.8
## 4 0.91 Premium  E      SI2     61.1   59  3639   6.24   6.2    3.8
## 5 0.91 Premium  E      SI2     62.8   61  3639   6.09   6.07   3.82
## 6 0.9 Premium   E      SI2     62.6   60  3599   6.18   6.09   3.84
## 7 0.9 Premium   E      SI2     62.2   60  3599   6.19   6.15   3.84
## 8 0.9 Ideal     E      SI2     62     55  3599   6.23   6.15   3.84
## 9 1.51 Premium  H      I1     61.9   58  6038   7.39   7.34   4.56
## 10 0.74 Fair     G      VVS2     65.2   58  2959   5.7    5.6    3.69
## # ... with 32,073 more rows, and 1 more variable: dollars_per_carat <dbl>
```

El ejemplo de la tarea 3 queda:

```
diamonds %>% # Comenzando con el df diamonds:
  mutate(dollars_per_carat = price / carat) %>% # Calcúlame la variable dollars per carat ... LUEGO
  filter(dollars_per_carat < 4000) %>% # Seleccióname los registros en que la variable dollars_per_carat
```

```

arrange(desc(dollars_per_carat)) # Ordénalo en orden descendente por la variable dollars per carat

## # A tibble: 32,083 x 11
##   carat cut      color clarity depth table price      x      y      z
##   <dbl> <ord>    <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  1.05 Very Good J      SI1      62.5   58  4199  6.47  6.52  4.06
## 2  0.92 Very Good E      SI2      63.2   54  3679  6.29  6.25  3.96
## 3  0.92 Premium  E      SI2      61.8   59  3679  6.19  6.11  3.8
## 4  0.91 Premium  E      SI2      61.1   59  3639  6.24  6.2   3.8
## 5  0.91 Premium  E      SI2      62.8   61  3639  6.09  6.07  3.82
## 6  0.9  Premium  E      SI2      62.6   60  3599  6.18  6.09  3.84
## 7  0.9  Premium  E      SI2      62.2   60  3599  6.19  6.15  3.84
## 8  0.9  Ideal    E      SI2      62     55  3599  6.23  6.15  3.84
## 9  1.51 Premium  H      I1      61.9   58  6038  7.39  7.34  4.56
## 10 0.74 Fair     G      VVS2     65.2   58  2959  5.7   5.6   3.69
## # ... with 32,073 more rows, and 1 more variable: dollars_per_carat <dbl>

```

Podemos también asignar el resultado de TODAS las transformaciones anteriores a una variable

```

diamonds_transformado_1 <- diamonds %>%
  mutate(dollars_per_carat = price / carat) %>%
  filter(dollars_per_carat < 4000) %>%
  arrange(desc(dollars_per_carat))
diamonds_transformado_1

```

```

## # A tibble: 32,083 x 11
##   carat cut      color clarity depth table price      x      y      z
##   <dbl> <ord>    <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  1.05 Very Good J      SI1      62.5   58  4199  6.47  6.52  4.06
## 2  0.92 Very Good E      SI2      63.2   54  3679  6.29  6.25  3.96
## 3  0.92 Premium  E      SI2      61.8   59  3679  6.19  6.11  3.8
## 4  0.91 Premium  E      SI2      61.1   59  3639  6.24  6.2   3.8
## 5  0.91 Premium  E      SI2      62.8   61  3639  6.09  6.07  3.82
## 6  0.9  Premium  E      SI2      62.6   60  3599  6.18  6.09  3.84
## 7  0.9  Premium  E      SI2      62.2   60  3599  6.19  6.15  3.84
## 8  0.9  Ideal    E      SI2      62     55  3599  6.23  6.15  3.84
## 9  1.51 Premium  H      I1      61.9   58  6038  7.39  7.34  4.56
## 10 0.74 Fair     G      VVS2     65.2   58  2959  5.7   5.6   3.69
## # ... with 32,073 more rows, and 1 more variable: dollars_per_carat <dbl>

```

Otro ejemplo:

- Por combinación de cut y color,
- Calcular el mínimo de x, y también el máximo de y.
- Al resultado ordenarlo por color de manera descendente.

```

diamonds_transformado_2 <- diamonds %>%
  # Primero agrupo por combinación de cut y color, ya que lo necesito para calcular
  # los resúmenes por grupo
  group_by(cut, color) %>%
  # Luego calculo los resúmenes por grupo
  summarise(minimo_x = min(x), maximo_y = max(y)) %>%
  # Finalmente ordeno por color
  arrange(desc(color))
diamonds_transformado_2

```

```
## # A tibble: 35 x 4
```

Table 3.1: Tipos de caracter

id	tipo
1	letra
2	número
3	caracter especial

```
## # Groups:   cut [5]
##   cut      color minimo_x maximo_y
##   <ord>    <ord>    <dbl>   <dbl>
## 1 Fair      J        4.24    10.5
## 2 Good      J        4.22    9.19
## 3 Very Good J        3.94    8.93
## 4 Premium   J        4.22    9.94
## 5 Ideal     J        3.93    9.2
## 6 Fair      I        4.62    9.02
## 7 Good      I        4.19    9.31
## 8 Very Good I        3.95    9.94
## 9 Premium   I        3.97    10.1
## 10 Ideal    I        3.94    9.42
## # ... with 25 more rows
```

3.3 Miscelánea de funcionalidades avanzadas de transformación de datos

Con **dplyr**:

1. Realizar **joins** entre dos tablas.

Con **tidyr**:

2. **Spread**: transformar registros en campos
3. **Gather**: transformar campos en registros
4. **Separate**: separar variables

3.3.1 1. Joins: `inner_join(df1, df2, columnas_a_seleccionar)`

Es común encontrarse tablas que hacer referencia la una a la otra, por ejemplo:

Para asociar a cada caracter su tipo, podemos utilizar una funcionalidad llamada **join**, que básicamente asocia registros de dos tablas usando campos en común.

```
# Definiendo las tablas anteriores (normalmente estas tablas se leerán de archivos
# CSV o bases de datos como se verá en esta clase).
```

```
tipos_caracter <- data_frame(
  id = c(1, 2, 3),
  tipo = c("letra", "número", "caracter especial")
)
tipos_caracter
```

```
## # A tibble: 3 x 2
```

Table 3.2: Caracteres

id	caracter	tipo_caracter_id
1	a	1
2	2	2
3	3	2
4	1	2
5	z	1
6	5	2
7	m	1
8	7	2
9	s	1
10	x	1

```
##      id tipo
## <dbl> <chr>
## 1     1 letra
## 2     2 número
## 3     3 caracter especial
```

```
caracteres <- data_frame(
  id = 1:10,
  caracter = c("a", "2", "3", "1", "z", "5", "m", "7", "s", "x"),
  tipo_caracter_id = c(1, 2, 2, 2, 1, 2, 1, 2, 1, 1)
)
caracteres
```

```
## # A tibble: 10 x 3
##       id caracter tipo_caracter_id
##   <int> <chr>         <dbl>
## 1     1 a             1
## 2     2 2             2
## 3     3 3             2
## 4     4 1             2
## 5     5 z             1
## 6     6 5             2
## 7     7 m             1
## 8     8 7             2
## 9     9 s             1
## 10    10 x            1
```

```
# Haciendo el join de las tablas anteriores
```

```
inner_join(caracteres, tipos_caracter, by = c("tipo_caracter_id" = "id"))
```

```
## # A tibble: 10 x 4
##       id caracter tipo_caracter_id tipo
##   <int> <chr>         <dbl> <chr>
## 1     1 a             1 letra
## 2     2 2             2 número
## 3     3 3             2 número
## 4     4 1             2 número
## 5     5 z             1 letra
## 6     6 5             2 número
## 7     7 m             1 letra
```

```
## 8      8 7          2 número
## 9      9 s          1 letra
## 10     10 x         1 letra
```

```
# Notemos que el orden importa para renombrar y ordenar las columnas
inner_join(tipos_caracter, caracteres, by = c("id" = "tipo_caracter_id"))
```

```
## # A tibble: 10 x 4
##       id tipo   id.y caracter
##   <dbl> <chr> <int> <chr>
## 1     1  1 letra     1 a
## 2     2  1 letra     5 z
## 3     3  1 letra     7 m
## 4     4  1 letra     9 s
## 5     5  1 letra    10 x
## 6     6  2 número     2 2
## 7     7  2 número     3 3
## 8     8  2 número     4 1
## 9     9  2 número     6 5
## 10    10  2 número     8 7
```

```
# Existen muchos tipos de joins, y también joins por más de un campo. Para ver
# estas opciones consultar la ayuda de R: ?inner_join.
```



Expresa el join anterior usando el pipeline.



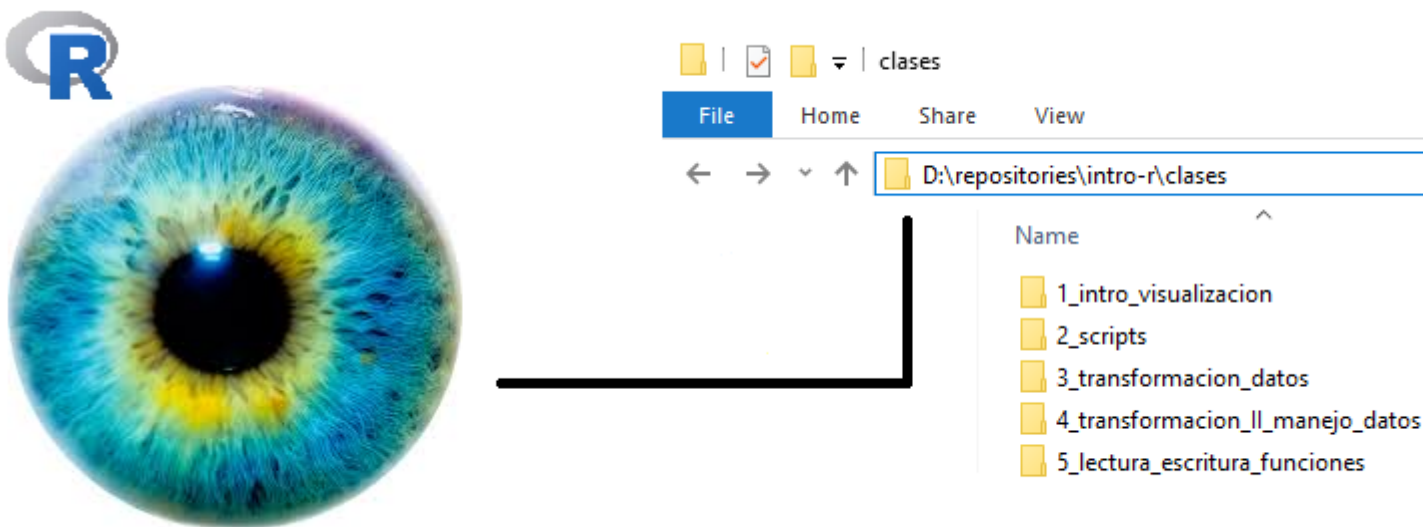
Evalúa las siguientes expresiones y explica con tus palabras el resultado.

```
left_join(tipos_caracter, caracteres, by = c("id" = "tipo_caracter_id"))
semi_join(tipos_caracter, caracteres, by = c("id" = "tipo_caracter_id"))
anti_join(tipos_caracter, caracteres, by = c("id" = "tipo_caracter_id"))
```

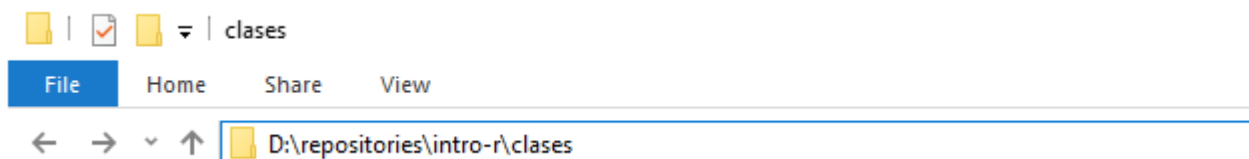
3.3.2 Un intermedio. Lectura de datos en R

La mecánica de lectura en R es sencilla y, sin importar el tipo de archivo que se quiera cargar a nuestro espacio de trabajo, siempre tiene la misma forma: se utiliza una función preparada para cargar un cierto tipo de archivo y luego se le debe indicar a R dónde está el archivo (de este tipo) que se desea cargar.

R tiene una única manera de saber dónde buscar un archivo. Debe recibir una dirección que le indique dónde buscar físicamente el archivo de interés.



R puede “ver” lo que sea que le muestres, esto es, puedes decirle exáctamente dónde debe buscar un archivo, por ejemplo indicando con una cadena de texto una ruta completa en nuestro sistema de archivos (disco duro): “D:\repositories\intro-r\”. Estas rutas se pueden escribir manualmente, o se pueden copiar del explorador de archivos de nuestro sistema operativo y luego pegarla en R.



Es muy importante notar que en Windows, la convención es usar diagonales invertidas “\” para separar los niveles de nuestra ruta. R no va a entender que algo es una ruta si está construída con estos símbolos. Si se copia y pega una ruta desde nuestro explorador de archivos en Windows, debemos cambiar las diagonales invertidas por diagonales: “D:/repositories/intro-r/”.

Otra posibilidad, también ya mencionada, es indicarle a R que “vea” una carpeta de trabajo. En este momento R está “viendo” la siguiente carpeta:

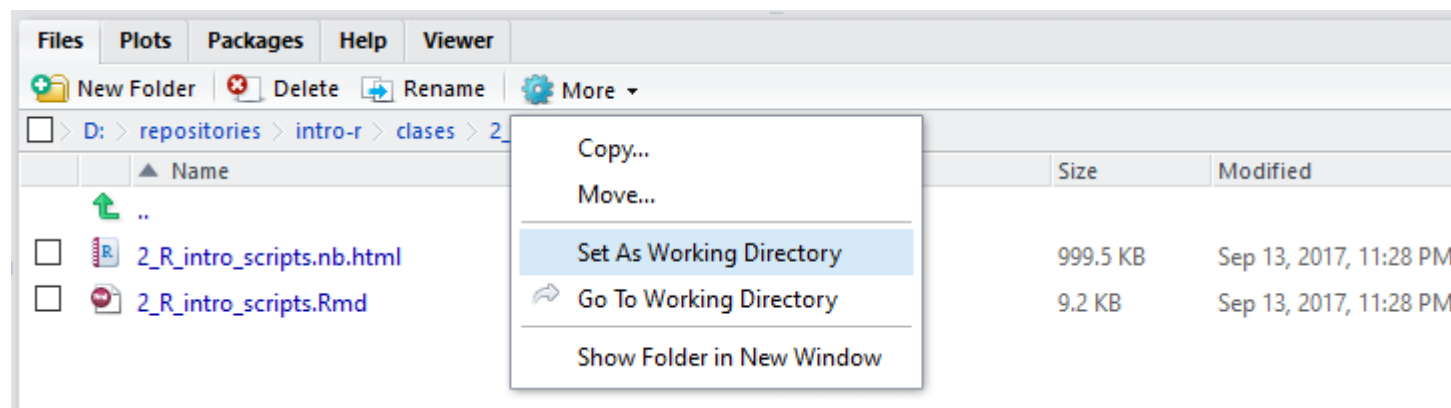
```
getwd()
```

```
## [1] "/Users/agutierrez/Documents/R/r"
```

Esto indica que R no necesita una ruta completa para leer cualquier cosa incluída en la carpeta anterior. Únicamente el archivo. Para cambiar la carpeta de trabajo se utiliza la función `setwd()` que como argumento esencial recibe una ruta.

Se recomienda trabajar un proyecto dado en una carpeta que a su vez contenga una carpeta que amacene los datos que se usen para ese proyecto en particular. Por ejemplo una carpeta llamada “datos”, así las rutas a los archivos siempre pueden ser carpetas relativas. No importa si la carpeta del proyecto se copia a otro equipo de cómputo, bastará con hacer `setwd()` a la carpeta del proyecto para que todo el código funcione.

Otra cosa que vale la pena mencionar es que RStudio incluye un explorador de archivos, ahí se puede navegar en las carpetas de nuestra computadora y con las opciones disponibles en el ícono de engrane se puede también asignar la carpeta de trabajo.



Es común que las tablas de datos se distribuyan como archivos de texto. Existen numerosas variaciones de estos archivos de texto y csvs.

El tidyverse incluye al paquete `readr` que tiene como objetivo convertir este tipo de archivos en data frames, aunque archivos de este tipo delimitados por comas es lo más común, nos podemos encontrar con archivos delimitados por otros símbolos, de aquí que existan las siguientes funciones:

- `read_csv2()` que lee archivos separados por punto y coma “;”
- `read_tsv()` lee archivos separados por Tabs
- `read_delim()` lee archivos separados por cualquier símbolo (tú lo determinas con un argumento)
- `read_fwf()` lee archivos de anchos fijos, se pueden especificar los anchos con `fwf_widths()` o su posición en el espacio (fila de datos) con `fwf_positions()`.
- `read_table()` lee un caso muy particular y popular de lo anterior que son archivos con datos separados por un único espacio.

Como ya se mencionó, todas estas funciones tienen una sintaxis similar. Lo más importante es alimentarles bien la ruta del archivo que se desea leer.

Aunque este tipo de archivos son extremadamente populares en el mundo de los archivos de datos, existen numerosas otras presentaciones. Por ejemplo, es muy común el uso de Microsoft Excel para análisis cuantitativo. R está bien preparado para leer y escribir archivos excel.

Bajaremos datos de Incidencia de Víctimas de homicidio, secuestro y extorsión de datos.gob.mx:

<https://datos.gob.mx/busca/dataset/victimas-de-homicidio-secuestro-y-extorsion-excel>

Para leer estos datos se utilizaremos el paquete `readxl` (no se les olvide instalarlo)

```
# cargar el paquete
library("readxl")

# ¿dónde está el archivo? recordar usar rutas relativas:
ruta_relativa <- "../datos/Estatal_Victimas_2015_2018_feb.xlsx"

# leer este archivo
datos <- read_excel(ruta_relativa,sheet=1)
```

3.3.3 2. Usar `spread` para transformar registros de un data frame en campos.

Al transformar registros en campos, se quitan renglones y se agregan columnas, lo que se llama datos **anchos**.

Es importante notar que `gather` y `spread` son funciones inversas.

Primero veamos el estado original de la tabla de datos, tiene múltiples columnas que corresponden a muchos cortes en la naturaleza de los crímenes cometidos (Estado donde se cometió, mes cuando se cometió, sexo de la víctima, tipo de crimen, etc).

```
# renombrar la primera variable porque incluye un molesto símbolo especial: ~
datos <- rename(datos, Anio = Año)

# generar totales por mes
datos <- datos %>%
  mutate(total = select(., Enero:Diciembre) %>% rowSums())
```

Ahora sí vamos a pasar la tabla de datos a un formato largo con base en los tipos de delitos

```
# trabajar sólo Homicidios y Feminicidios, pasar a los datos a formato largo por año

# Usando la función spread para transformar registros en campos:
# key: variable cuyos valores definirán los nombres de nuestros campos. Para
# revertir el data frame usaremos "enfermedad"

tipos_delito = datos %>%
  spread(key = `Tipo de delito`, value = total)
```

Ya que está en formato largo, se puede generar una útil tabla de conteo de delitos cruzado por Entidad, Año y Sexo.

```
# cantidad de subtipos de delito por entidad, año y sexo
cantidad_tipos <- tipos_delito %>%
  group_by(Entidad, Anio, Sexo) %>%
  summarise_at (vars=vars(Aborto:Secuestro), .funs=sum, na.rm=TRUE)
```

3.3.4 3. Usar gather para transformar campos de un data frame en registros

Al transformar campos en registros, se quitan columnas y se agregan renglones al data frame. Esto se llama datos **largos**.

Podemos usar esta operación para eliminar las columnas-delitos de la tabla de datos anterior y agregarla a unas columnas de conteos por delito (estructura: nombre delito, conteo)

```
# Usando la función gather para transformar campos en registros:
# key: nombre de la columna con los nombres de los campos (ahora registros)
# value: nombre de la columna con los valores de los campos (numero_pacientes)
# lo que sigue son las columnas que definen los campos que se transformarán en renglones
cantidad_tipos_largos <- gather(cantidad_tipos,
  key = "tipo_delito",
  value = "num_delitos_cometidos",
  Aborto:Secuestro)
```



A partir de los datos originales, crear otra tabla que sea interesante utilizando las funciones gather o spread y las otras funciones de manipulación vistas hasta ahora. Investigar el uso de la función de escritura de texto delimitado write_csv(), también del paquete reader, y utilizarla para guardar la tabla de datos anterior a su carpeta “datos”.

3.3.5 4. Separar una columna en dos o más: `separate(df, col = columna, into = c(nueva_variable_1, nueva_variable_2, etc))`

`Separate` es una función útil para separar una columna de un data frame en varias columnas, cuyos nombres se especifican. La separación default se realiza por caracteres especiales (`.`, `_`, espacios, etc). Por ejemplo:

```
instructores_curso_r <- data_frame(
  id = c(1,2,3),
  nombre = c(
    "Amaury Gutiérrez",
    "Teresa Ortiz",
    "Julian Equihua"
  )
)
```

```
## # A tibble: 3 x 2
##       id nombre
##   <dbl> <chr>
## 1     1 Amaury Gutiérrez
## 2     2 Teresa Ortiz
## 3     3 Julian Equihua
```

```
separate(instructores_curso_r, nombre, into = c("nombre", "apellido_1"))
```

```
## # A tibble: 3 x 3
##       id nombre apellido_1
##   <dbl> <chr>   <chr>
## 1     1 Amaury Gutiérrez
## 2     2 Teresa Ortiz
## 3     3 Julian Equihua
```



Evalúa las siguientes expresiones, y explica con tus palabras lo que sucede

```
instructores_curso_r_1 <- data_frame(
  id = c(1,2,3),
  nombre = c(
    "Fernando Pardo Urrutia",
    "Teresa Ortiz",
    "Julian Equihua"
  )
)
```

```
separate(instructores_curso_r_1, nombre, into = c("nombre", "apellido_1"))
separate(instructores_curso_r_1, nombre, into = c("nombre", "apellido_1", "apellido_2"))
```



Da una explicación intuitiva de lo que es el **NA**

3.4 Datos faltantes

Un **NA** es un dato faltante, es decir, un vacío en una tabla. Como en R los data frames contienen un elemento en cada campo, estos vacíos se traducen como datos faltantes.

Como son vacíos de información, los datos faltantes se pueden pensar como “no sé”

```

NA # Dato Faltante

## [1] NA
NA + 3 # No se + 3 = No sé

## [1] NA
NA * 3 # No se * 3 = No sé

## [1] NA
is.na(NA) # Un operador binario para preguntar si un dato es faltante (NA)

## [1] TRUE
is.na(5.3)

## [1] FALSE
is.na(FALSE)

## [1] FALSE
FALSE | NA # No se cuánto da FALSE ó NA

## [1] NA
TRUE | NA # Pero TRUE o NA sí, porque sabemos que verdadero ó lo que sea ya es verdadero: como (5 > 3)

## [1] TRUE
NA > 5 # ¿Es NO SE > 5? NO Sé

## [1] NA
NA == NA # ¿Es NO SE igual a NO SÉ? NO SÉ

## [1] NA
sum(c(4, 5, 6, NA)) # No se cuanto da la suma de algo 4, 5, 6, y no sé.

## [1] NA
sum(c(4, 5, 6, NA), na.rm = TRUE) # Pero puedo decirle a R que remueva los NA's

## [1] 15
mean(c(4, 5, 6, NA), na.rm = TRUE)

## [1] 5
# Dado un data frame con datos faltantes
registro <- data_frame(
  id = c(1, 2, 3),
  persona = c("Fernando", NA, "Julián"),
  numero_socio = c(13, 12, NA)
)
registro

## # A tibble: 3 x 3
##       id persona  numero_socio
##   <dbl> <chr>      <dbl>
## 1     1 Fernando        13

```

```
## 2      2 <NA>          12
## 3      3 Julián       NA
```

Puedo seleccionar renglones con NA:

```
registro %>%
  filter(is.na(persona))
```

```
## # A tibble: 1 x 3
##   id persona numero_socio
##   <dbl> <chr>         <dbl>
## 1     2 <NA>           12
```

0 renglones sin NA:

```
registro %>%
  filter(!is.na(persona))
```

```
## # A tibble: 2 x 3
##   id persona numero_socio
##   <dbl> <chr>         <dbl>
## 1     1 Fernando       13
## 2     3 Julián        NA
```

Puedo usar las reglas anteriores para calcular nuevas columnas:

```
registro %>%
  mutate(numero_socio_nuevo = numero_socio + 1)
```

```
## # A tibble: 3 x 4
##   id persona numero_socio numero_socio_nuevo
##   <dbl> <chr>         <dbl>         <dbl>
## 1     1 Fernando       13             14
## 2     2 <NA>           12             13
## 3     3 Julián        NA              NA
```

Puedo ordenar y los NA's quedan al final

```
registro %>%
  arrange(numero_socio)
```

```
## # A tibble: 3 x 3
##   id persona numero_socio
##   <dbl> <chr>         <dbl>
## 1     2 <NA>           12
## 2     1 Fernando       13
## 3     3 Julián        NA
```

Puedo ordenar y los NA's quedan al final

```
registro %>%
  arrange(desc(numero_socio))
```

```
## # A tibble: 3 x 3
##   id persona numero_socio
##   <dbl> <chr>         <dbl>
## 1     1 Fernando       13
## 2     2 <NA>           12
## 3     3 Julián        NA
```

Tengo que tener cuidado con calcular resúmenes de data frames que contienen NA's

```
registro %>%
  summarise(promedio = mean(numero_socio))
```

```
## # A tibble: 1 x 1
##   promedio
##   <dbl>
## 1      NA

# Pero puedo arreglarlo fácilmente
registro %>%
  summarise(promedio = mean(numero_socio, na.rm = TRUE))

## # A tibble: 1 x 1
##   promedio
##   <dbl>
## 1    12.5
```



Entregar un script donde se lleve a cabo un proceso de manipulación y transformación de datos sobre la tabla de delitos. Adicionalmente generar un gráfico interesante utilizando lo visto en las clases de visualización de datos. Comentar cada paso y explicar qué comunica la gráfica generada.

Chapter 4

Funciones

4.1 ¿Qué aprendimos la clase pasada?

- Se aprendió a usar el operador pipe `%>%`
- Se mostró el concepto de hacer joins entre tablas
- Lectura de datos en R (excel)
- Breve introducción a los datos faltantes

4.2 Funciones en R

Las funciones permiten la automatización de tareas. La escritura de funciones tiene tres grandes ventajas:

- Puedes nombrar una función de una manera tal que nunca olvides qué hace, esto además hace a tu código más legible
- Cuando cambien los requerimientos de los trabajos día a día, deberás de modificar menos código.
- Se reduce la probabilidad de cometer errores llevando a cabo procesos manuales como copiar/pegar datos e instrucciones.

Aprender a escribir buenas funciones es un proceso que nunca termina. Siempre se encontrarán maneras novedosas de mejorar estilos a la hora de escribir funciones. Esta sección no tiene como objetivo profundizar en la escritura de funciones, si no empezar a escribir funciones útiles lo más pronto posible.

4.2.1 ¿Cuándo es conveniente escribir funciones?

Como regla general, si notamos que para llevar a cabo una cierta tarea se está copiando y pegando un bloque de código más de dos veces, es hora de escribir una función para ello. Motivaremos la discusión con el cálculo de áreas y volúmenes. Si escribimos una función en R que permita llevar a cabo estas operaciones, las podremos usar en cualquier momento.

4.2.2 Escribiendo funciones en R

Pasos para crear una función:

- Elegir un nombre apropiado
- Pensar en qué hace y cuántos argumentos necesita para ejecutarse

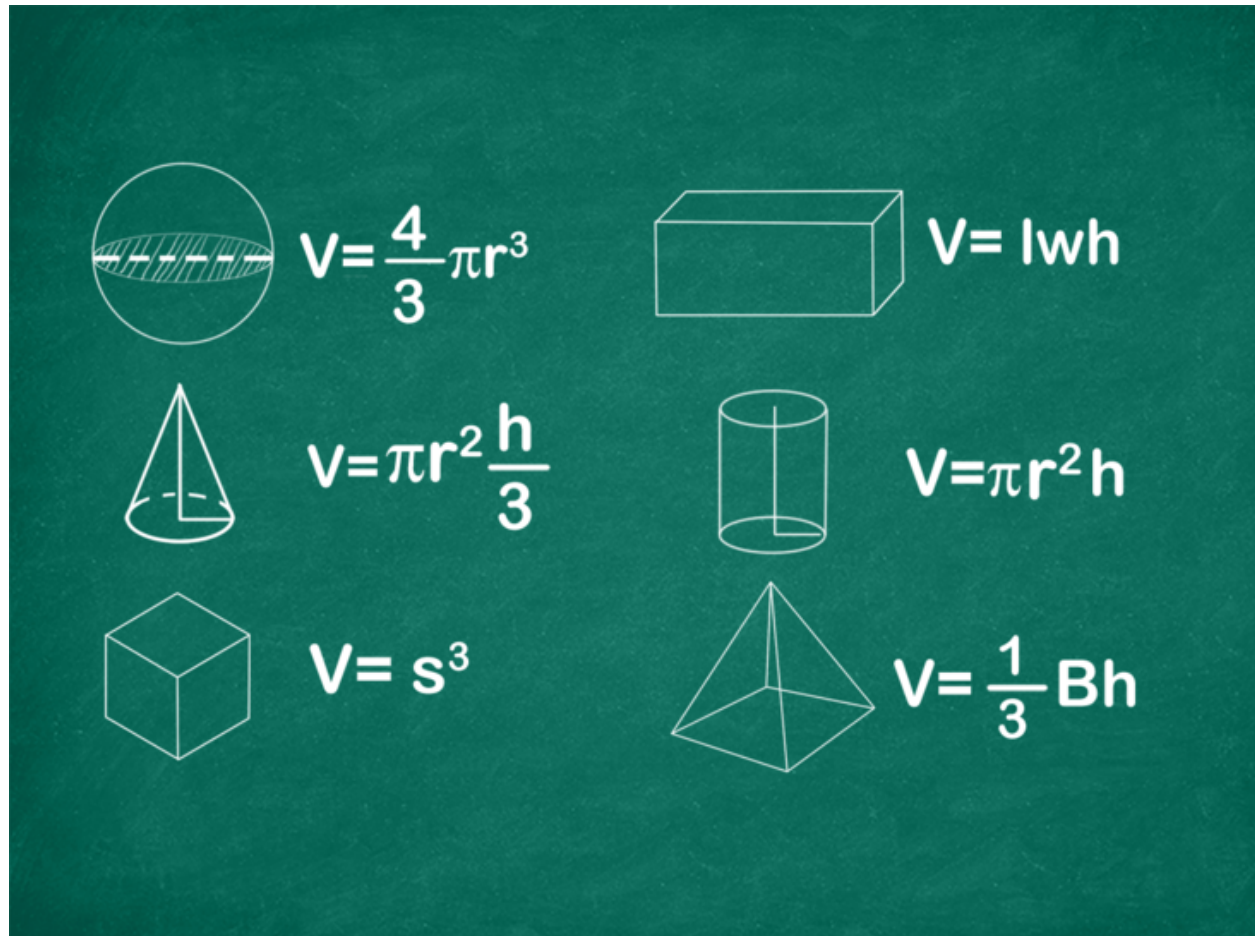


Figure 4.1:

- Traducir lo que queremos que haga a código de R dentro de un bloque asociado a una función y a los argumentos anteriores

Pensando en estos pasos decidamos que nuestra función se llame `area_circulo()` (sin acento, es en general, mala práctica utilizar caracteres especiales en código). Luego sabemos que la función recibe un único parámetro, el radio del círculo al que se le va a calcular el área. Finalmente se debe plasmar la fórmula para el cálculo del área de un círculo en R.

Como un primer ejemplo escribamos una función que permita calcular el área de un círculo.

```
# nombre ----- argumentos
area_circulo <- function (radio) {
  if (radio < 0) # expresión condicional se ejecuta en caso de que el radio sea negativo.
  {
    stop("Para calcular un area es necesario un valor positivo o igual a 0.") # detiene la ejecución
  } else # sigue la expresión condicional, este bloque se ejecutará cuando el radio sea positivo.
  {
    area_calculada <- pi * radio ** 2 # operamos sobre el radio del círculo
    return(area_calculada) # regresamos el resultado (se asigna después de aplicar esta función)
  }
}
```

Para probar la función la llamamos como cualquier función de una librería, notemos que es posible asignar el resultado a una variable:

```
area_radio_cinco <- area_circulo(5)
area_radio_cinco
```

```
## [1] 78.53982
```



Ejercicio 1: Escribir una función análoga a la anterior pero para calcular el área de un rectángulo.



Ejercicio 2: ¿Como podría usar la función del ejercicio 1 para calcular el area de un cuadrado?

4.2.3 Expresiones condicionales

En ocasiones es deseable ejecutar bloques de código únicamente cuando cierta condición se cumple. Existe una sentencia que nos permite hacer esto y tiene la forma siguiente:

```
if (cond) {
  # código ejecutado en caso de que la condición sea verdadera (TRUE).
} else {
  # código ejecutado en caso de que la condición sea falsa (FALSE).
}
```

Es posible omitir la segunda condición:

```
if (cond) {
  # si la condición es verdadera se ejecutará este bloque, en caso contrario no hará nada.
}
```

También es posible encadenar sentencias condicionales:

```
if (cond_1) {
  # código ejecutado en caso de que la condición 1 sea verdadera.
} else if (cond_2) {
  # código ejecutado en caso de que la condición 1 sea falsa pero la condición 2 es verdadera.
}
```

```

} else {
  # cuando ambas condiciones son falsas se ejecuta este bloque.
}

```

Regresando a nuestra función `area_circulo`, notemos que nuestra formulación tiene un problema. ¿Qué pasa cuando la función recibe números negativos? ¿Tiene sentido hablar de áreas negativas? Usaremos las sentencias condicionales para arreglar este detalle:

```

# nombre ----- argumentos
area_circulo <- function (radio) {
  if (radio < 0) # expresión condicional se ejecuta en caso de que el radio sea negativo.
  {
    stop("Para calcular un area es necesario un valor positivo o igual a 0.") # detiene la ejecución
  } else # sigue la expresión condicional, este bloque se ejecutará cuando el radio sea positivo.
  {
    area_calculada <- pi * radio^2 # operamos sobre el radio del círculo
    return(area_calculada) # regresamos el resultado (se asigna después de aplicar esta función)
  }
}

```

Probemos nuestra función mejorada con ambos tipos de valores. ¿Qué pasa cuando la usamos con un número negativo?

```

# probemos nuestra función en
area_resultante <- area_circulo(-10)

```

```

# probemos nuestra función en
area_resultante <- area_circulo(7)

```



Ejercicio 3: Reescribir la función que calcula el área de un rectángulo para solventar el error permitir parámetros negativos.



Ejercicio 4: Escribir una función llamada `salario` que reciba un número que representara el salario de un empleado. Si el salario es mayor a 25,000 deberá regresar el texto “alto”, si el salario es menor a 5000 regresará “bajo”, en otro caso regresará “medio”. Si recibe un salario negativo la función deberá detenerse y lanzar un mensaje de error.

4.2.4 De regreso al tidyverse

Para conectar con lo visto en clases pasadas veremos que es posible encadenar nuestras funciones con el método de los pipes. Para ejemplificar esto regresaremos a nuestro objetivo inicial, el cálculo de volúmenes. Recordemos que los volúmenes de figuras regulares se pueden calcular multiplicando el área de la base del objeto por su altura, nuestra función queda definida de la siguiente forma:

```

volumen_regular <- function (area_base, altura) {
  if (area_base < 0 || altura < 0) { # nos aseguramos que ninguno de los argumentos sea negativo.
    stop("Tanto el area como la altura deben ser mayores o iguales a 0.")
  } else {
    volumen <- area_base * altura
    return(volumen)
  }
}

```

Podemos calcular el volumen de un cilindro de área 3 y altura 7 de la siguiente forma:



Figure 4.2:

```
# definimos las dimensiones
radio <- 3
altura <- 7
# calculamos área y volumen
area_base <- area_circulo(radio)
volumen_cilindro <- volumen_regular(area_base, altura)
#imprimimos el resultado
volumen_cilindro
```

```
## [1] 197.9203
```

En una sola sentencia:

```
volumen_cilindro <- volumen_regular(area_circulo(3), 7)
#imprimimos el resultado
volumen_cilindro
```

```
## [1] 197.9203
```

Sin embargo nada impide hacer el uso del flujo que implementan los pipes. De este modo es posible hacer que el código sea más legible al encadenar las operaciones de izquierda a derecha en vez de adentro hacia afuera:

```
library(tidyverse)
volumen_cilindro <- 3 %>%
  area_circulo %>%
  volumen_regular(7)
volumen_cilindro
```

```
## [1] 197.9203
```

Nota: Recordemos que al usar los pipes, el resultado de una función en la cadena será el argumento de la siguiente. En caso de que la siguiente función requiera más de un argumento el resultado de la función se usará como el primer argumento y resto se tienen que especificar explícitamente.



Ejercicio 5: Reusa este procedimiento para calcular el volumen de un cubo de lado 6 y un paralelepípedo de dimensiones 4, 5 y 7.

4.2.5 Ambientes

Otro componente de las funciones es el ambiente. Hasta ahora no ha sido mencionado porque hemos sido muy correctos en la definición de nuestras funciones. Cada variable dentro de la función ha sido pasada como un argumento o bien definida dentro de la función. ¿Qué pasaría si usáramos una variable que no ha sido definida dentro del cuerpo de la función ni ha sido pasada como argumento? Resulta que R es laxo con este tipo de situaciones y lo que hace es buscar si la variable ha sido definida en el contexto en el que se definió la función:

```
y <- 19

suma_incorrecta <- function(x) {
  return(x + y)
}

suma_incorrecta(6)
```

```
## [1] 25
```

Esto no es recomendable ya que el valor de *y* puede perderse cuando se vuelva a reiniciar RStudio. Este tipo de código puede introducir errores difíciles de rastrear.

4.2.6 Estilos de código y buenas prácticas

Existen diversos estilos de programación, especialmente en R. Ninguno de estos estilos es correcto, pero la consistencia permite que leer y mantener el código sea más sencillo. Para nombrar variables y funciones existen varias convenciones:

```
este_se_llama_snake_case
losProgramadoresSuelenUsarCamelCase
a.mi.no.me.gusta.usar.puntos
Some.menJust_WantTO.Watch.theWorld_BURN
```

Como han podido darse cuenta nosotros usamos snake case.

Cuando los statements condicionales son de una sola sentencia es posible no usar las llaves `{ }`. Sin embargo esto reduce la legibilidad del código.

```
# BIEN las llaves permiten identificar que el código será ejecutado cuando la condición se cumple
muestra_mensaje <- TRUE
if (y < 0 && muestra_mensaje) {
  message("y es negativo")
}

# MAL no hay jerarquía visual
muestra_mensaje <- TRUE
if (y < 0 && muestra_mensaje)
  message("y es negativo")
```

Ahora que saben que se puede hacer... ¡No lo hagan!

La elección de nombres es un tema importante a considerar. El código es un vínculo entre las instrucciones que una computadora puede ejecutar y la forma en la que nosotros resolvemos un problema. El lenguaje de programación nos permite expresar la serie de pasos necesarios para llegar a una solución. Por esto el lenguaje debe no solo ejecutarse correctamente sino también legible para el ojo humano. Los nombres en las variables nos pueden ayudar a esto. Revisemos los nombres de nuestras funciones de área:

```
# BIEN la ayuda de r aparece cuando escribo area y me permite recordar que nombre asigné a cada función
area_circulo(4)
area_cuadrado(3)
area_rectangulo(5,6)

# MAL
circulo_area(4) # el elemento en comun de las funciones no está al principio, dificulta la búsqueda
calcula_area_cuadrado(3) # nomrbe demasiado largo, el verbo sobra porque está implícito
mi_funcion(5,6) # todo mal, ¡no tenemos idea que hace la función!
```

Repitiendo lo que se menciona al principio de la sección no hay una forma correcta de escribir código (existen muchas incorrectas), pero lo más importante es **ser consistente**.

4.2.7 Temas avanzados de funciones (OPCIONAL)

Esta sección solo es demostrativa de las cosas que permite el lenguaje R sobre las funciones. Se mencionan para hacer más amplio el repertorio en caso de que alguien se interese en indagar más al respecto.

4.2.7.1 Funciones recursivas

4.3 Las funciones pueden llamarse a si mismas.

¿Para que querríamos algo así? Las funciones recursivas permiten calcular problemas cuya representación se puede formular en terminos de si mismas. El ejemplo por excelencia es la operación factorial:

$$n! = n \times (n - 1) \times (n - 2) \times \dots \times 2 \times 1$$

Notemos que:

$$n! = n \times (n - 1)!$$

Y a su vez es:

$$n! = n \times (n - 1) \times (n - 2)!$$

Y sí sucesivamente hasta que llegamos hasta el 1.

Por lo que podemos definir una función que calcule dicha operación haciendo uso de la misma definición:

```
factorial <- function (n) {
  if(n == 1) {
    return(1) # caso base, no hace falta calcular más
  } else{
    return(n * factorial(n - 1)) # multiplico el número actual y calculo el factorial del número menos 1
  }
}
```



Figure 4.3:

```
}  
  
factorial(5)  
  
## [1] 120
```

4.3.0.1 Funciones de orden alto

4.4 *Las funciones pueden regresar funciones:*

R es un lenguaje en el que las funciones son objetos que se pueden operar como los números, las cadenas de texto, los data frames.

Con lo que hemos aprendido es posible crear una función que permita calcular el cuadrado de un número con ayuda del operador `**`:

```
cuadrado <- function (x) {  
  return(x ** 2)  
}  
  
cuadrado(4)  
  
## [1] 16
```

De la misma forma podemos obtener una función que calcule el cubo de un número:

```
cubo <- function (x) {  
  return(x ** 3)  
}
```




Figure 4.4:

```
cubo(4)
```

```
## [1] 64
```

Pero la estructura es básicamente la misma. ¿Es indispensable repetir el mismo código cuando el patrón es tan claro? Resulta que R nos permite hacer plantillas para este tipo de casos:

```
fabrica_potencia <- function (n) {  
  function(x) {  
    return(x ** n)  
  }  
}
```

Probemos el resultado:

```
cuadrado <- fabrica_potencia(2)  
cuadrado(4)
```

```
## [1] 16
```

```
cubo <- fabrica_potencia(3)  
cubo(4)
```

```
## [1] 64
```

```
potencia_doce <- fabrica_potencia(12)  
potencia_doce(4)
```

```
## [1] 16777216
```


Chapter 5

Vectores

5.1 ¿Qué aprendimos la clase pasada?

- Evitar repeticiones de código por medio de funciones
- Ejecutar bloques de código usando una condición de control
- Como usar funciones creadas por nosotros en conjunto con las herramientas del tidyverse
- Buenas prácticas para la escritura de código y por qué es deseable el código limpio

5.2 Vectores, listas y arreglos

Existen dos tipos de vectores en R:

- Los vectores atómicos, que son homogéneos, es decir contienen el mismo tipo de dato en cada una de sus entradas.
- Las listas, que son heterogéneas, es decir pueden contener distintos tipos de datos en cada entrada incluso otras listas.

Los vectores atómicos pueden ser de 6 distintos tipos: *logical*, *integer*, *double*, *character*, *complex*, y *raw*.

```
vector_logico <- c(TRUE, FALSE, FALSE, TRUE)
vector_logico
```

```
## [1] TRUE FALSE FALSE TRUE
```

```
typeof(vector_logico)
```

```
## [1] "logical"
```

```
vector_entero <- 1:4
vector_entero
```

```
## [1] 1 2 3 4
```

```
typeof(vector_entero)
```

```
## [1] "integer"
```

```
vector_double <- c(1.2, pi, sqrt(3))
vector_double
```

```
## [1] 1.200000 3.141593 1.732051
```

```
typeof(vector_double)

## [1] "double"
vector_char <- letters[1:4]
vector_char

## [1] "a" "b" "c" "d"
typeof(vector_char)

## [1] "character"
vector_complex <- c(1 + 1i, 5i, 5)
vector_complex

## [1] 1+1i 0+5i 5+0i
typeof(vector_complex)

## [1] "complex"
vector_raw <- c(charToRaw("the quick brown fox jumps over the lazy dog"))
vector_raw

## [1] 74 68 65 20 71 75 69 63 6b 20 62 72 6f 77 6e 20 66 6f 78 20 6a 75 6d
## [24] 70 73 20 6f 76 65 72 20 74 68 65 20 6c 61 7a 79 20 64 6f 67
typeof(vector_raw)

## [1] "raw"
```

Adicionalmente, los vectores integer y double se consideran en la categoría *numeric*.

```
is.numeric(vector_entero)

## [1] TRUE
is.numeric(vector_double)
```

```
## [1] TRUE
```

En R los números por default son de tipo *double*:

```
typeof(6)

## [1] "double"
```

si queremos un entero anexamos una “L” al final:

```
typeof(1)

## [1] "double"
```

Tenemos que tener en cuenta que los *doubles* son aproximaciones entonces pueden pasar cosas como:

```
2 == sqrt(2) ** 2

## [1] FALSE
```

Cuando queremos comprobar igualdades con doubles es mejor usar la función `near` del paquete `tidyverse`:

```
library(tidyverse)
near(2, sqrt(2) ** 2)
```

```
## [1] TRUE
```

Existen valores especiales para los datos double:

```
c(-1, 0, 1) / 0
```

```
## [1] -Inf NaN Inf
```

En lugar de usar “==” para comprobar igualdades, existen funciones especiales:

```
is.infinite(-1/0)
```

```
## [1] TRUE
```

```
is.finite(1.5)
```

```
## [1] TRUE
```

```
is.na(1/0)
```

```
## [1] FALSE
```

```
is.nan(1/0)
```

```
## [1] FALSE
```

```
is.infinite(-1/0)
```

```
## [1] TRUE
```

```
is.finite(1.5)
```

```
## [1] TRUE
```

```
is.na(1/0)
```

```
## [1] FALSE
```

```
is.nan(1/0)
```

```
## [1] FALSE
```

5.2.0.1 Coerción

Cuando usamos un tipo de valor en un contexto que no es el natural, R intentará convertir esos valores en el tipo adecuado para el contexto:

```
sum(c(TRUE, FALSE, FALSE, TRUE, TRUE))
```

```
## [1] 3
```

```
x <- c()
```

```
if(length(x)){ # El valor 0 se interpreta como FALSE, cualquier otro valor como TRUE
  print(x)
} else {
  print("El vector está vacío.")
}
```

```
## [1] "El vector está vacío."
```

Las listas sirven para generar objetos más complejos:

```
x1 <- list(c(1, 2), c(3, 4))
x1

## [[1]]
## [1] 1 2
##
## [[2]]
## [1] 3 4

x2 <- list(list(1, 2), list(3, 4))
x2

## [[1]]
## [[1]][[1]]
## [1] 1
##
## [[1]][[2]]
## [1] 2
##
##
## [[2]]
## [[2]][[1]]
## [1] 3
##
## [[2]][[2]]
## [1] 4
```

Los arreglos son arreglos numéricos de varias dimensiones, por ejemplo matrices:

```
mi_matriz <- array(c(1, 2, 3, 4), dim= c(2, 2))
mi_matriz

##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4

typeof(mi_matriz)

## [1] "double"

mi_matriz <- matrix(c(1, 2, 3, 4), ncol= 2, nrow=2)
mi_matriz

##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4

typeof(mi_matriz)

## [1] "double"
```

5.3 Iteración en R

En la sección anterior se vió cómo evitar duplicar código que se usa de manera recurrente utilizando funciones. Una función abstrae una tarea y permite llevarla a cabo sin escribir código adicional, sólo necesitando ciertos parámetros de entrada.

Otra herramienta útil para evitar escribir más código son los mecanismos de iteración. Estos son útiles cuando se quiere llevar a cabo la misma tarea múltiples veces.

En esta clase aprenderemos sobre dos paradigmas de iteración: la programación imperativa y la programación funcional.

La primera es un paradigma más antiguo y permite introducirse fácilmente al tema pues hace que la iteración sea muy explícita. Como desventaja, las estructuras de este paradigma, llamados bucles (en inglés loops), tienden a ser más extensos en código. Por otro lado, la programación funcional ofrece herramientas para extraer todo el código duplicado para que cada bucle tenga su propia función. Después de un poco de práctica se puede resolver los problemas más comunes en iteración de manera más sencilla, utilizando menos código y por lo mismo cometiendo menos errores.

Antes que nada, cargaremos el conjunto de datos que se usó hace dos clases sobre violencia en México:

```
library("readxl")
ruta_relativa <- "./datos/Estatal_Victimas_2015_2018_feb.xlsx"
datos <- read_excel(ruta_relativa,sheet=1)
```

5.3.1 Iteración imperativa

5.3.1.1 Bucle *for*

Un bucle está compuesto de una secuencia y un cuerpo que se ejecuta con base en esta secuencia.

Una secuencia:

```
1:5
```

```
## [1] 1 2 3 4 5
```

Un bucle for puede ejecutar el mismo código (el contenido en su cuerpo) variando un iterador. Por ejemplo el siguiente código nos muestra uno por uno los números en la secuencia misma y en cada una de esas iteraciones nos muestra la palabra “gatito”.

```
x <- list(7, "a", 6, "gatito")
for (i in 1:length(x)) # para la secuencia del 1 al tamaño de la lista
{
  print(unlist(x[i])) # ejecutar esto
}
```

```
## [1] 7
## [1] "a"
## [1] 6
## [1] "gatito"
```

Podríamos hacer un bucle que obtenga una suma análogo a lo que hace la función `sum()`:

```
suma <- 0

for(i in 1:20) {
  suma <- suma + i
}

suma
```

```
## [1] 210
sum(1:20)
```

```
## [1] 210
```

La siguiente función determina si un número es par.

```
es_par <- function(x) {
  return(x %% 2 == 0)
}
```



Ejercicio: Utilizar la función anterior para escribir los números del 1 al 100 si el número es impar, imprimir “impar” y si el número es par, escribir el número.

```
impar
2
impar
4
impar
6
impar
8
.
.
.
```

```
library("readxl")

setwd("/Users/agutierrez/Documents/R/r/")

ruta_relativa <- "../datos/Estatal_Victimas_2015_2018_feb.xlsx"
datos <- read_excel(ruta_relativa,sheet=1)

datos <- datos %>%
  mutate(Total = select(.,Enero:Diciembre) %>% rowSums())
```

```
total_zacatecas <- 0

for(i in 1:nrow(datos)) {
  fila <- datos[i,]
  if(fila$Entidad == "Aguascalientes" && fila$`Tipo de delito` == "Homicidio"){
    total_zacatecas = total_zacatecas + fila$Total
  }
}

print(total_zacatecas)
```

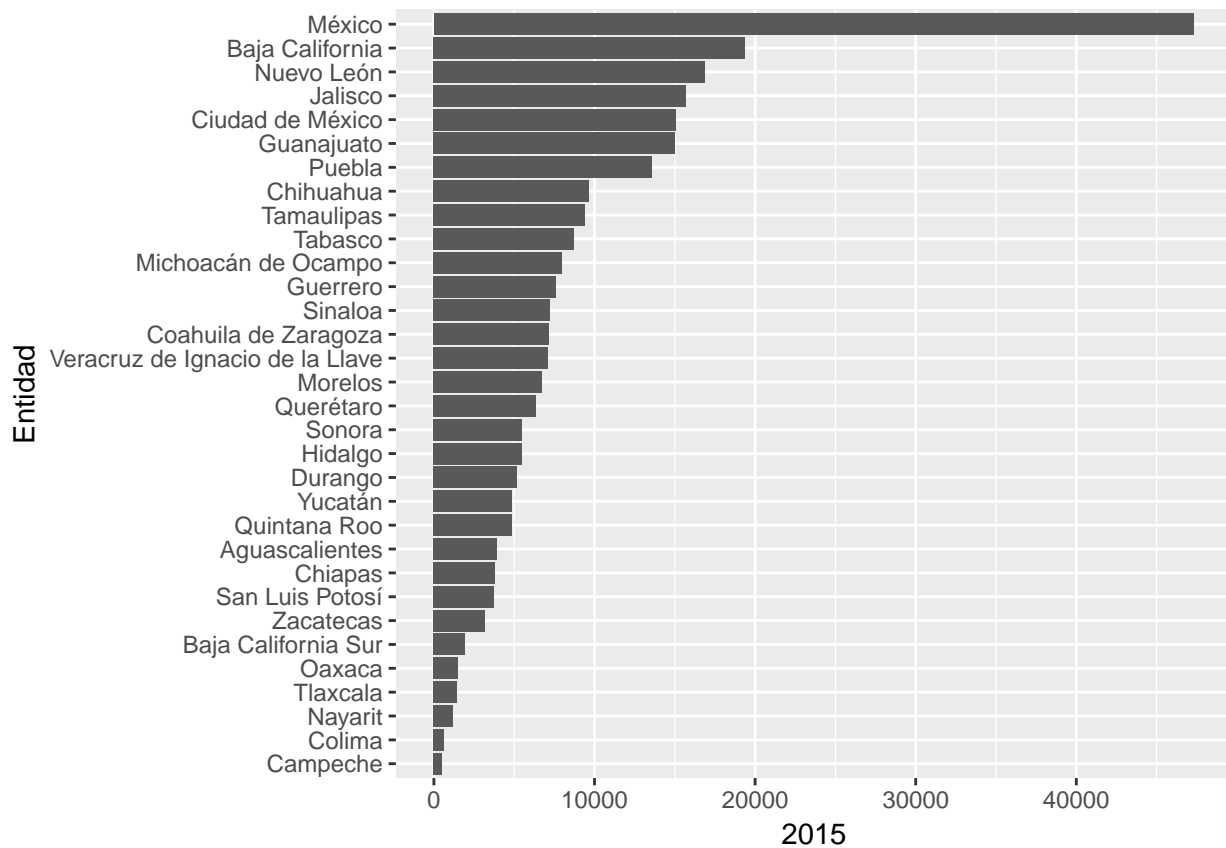
```
## [1] 819
```

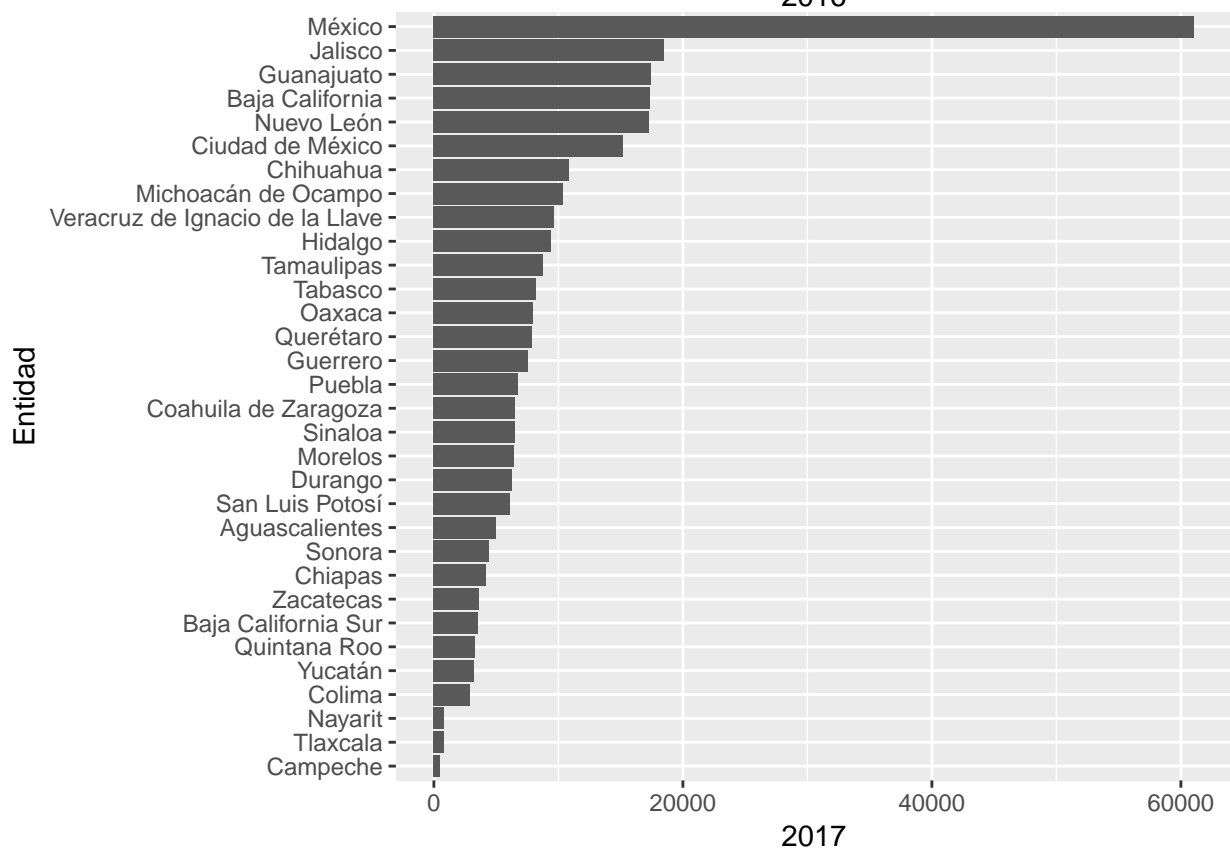
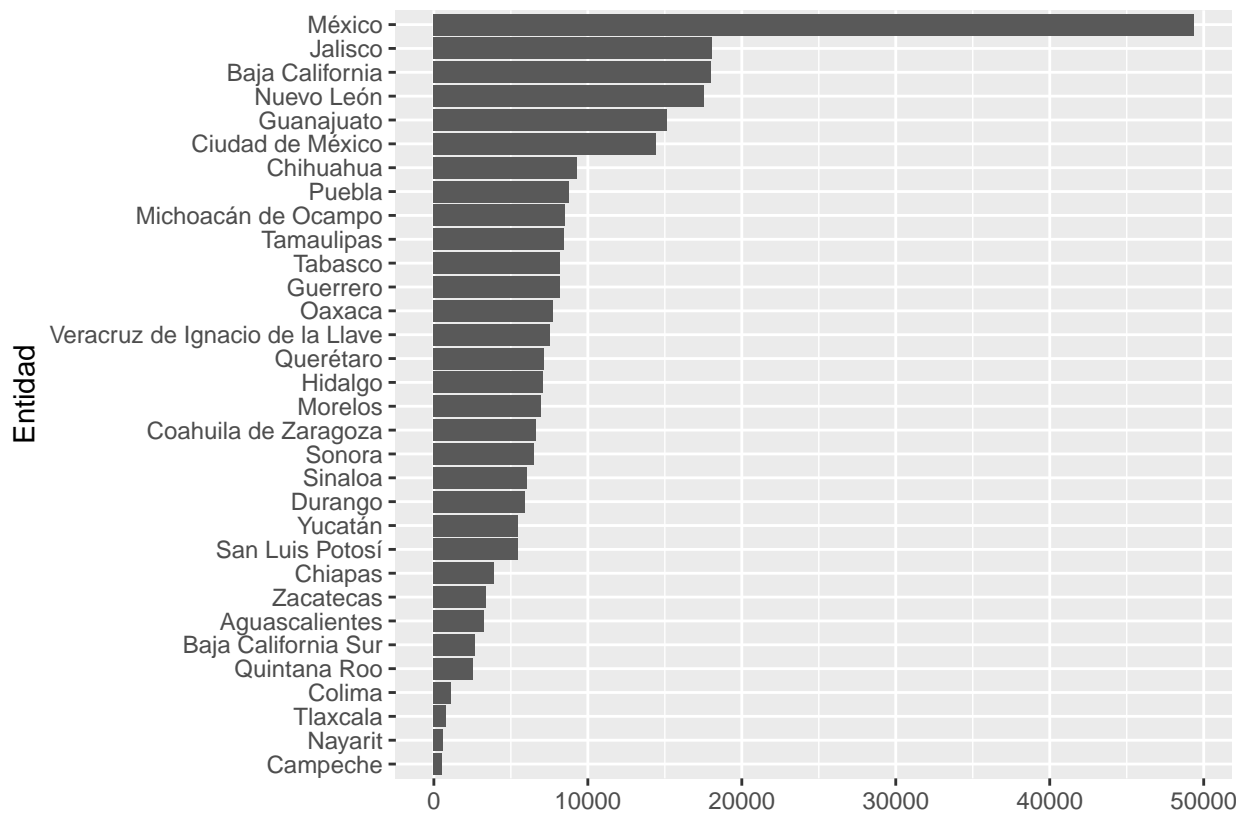
Ahora un ejemplo que une todo lo que hemos visto hasta el momento. Usando la función que extrae los datos de homicidios dolosos para un año y un bucle generaremos una gráfica de barras por año:

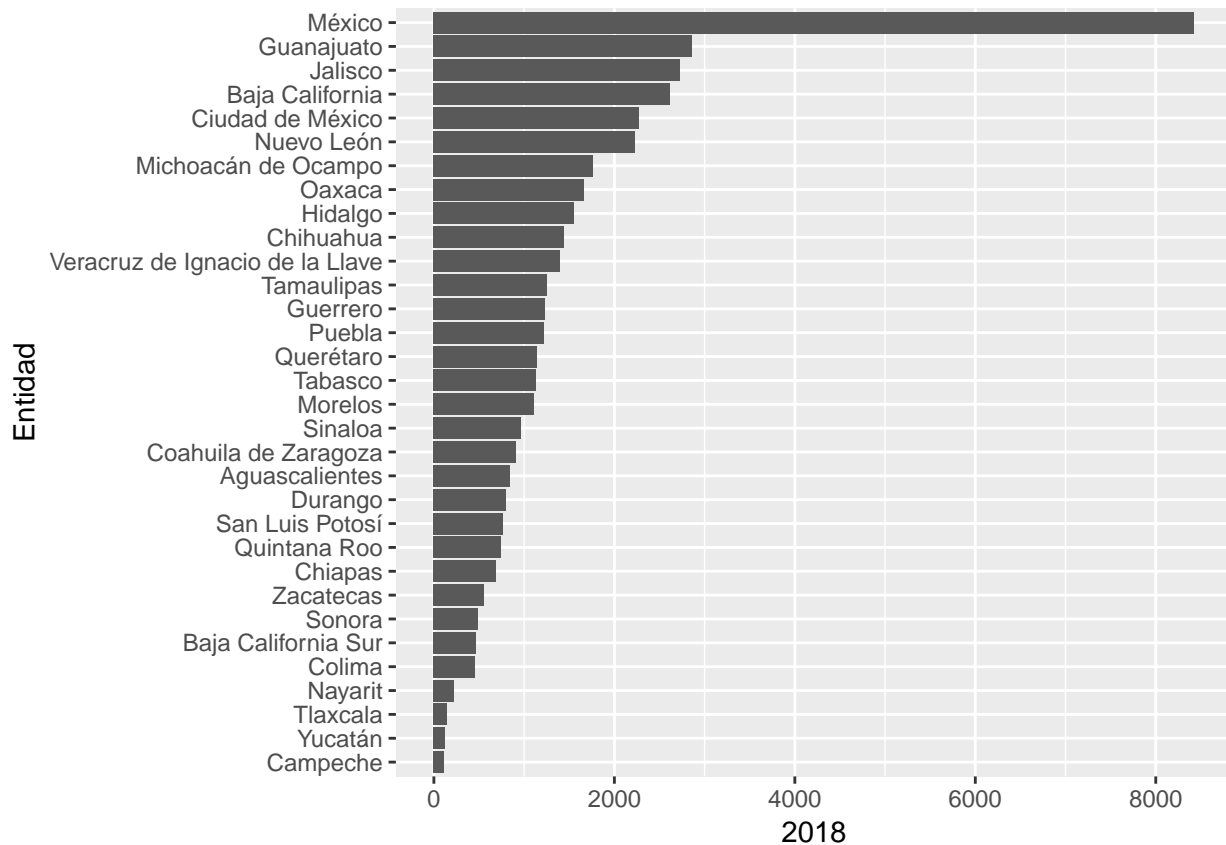
```
extrae_homicidios_fecha <- function(datos, anio)
{
  datos <- datos %>%
    select(Año, Entidad, `Tipo de delito`, Modalidad, Total) %>%
    filter(Año == anio)
  return(datos)
}
```



```
# bucle
for (i in 2015:2018)
{
  homicidios_anio <- extrae_homicidios_fecha(datos, i)
  homicidios_fecha <- toString(i)
  print(ggplot(data = homicidios_anio, aes(x = reorder(Entidad, Total), y = Total)) +
        geom_bar(stat = "identity") +
        labs(y=homicidios_fecha, x="Entidad") +
        coord_flip())
}
```







5.3.1.2 Bucle *while*

While es otro tipo de bucle que se usa cuando no se conoce de antemano el número de iteraciones que se harán. ¿En qué contexto puede ocurrir esto? Usualmente ocurre en problemas de simulación o muestreo, por ejemplo:

Si quisieramos simular un dado:

```

dado <- function(x) {
  sample(1:6, 1, replace=TRUE)
}

```

Ahora pensamos que nos interesa una muestra de tamaño 10 de tiros de dos dados, pero con la particularidad de que la suma de los números sea menor que 8:

```

lanzamientos <- list()

while(length(lanzamientos) < 10) {
  dado_a <- dado()
  dado_b <- dado()
  if(dado_a + dado_b < 8) {
    lanzamientos <- c(lanzamientos, list(c(dado_a, dado_b)))
  }
}

lanzamientos

```

```
## [[1]]
## [1] 3 3
##
## [[2]]
## [1] 5 1
##
## [[3]]
## [1] 4 1
##
## [[4]]
## [1] 1 2
##
## [[5]]
## [1] 1 3
##
## [[6]]
## [1] 5 1
##
## [[7]]
## [1] 3 3
##
## [[8]]
## [1] 1 1
##
## [[9]]
## [1] 1 2
##
## [[10]]
## [1] 1 1
```

5.3.2 Iteración funcional

Ya se introdujo la idea de que se puede utilizar una función dentro de otra función. En esta sección se aprenderá a utilizar el paquete `purrr`, que elimina la necesidad de aprender a generar bucles complejos. La base de R tiene funciones con la misma idea (`apply()`, `lapply()`, `tapply()`, etc) pero `purrr` tiende a ser más consistente y por lo tanto más fácil de aprender a usar.

Si se quiere indagar en las funciones `apply` se puede consultar esta liga:

<https://www.datacamp.com/community/tutorials/r-tutorial-apply-family#gs.bJ=BAKY>

El objetivo de las funciones de `purrr` es ayudar a romper las tareas de manipulación de listas en pedazos independientes:

Primero se debe plantear la interrogante ¿Cómo se puede resolver el problema de interés para un único elemento de la lista? Luego `purrr` ayuda a generalizarlo a cada elemento de la lista.

5.3.2.0.1 La función `map`

La tarea de barrer un vector, hacer algo a cada elemento y luego guardar los resultados es tan común que el paquete `purrr` provee una familia de funciones para llevar esto a cabo.

Existe una función para cada tipo de salida:

- `map()` genera una lista.
- `map_lgl()` genera un vector de valores lógicos.

- `map_int()` genera un vector de valores enteros.
- `map_dbl()` genera un vector de valores dobles (números reales).
- `map_chr()` genera un vector de valores texto.

Cada una de estas funciones recibe como entrada un vector, aplica una función elegida por el usuario a cada pedazo y regresa un vector de la misma longitud que el original (y con los mismos nombres).

Por ejemplo para calcular la media de cada columna de la tabla de datos de coches basta con hacer

```
map_dbl(mtcars, mean)
```

Supongamos que tenemos ahora diversas fuentes de datos que queremos operar en paralelo.

```
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
y <- c(32, 54, 52, 53, 67, 89, 100, 54, 75, 27)

z <- map2_dbl(x, y, function(x,y){ return(x + y) })
z
```

Para evitar la declaración de la función que suma dentro de la llamada podemos hacer:

```
mi_suma <- function(x,y){
  return(x + y)
}

z <- map2_dbl(x, y, mi_suma)
z
```

Pero el paquete `purrr` (parte del `tidyverse`) permite una sintaxis especial. Sustituimos la palabra *function* por una “~” y accedamos a las variables de entrada por medio del placeholder “.”:

```
z <- map2_dbl(x, y, ~ .x + .y)
z
```



Ejercicio: ¿Qué diferencia hay entre usar la función `map2_dbl` y la función `map2`?



Ejercicio: Existe un análogo a la función `map2` para más fuentes de datos llamada `pmap`. Escribe un ejemplo para sumar 3 vectores usando esta función.

5.3.2.0.2 La función `walk`

Es una alternativa a `map` cuando se quiere llamar una función más por sus efectos que por sus resultados. Esto es, sirve para llevar a cabo un proceso a lo largo de, por ejemplo, un vector.

Un ejemplo muy simple

```
x <- list(7, "a", 6, "gatito")

x %>%
  walk(print) # si se fijan hace algo muy parecido a nuestro primer ejemplo de bucle
```

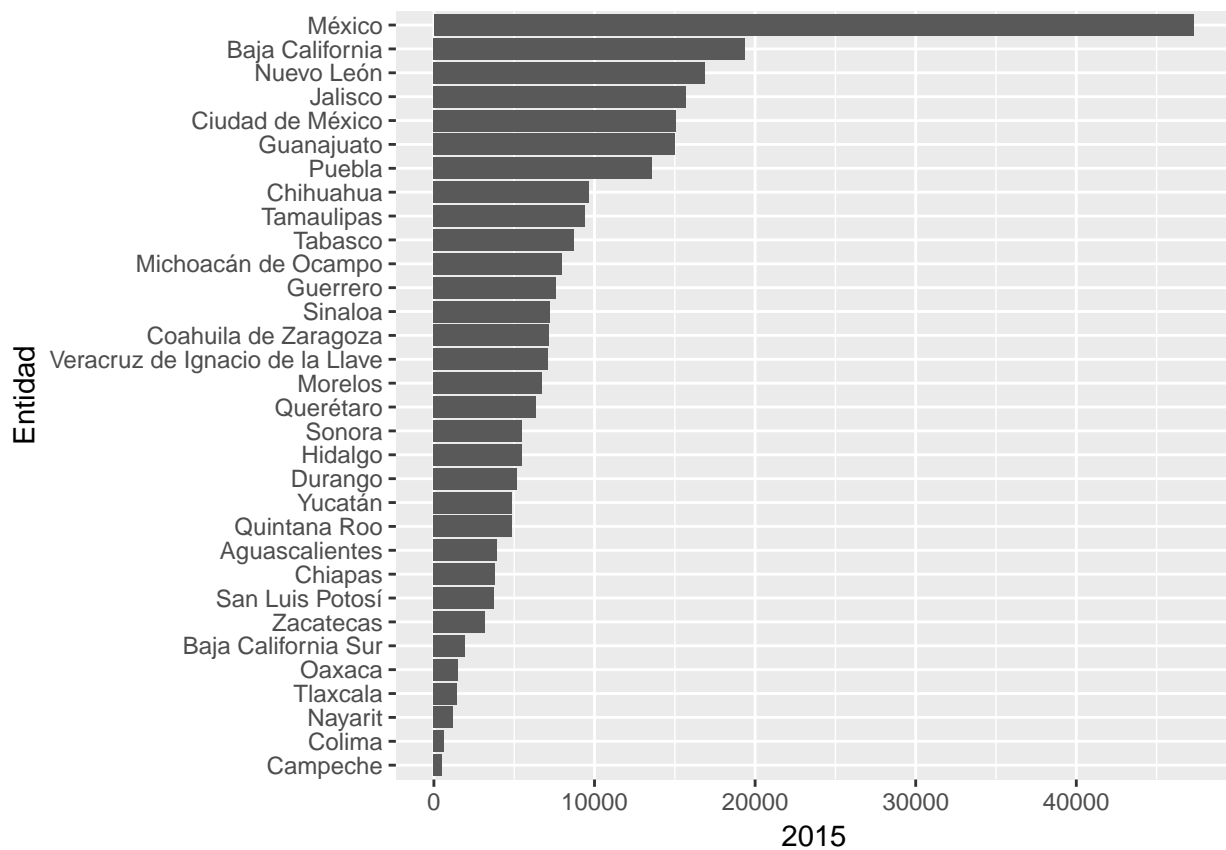
Las funciones `map()` y `walk()` iteran sobre múltiples argumentos en paralelo, `map2()` y `walk2()` se especializan en el caso particular de 2 argumentos y `pmap()` y `pwalk()` en el caso de un número ilimitado de argumentos en una lista. La función `walk`, en general, no es tan útil como las funciones `walk2()` o `pwalk()`.

Para replicar el ejercicio de separar la tabla de datos de homicidios se puede usar `map()` seguido de `pwalk()` para además guardar los plots como pdf.

```
plots <- datos %>%
  split(.$Año) %>% # nota importante: "." al usar map sirve para algo análogo a "i" en los bucles
  map(~ggplot(data = ., aes(x = reorder(Entidad, Total), y = Total)) +
      geom_bar(stat = "identity") +
      labs(y=.$Año, x="Entidad") +
      coord_flip())

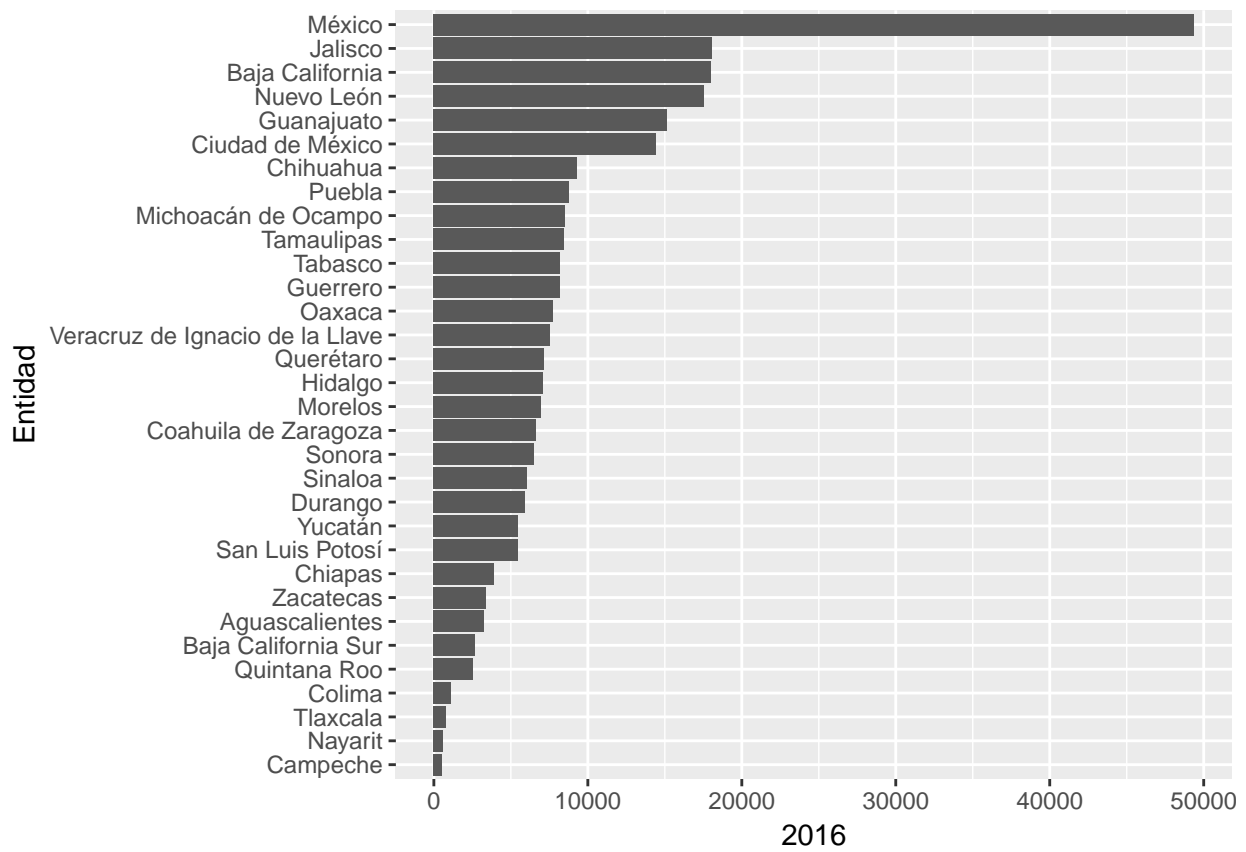
print(plots)
```

```
## $`2015`
```



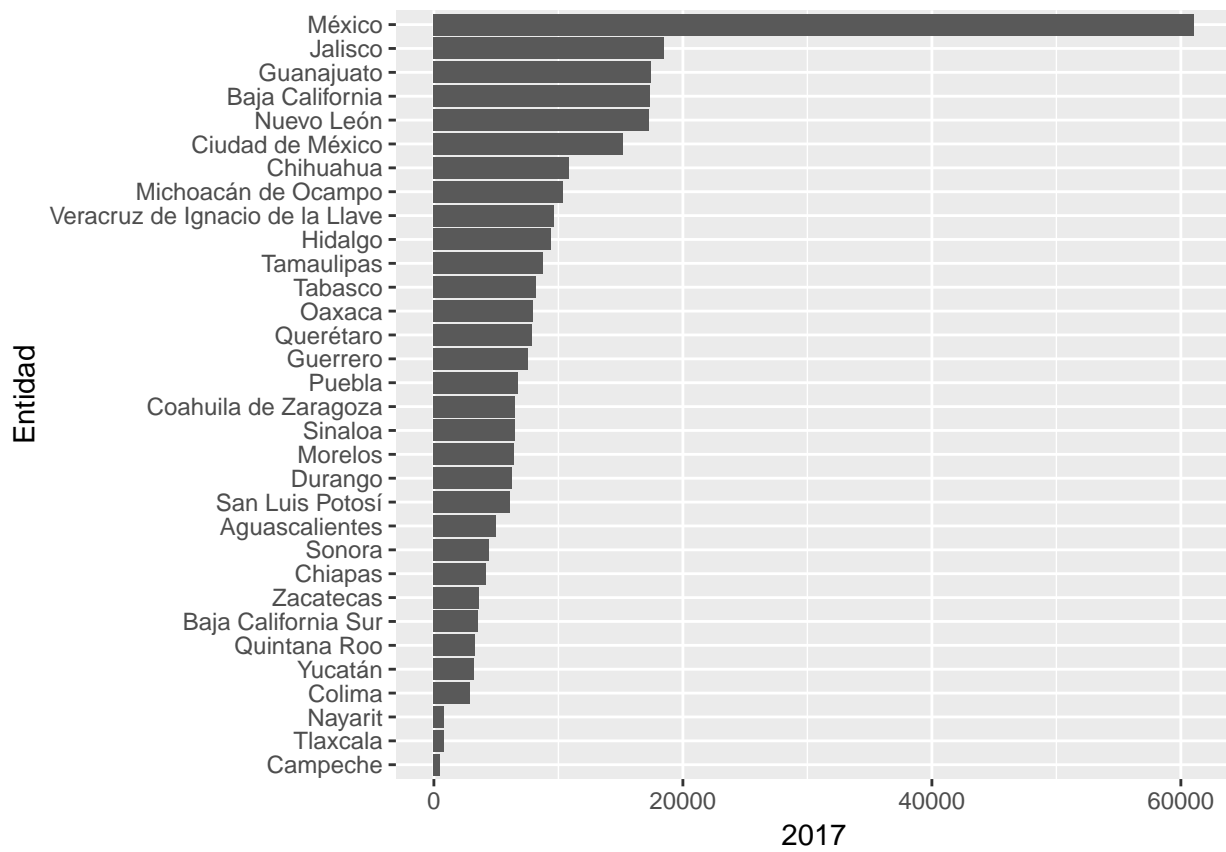
```
##
```

```
## $`2016`
```



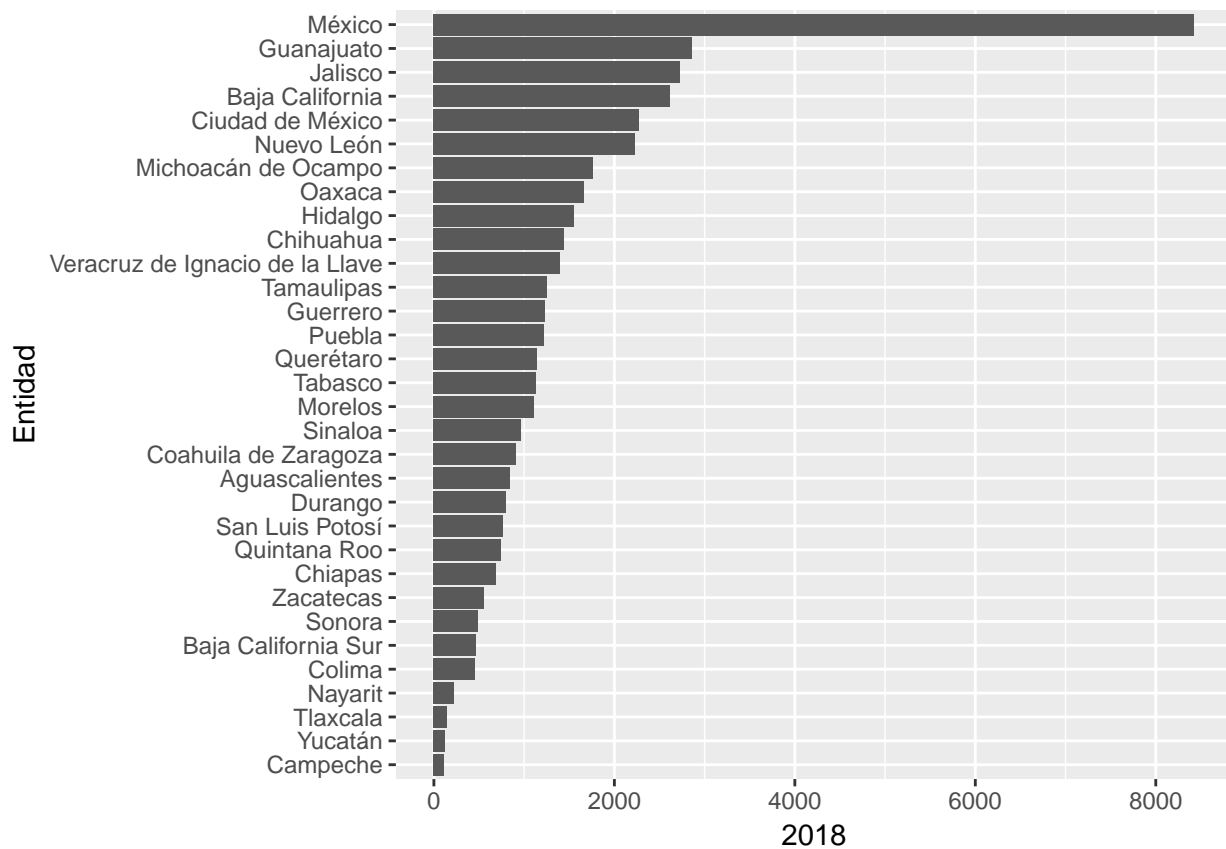
##

\$`2017`



##

\$`2018`



```
paths <- stringr::str_c(names(plots), ".pdf")
pwalk(list(paths, plots), ggsave, path = getwd())
getwd()
```



Tarea: desarrollar un script que incluya una función que lleve a cabo un proceso que generalmente llevarías a cabo usando otra herramienta, por ejemplo excel, sobre una tabla de datos propia. Explicar lo que se llevó a cabo.

Chapter 6

Modelado

6.1 ¿Qué aprendimos la clase pasada?

- Cómo analizar muestras complejas en R

En esta clase aprenderemos a:

- Utilizar modelos estadísticos para exploración de datos y para probar hipótesis.

6.2 Modelos

El objetivo primordial de los modelos estadísticos es el de proveer un resumen simple de baja dimensión de un conjunto de datos.

Idealmente un modelo captura la señal de interés (los patrones reales y generales del fenómeno que queremos estudiar) e ignora el “ruido” (e.g. variación aleatoria que no es de nuestro interés).

Vamos a proceder a estudiar cuál es la mecánica detrás de los modelos estadísticos concentrándonos en una familia muy importante de ellos: los modelos lineales.

6.2.1 Generación de hipótesis vs confirmación de hipótesis

Tradicionalmente, el objetivo del modelado estadístico fue la inferencia: plantear y comprobar hipótesis. Hacer esto correctamente no es complicado pero es difícil.

Hay que entender un par de ideas para poder llevar a cabo inferencia de una manera correcta:

- Cada observación de nuestro conjunto de datos base se puede usar para exploración o para confirmación pero nunca para ambas.
- Puedes usar una observación las veces que quieras para exploración, pero sólo la puedes usar una vez para confirmación. En cuanto uses una observación más de una vez estás automáticamente participando en un ejercicio exploratorio y no de confirmación.

Esto es necesario porque para confirmar una hipótesis se deben de usar datos independientes a los que se usaron para generar la hipótesis. De otra manera se está siendo optimista en cuanto a la solidez de la hipótesis.

Se verá más adelante pero para llevar a cabo un análisis de confirmación para modelos estadísticos una manera de proceder es partiendo tu conjunto de datos en tres subconjuntos:

- ~60% para explorar y entrenar tu modelo. Tienes permitido hacer lo que sea con este subconjunto de tus datos: manipularlo a gusto, visualizarlo, ajustarle un montón de modelos, etc.
- ~20% un conjunto de consulta. Sirve para comparar modelos o visualizaciones, pero no está permitido usarlo en el ajuste de modelos.
- ~20% es el conjunto de prueba final. Sólo se puede usar este conjunto de datos UNA vez, para probar tu modelo final.

Esta partición permite explorar tus datos de entrenamiento, ocasionalmente generando hipótesis candidatas que se contrastan usando el conjunto de consulta. Cuando se tiene confianza en tener un buen modelo, se puede evaluar en el conjunto de prueba.

6.2.2 Especificación de modelos

Hay dos partes fundamentales en un modelo:

- Primero, se define una familia de modelos que expresa de manera precisa, pero genérica, el patrón que se quiere capturar. Por ejemplo el patrón puede ser una línea recta:

$$Y = \beta_0 + \beta_1 * X$$

Aquí, Y y X son variables conocidos de tus datos. β_0 y β_1 son parámetros que pueden capturar distintos patrones.

- Luego, en el ejercicio de modelado se encuentran los parámetros a la hora de ajustar el modelo, de manera tal que el modelo se encuentre “lo más cerca posible” a tus datos, por ejemplo:

$$Y = 3 + 11X$$

Es importante recalcar que el modelo ajustado es el modelo más cercano a tus datos pero de una familia particular de modelos escogida a priori. Esto implica que se tiene el “mejor” modelo dado un cierto criterio muy particular. Definitivamente no implica que se tiene un buen modelo o que el modelo es “cierto”:

Box, George

Todos los modelos son falsos, pero algunos modelos son útiles.



6.2.3 Un primer modelo simple

Echémosle un vistazo a el conjunto de datos simulados `sim1` (vienen en el tidyverse)

```
# carguemos los paquetes
```

```
library(tidyverse)
```

```
library(modelr)
```

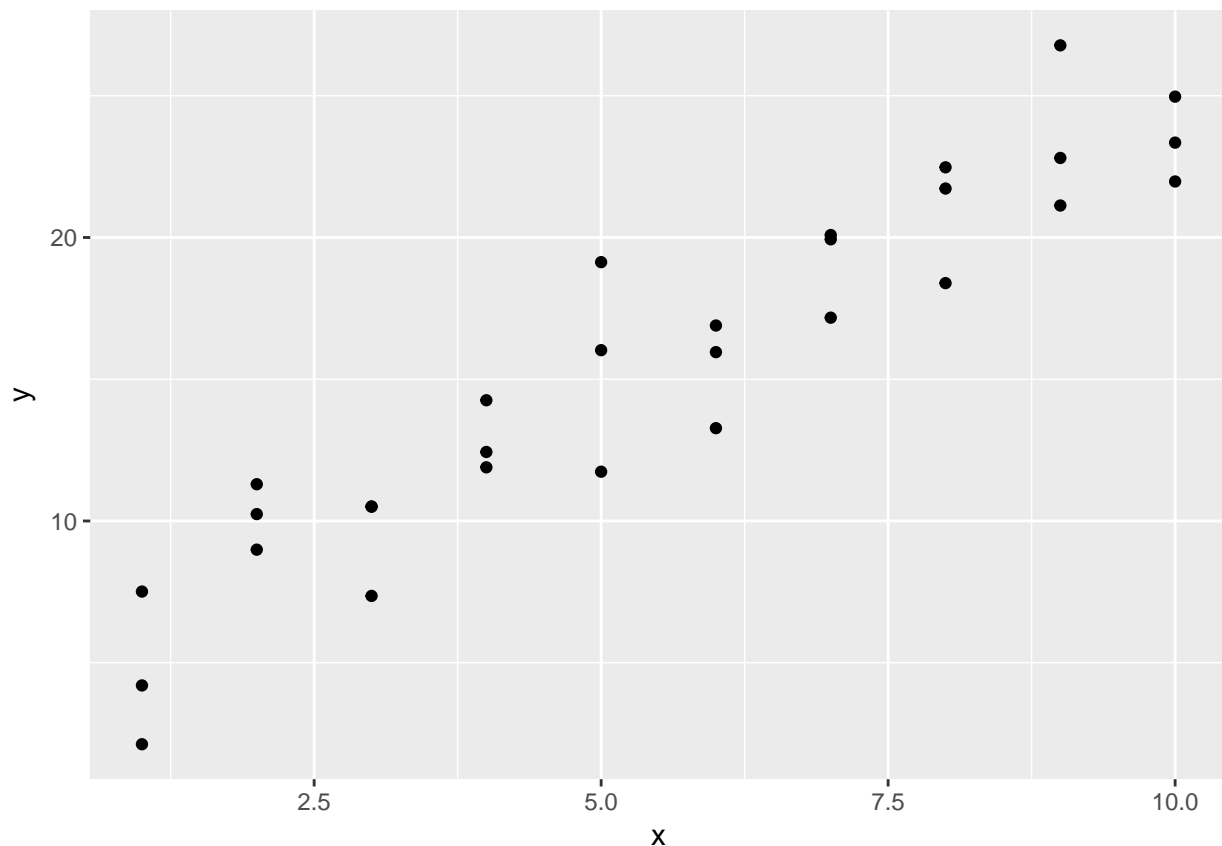
```
head(sim1)
```

```
## # A tibble: 6 x 2
##       x     y
##   <int> <dbl>
## 1     1  4.20
## 2     1  7.51
## 3     1  2.13
## 4     2  8.99
## 5     2 10.2
## 6     2 11.3
```

Como podrán ver, consta de dos variables continuas únicamente (x,y).

Podemos usar lo aprendido en clases pasadas para visualizar la relación entre estas dos variables:

```
ggplot(sim1, aes(x, y)) +
  geom_point()
```



Es claro que existe un patrón fuerte entre las dos variables (¿Qué tipo?).

Ahora es nuestra tarea proponer un modelo para este conjunto de datos que capture el patrón de la mejor manera posible.

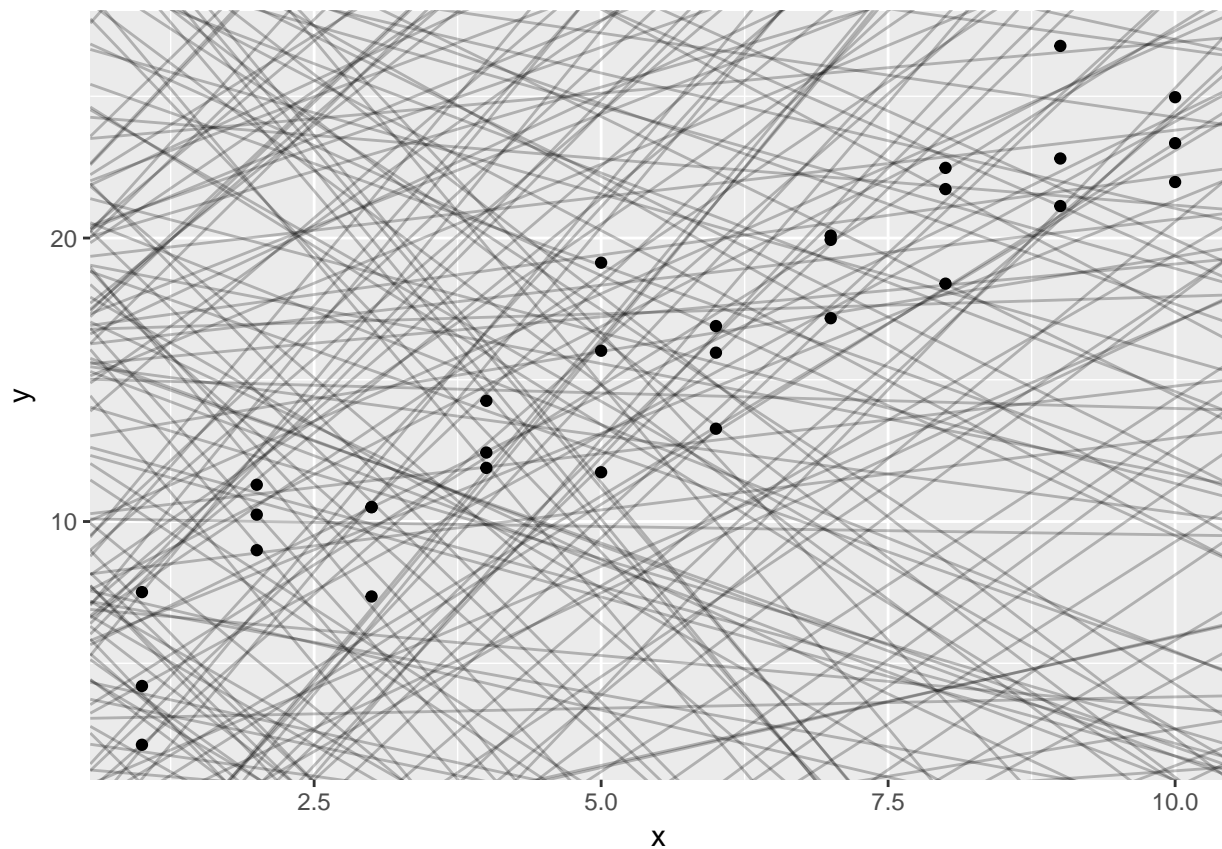


Usar la función `runif()`, consultar su ayuda y explicar con sus palabras qué hace.

La función anterior nos dejará generar una gran cantidad de parámetros para producir modelos candidatos (como hemos venido trabajando sólo son dos parámetros los que requerimos para datos con dos variables).

```
# parametros
modelos <- tibble(
  beta0 = runif(250, -20, 40),
  beta1 = runif(250, -5, 5)
)

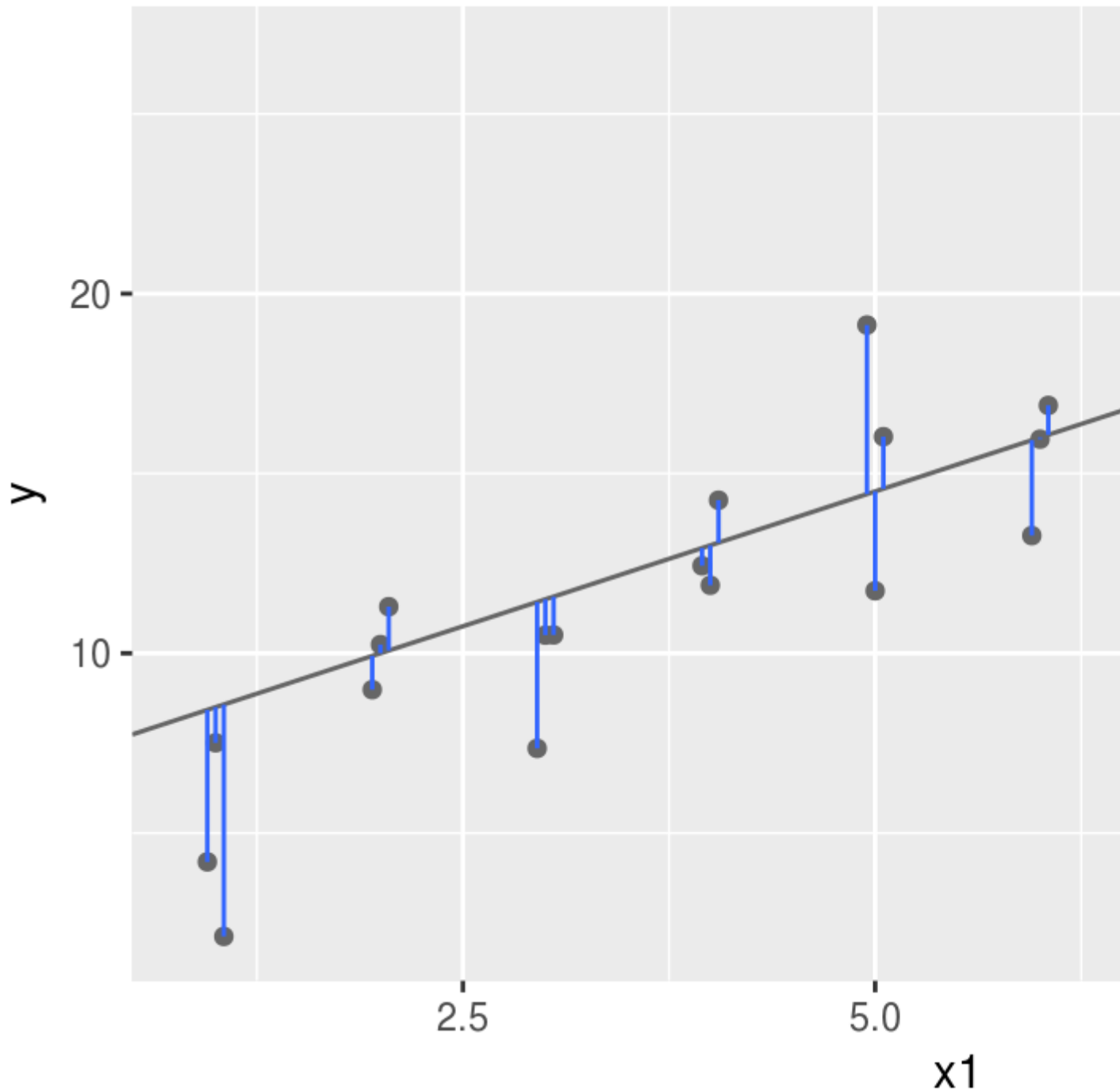
ggplot(sim1, aes(x, y)) +
  geom_abline(aes(intercept = beta0, slope = beta1), data = modelos, alpha = 1/4) +
  geom_point()
```



Pintamos 250 modelos candidatos sobre nuestros datos simulados ¡Muchos de ellos son realmente malos!

¿Cómo le hacemos para encontrar un buen modelo? ¿Cómo se formaliza la idea de que un buen modelo es uno que está “cerca” de nuestros datos? Necesitamos una manera de cuantificar la distancia de un modelo candidato a nuestros datos. Luego podemos buscar un modelo que encuentre valores para β_0 y β_1 de manera tal que el modelo esté a la menor distancia posible de los datos.

Un buen lugar para empezar es encontrar la distancia vertical de cada punto de nuestros datos a nuestro modelo (en este caso una línea recta).



La intuición detrás de el ajuste de varias familias de modelos es que si puedes establecer una función que defina la distancia entre el modelo y el conjunto de datos, entonces un algoritmo puede minimizar tal distancia, efectivamente encontrando los mejores parámetros para tu modelo. Una muy común es la suma de los cuadrados de las distancias verticales antes mostradas, eso se denomina ajuste por mínimos cuadrados.

R tiene varias herramientas y paquetes para trabajar con modelos. Una función clásica es `lm()` que permite ajustar modelos lineales como los que hemos venido viendo. R tiene varias manera de especificar los modelos que se buscan ajustar. Por ejemplo, con una estructura llamada fórmula: $Y \sim X$ que se traduce en:

$$Y = \beta_0 + \beta_1 * X$$

```
sim1_mod <- lm(y ~ x, data = sim1)

coef(sim1_mod)
```

```
## (Intercept)          x
##    4.220822    2.051533
```



Una desventaja de los modelos lineales ajustados con mínimos cuadrados es que por tener términos al cuadrado se vuelven sensibles a valores inusuales (muy grandes - muy chicos). Ajusta un modelo a los datos simulados con el código abajo y visualiza los resultados ¿Qué notas sobre estos modelos?

```
sim1a <- tibble(
  x = rep(1:10, each = 3),
  y = x * 1.5 + 6 + rt(length(x), df = 2)
)
```

6.2.4 Un modelo sobre datos más interesantes

En la carpeta datos se encuentran dos archivos. Uno de excel, con datos sobre ingresos per cápita por países. Otro, un archivo csv, con datos de esperanza de vida por países.



Carguen estos datos en sus espacio de trabajo.

```
## Parsed with column specification:
## cols(
##   iso3 = col_character(),
##   country_name = col_character(),
##   year = col_integer(),
##   age_name = col_character(),
##   sex_name = col_character(),
##   le = col_double(),
##   le_ui = col_character(),
##   hale = col_double(),
##   hale_ui = col_character()
## )
```

Con lo aprendido en las clases de manipulación y transformación de datos podemos convertir estas tablas de datos a otras que nos sirvan para estudiar la relación entre el ingreso per cápita y la esperanza de vida.

```
# seleccionar campos de datos de ingreso: sólo año 2010
ingreso = ingreso %>%
  select(1, '2010')

# renombrar columnas
colnames(ingreso) = c("pais", "GNI_capita_2010")

# filtrar datos de esperanza de vida: año 2010, ambos sexos, 20-24 años
esperanza = esperanza %>%
  filter(year==2010, age_name=="20-24 years", sex_name=="Both")
```



¿Qué usaríamos para juntar estas dos tablas en una sola?.

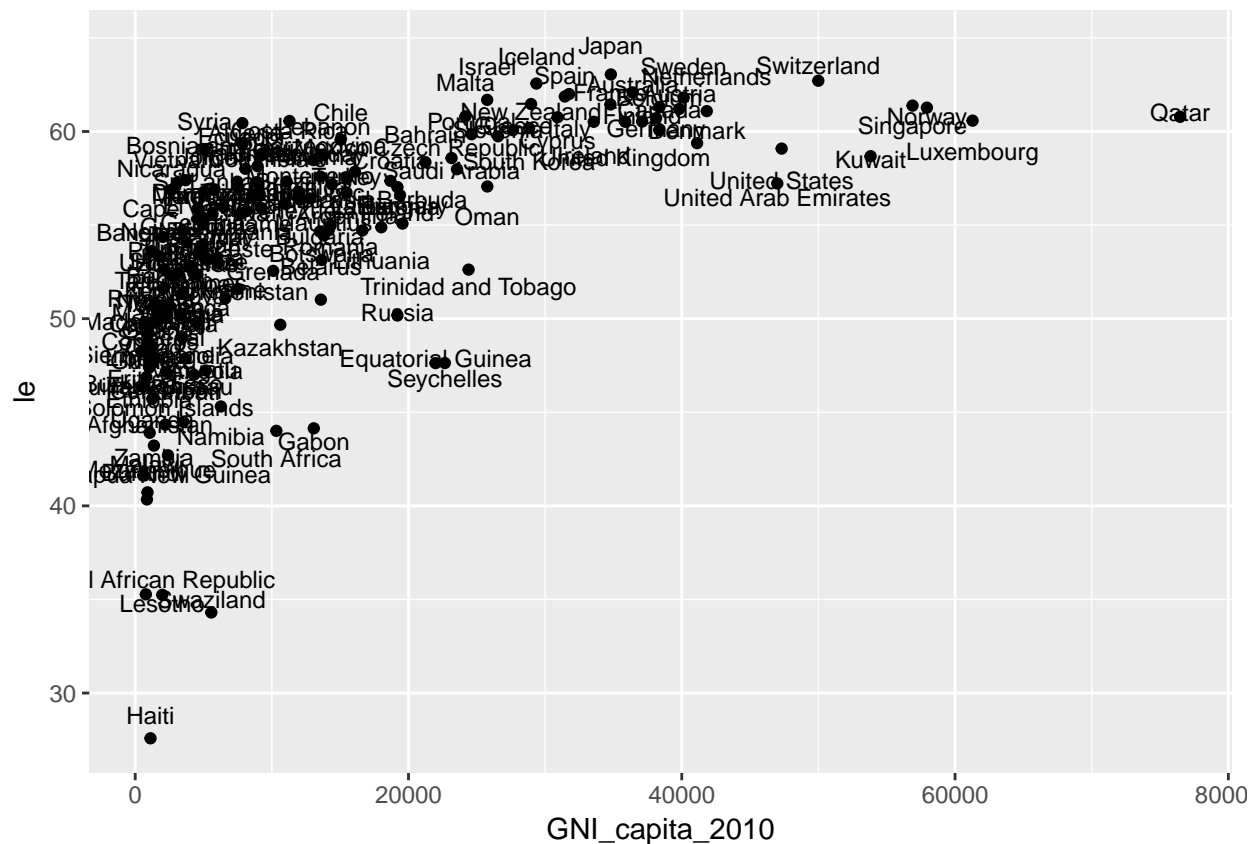
Si vemos la tabla de datos unificada, nos daremos cuenta que hay valores faltantes en el ingreso per cápita por país.

Podemos nuevamente usar `filter` para tirar tales registros.

```
# tirar datos faltantes
esperanza_ingreso = esperanza_ingreso %>%
  filter(!is.na(GNI_capita_2010))
```

Ahora estamos listos para ajustar un model lineal entre estas dos variables.

```
# visualizar datos
ggplot(esperanza_ingreso, aes(GNI_capita_2010, le)) +
  geom_point() +
  geom_text(aes(label=country_name), position=position_jitter(width=2, height=2), size=3)
```



```
# ajustar modelo
modelo <- lm(le ~ GNI_capita_2010, data = esperanza_ingreso)

# resumen del modelo
summary(modelo)

##
## Call:
## lm(formula = le ~ GNI_capita_2010, data = esperanza_ingreso)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -22.887  -2.196   1.015   3.441   8.216
```

```
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   5.018e+01  5.317e-01  94.374  <2e-16 ***
## GNI_capita_2010 2.620e-04  2.623e-05   9.991  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.939 on 158 degrees of freedom
## Multiple R-squared:  0.3872, Adjusted R-squared:  0.3833
## F-statistic: 99.82 on 1 and 158 DF,  p-value: < 2.2e-16
```

6.2.5 visualización de predicciones

Para visualizar las predicciones de nuestro modelo se puede usar `abline` como se hizo anteriormente. Una manera más general de hacerlo es primero generar una gradilla regular sobre la región donde se encuentran nuestros datos. La manera más fácil de hacer esto es usando la función `data_grid()` del paquete `modelr`. Su primer argumento es un data frame y para cada argumento subsecuente encuentra valores únicos y genera todas las combinaciones:

```
gradilla <- esperanza_ingreso %>%
  data_grid(GNI_capita_2010)

gradilla
```

```
## # A tibble: 155 x 1
##   GNI_capita_2010
##   <dbl>
## 1           440
## 2           540
## 3           580
## 4           720
## 5           780
## 6           820
## 7           860
## 8           900
## 9           950
## 10          990
## # ... with 145 more rows
```

Luego se pueden agregar predicciones, esto se hace con la función `add_predictions()`. Las agrega a una nueva columna del data frame.

```
gradilla <- gradilla %>%
  add_predictions(modelo)

gradilla
```

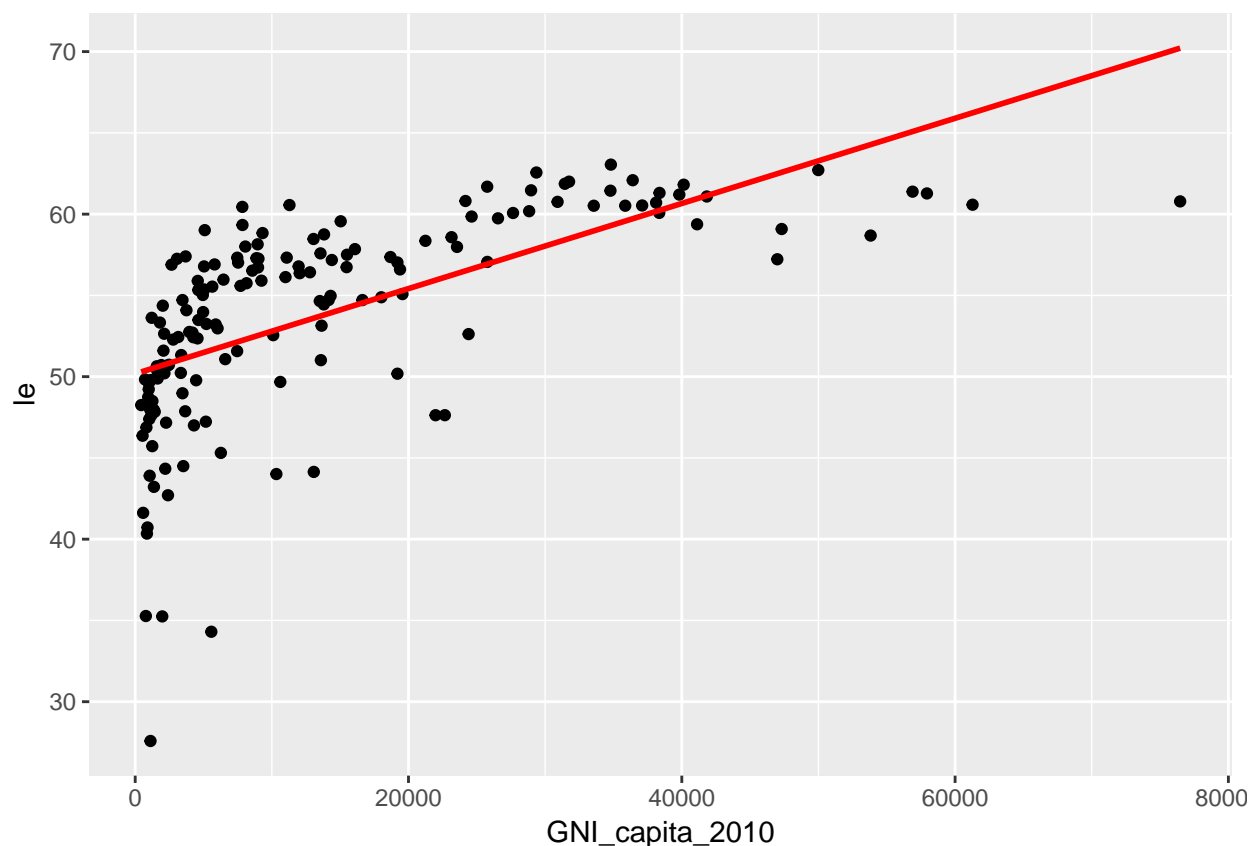
```
## # A tibble: 155 x 2
##   GNI_capita_2010 pred
##   <dbl> <dbl>
## 1           440  50.3
## 2           540  50.3
## 3           580  50.3
## 4           720  50.4
```

```
## 5          780  50.4
## 6          820  50.4
## 7          860  50.4
## 8          900  50.4
## 9          950  50.4
## 10         990  50.4
## # ... with 145 more rows
```

Ahora podemos visualizar las predicciones. Te podrás preguntar por qué hacer todo esto si lo resolvimos antes de manera sencilla utilizando `geom_abline()`. La ventaja de esta manera de hacerlo es que funciona para cualquier modelo, del más simple al más complejo.

Ver: <http://vita.had.co.nz/papers/model-vis.html>.

```
ggplot(esperanza_ingreso, aes(GNI_capita_2010)) +
  geom_point(aes(y = le)) +
  geom_line(aes(y = pred), data = gradilla, colour = "red", size = 1)
```



Se puede observar que el modelo no es particularmente bueno ¿Cómo se puede cuantificar lo bueno o malo que es? Una posibilidad es con la información arrojada por la función `summary`. Otra es checando los residuales y llevando a cabo un ejercicio de particionado de los datos entrenamiento/prueba como se mencionó al principio de la clase.

6.2.6 visualización de residuales

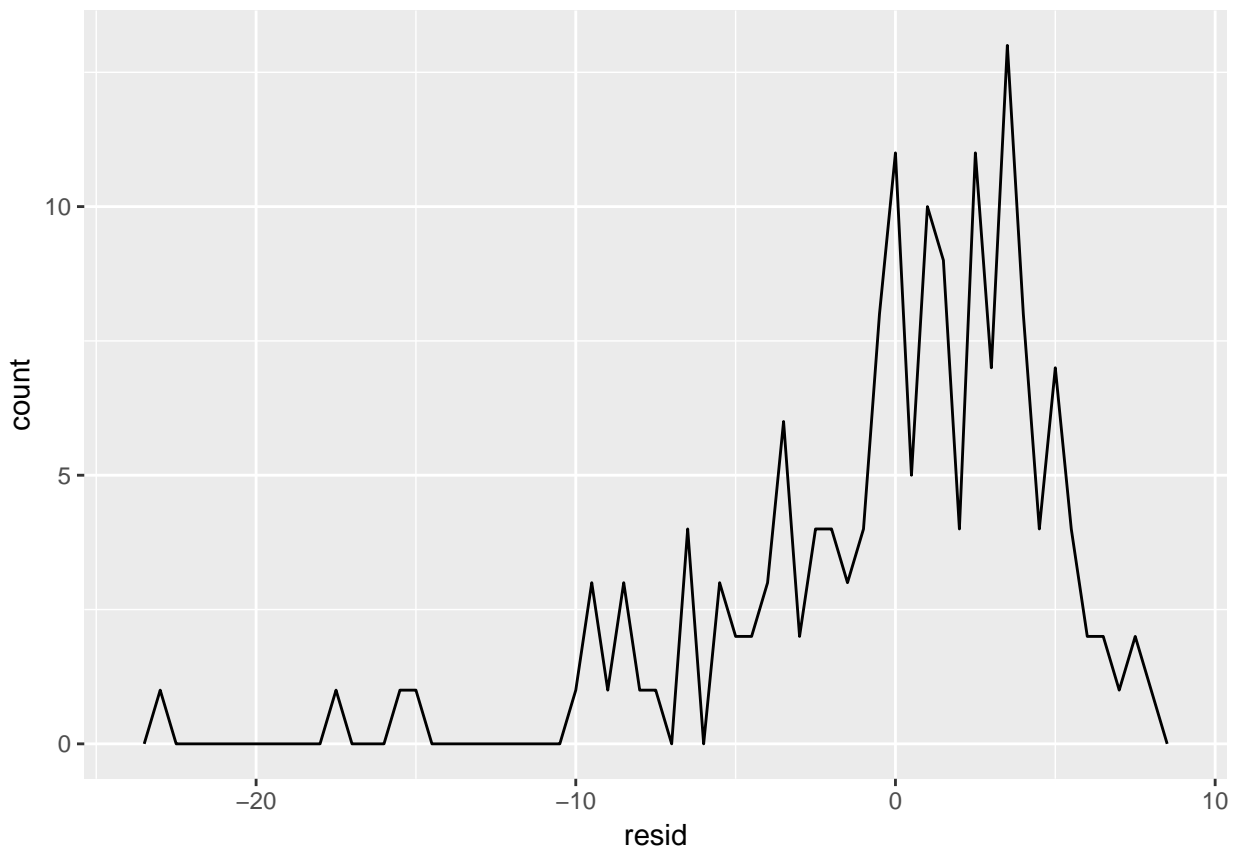
Los residuales son como el otro lado de la moneda de las predicciones. Las predicciones te indican qué patrón ha capturado el modelo, los residuales indican en cuánto ha fallado el modelo. Los residuales son simplemente las distancias entre los valores observados y predichos (por el modelo).

Agregamos los residuales a la tabla de datos con la función `add_residuals()`. Aunque aquí usaremos la tabla de datos original puesto que para computar los residuales se necesitan los valores observados.

```
esperanza_ingreso <- esperanza_ingreso %>%  
  add_residuals(modelo)
```

Hay varias cosas que se pueden estudiar con los residuales. Por ejemplo, si se grafica un polígono de frecuencias podemos visualizar su dispersión.

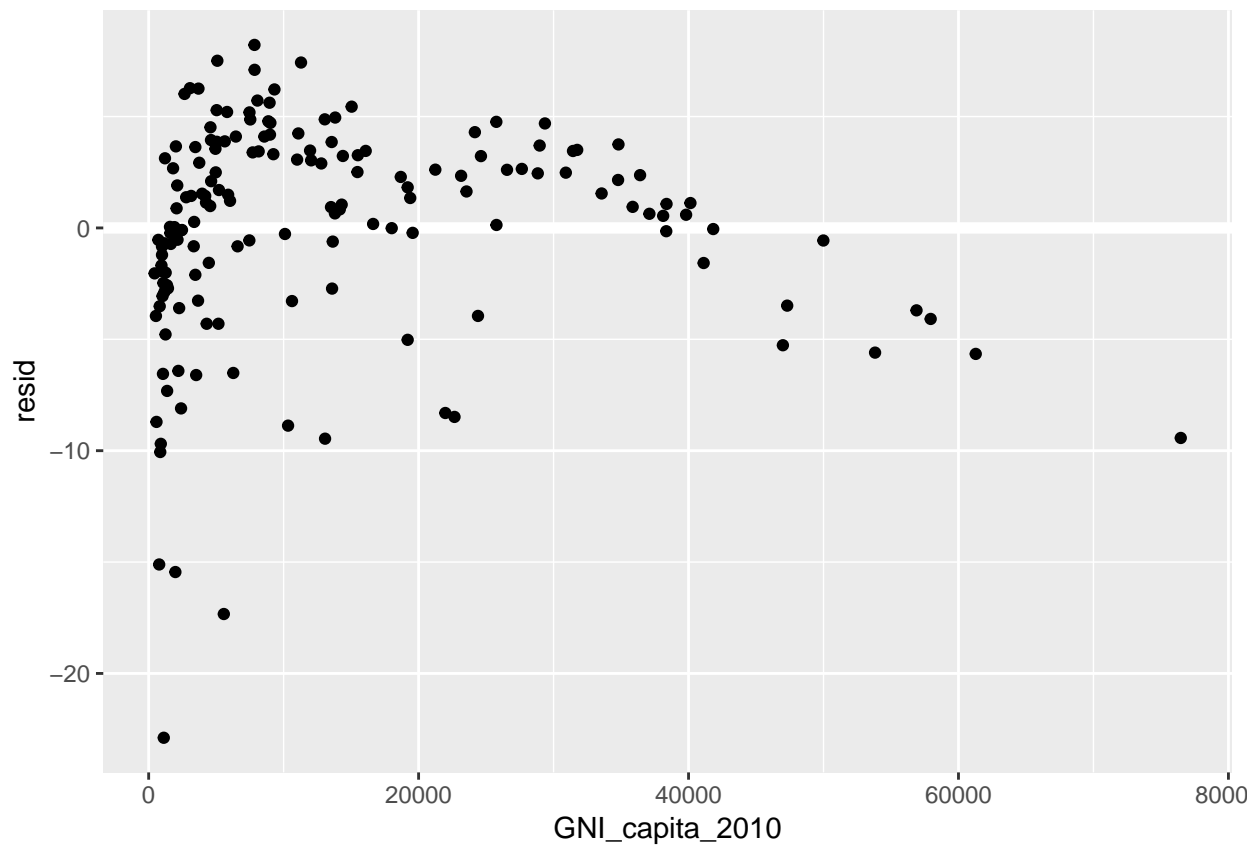
```
ggplot(esperanza_ingreso, aes(resid)) +  
  geom_freqpoly(binwidth = 0.5)
```



Esto puede ayudar a calibrar el modelo: ¿qué tan lejos están las predicciones de los valores originales observados? En los modelos lineales de este tipo, por construcción la media de los residuales es 0.

También es informativo hacer visualizaciones utilizando los residuales en vez de la variable dependiente original.

```
ggplot(esperanza_ingreso, aes(GNI_capita_2010, resid)) +  
  geom_ref_line(h = 0) +  
  geom_point()
```



En un buen modelo el gráfico anterior debe verse como ruido aleatorio. Aquí este no es el caso, por lo que es claro que no se han capturado bien los patrones del conjunto de datos.

Aún así, el modelo sí ayuda a concluir que hay una relación positiva entre ingreso y esperanza de vida.



¿Por qué crees que el modelo no logró capturar bien el patrón del conjunto de datos?.



Entregar un script donde se lleve a cabo un ejercicio de particionado del conjunto de datos para encontrar el mejor modelo multivariado que explique el ingreso (al revés de como se ha venido trabajando):

$$\text{ingreso} = \beta_0 + \beta_1 * \text{esperanzaVidaGrupoEdad}_1, \dots, \beta_n * \text{esperanzaVidaGrupoEdad}_n$$

Utilizar este modelo para estimar el ingreso de los datos faltantes 2010 ¿Se obtienen valores razonables?