

Notas de clase del curso de introducción a Data Science

Diego Kozlowski y Natsumi Shokida

2019-09-18

Contents

Introducción	6
1 Introducción a R	11
1.1 Explicación	11
1.2 Práctica Guiada	27
2 Tidyverse	33
2.1 Explicación	33
2.2 Práctica Guiada	44
3 Programacion Funcional	49
3.1 Explicación	49
3.2 Práctica Guiada	58
4 Visualización de la información	69
4.1 Explicación	69
4.2 Práctica Guiada	84
5 Documentación en R	97
5.1 Explicación	97
5.2 Práctica Guiada	104
6 Shiny apps	109
6.1 Explicación	110
6.2 Práctica Guiada	112
7 Probabilidad y Estadística	119
7.1 Explicación	119
7.2 Práctica Guiada	140
8 Modelo Lineal	147
8.1 Explicación	147
8.2 Práctica Guiada	165

9	Diseño y análisis de encuestas	175
9.1	Explicación	175
9.2	Práctica Guiada	175

Introducción



Presentación

En los últimos años se han difundido muchas herramientas estadísticas novedosas para el análisis de información socioeconómica y geográfica. En particular el software denominado “R”, por tratarse de un software libre, se extiende cada vez más en diferentes disciplinas y recibe el aporte de investigadores e investigadoras en todo el mundo, multiplicando sistemáticamente sus capacidades.

Este programa se destaca, entre otras cosas, por su capacidad de trabajar con grandes volúmenes de información, utilizar múltiples bases de datos en simultáneo, generar reportes, realizar gráficos a nivel de publicación y por su comunidad de usuarios que publican sus sintaxis y comparten sus problemas, hecho que potencia la capacidad de consulta y de crecimiento. A su vez, la expresividad del lenguaje permite diseñar funciones específicas que permiten optimizar de forma personalizada el trabajo cotidiano con R.

Objetivos del curso

El presente Taller tiene como objetivo principal introducir a los participantes en la ciencia de datos, sobre la base de la utilización del lenguaje R aplicado procesamiento de diferentes bases de datos provistas por el programa de Gobierno Abierto y la Encuesta Permanente de Hogares (EPH) - INDEC. Se apunta a brindar las herramientas necesarias para la gestión de la información, presentación de resultados y algunas técnicas de modelado de datos, de forma tal que los participantes puedan luego avanzar por su cuenta a técnicas más avanzadas.

Webpage

Temario:

Eje 1. Programación en R

clase 1: Introducción al entorno R:

- Descripción del programa “R”. Lógica sintáctica del lenguaje y comandos básicos
- Presentación de la plataforma RStudio para trabajar en “R”
- Caracteres especiales en “R”
- Operadores lógicos y aritméticos
- Definición de Objetos: Valores, Vectores y DataFrames
- Tipos de variable (numérica, de caracteres, lógicas)
- Lectura y Escritura de Archivos

clase 2: Tidyverse:

- Limpieza de Base de datos: Renombrar y recodificar variables, tratamiento de valores faltantes (missing values/ NA's)

- Seleccionar variables, ordenar y agrupar la base de datos para realizar cálculos
- Creación de nuevas variables
- Aplicar filtros sobre la base de datos
- Construir medidas de resumen de la información
- Tratamiento de variables numéricas (edad, ingresos, horas de trabajo, cantidad de hijos / componentes del hogar, entre otras).

clase 3: Programación funcional

- Estructuras de código condicionales
- Loops
- Creación de funciones a medida del usuario
- Librería purrr para programación funcional

Eje 2. Presentación de resultados

clase 4: Visualización de la información

- Gráficos básicos de R (función “plot”): Comandos para la visualización ágil de la información
- Gráficos elaborados en R (función “ggplot”):
- Gráficos de línea, barras, Boxplots y distribuciones de densidad
- Parámetros de los gráficos: Leyendas, ejes, títulos, notas, colores
- Gráficos con múltiples cruces de variables.

clase 5: Documentación en R

- Manejo de las extensiones del software “Rmarkdown” y “RNotebook” para elaborar documentos de trabajo, presentaciones interactivas e informes:
- Opciones para mostrar u ocultar código en los reportes
- Definición de tamaño, títulos y formato con el cual se despliegan los gráficos y tablas en el informe
- Caracteres especiales para incluir múltiples recursos en el texto del informe: Links a páginas web, notas al pie, enumeraciones, cambios en el formato de letra (tamaño, negrita, cursiva)
- Código embebido en el texto para automatización de reportes

clase 6: Shiny

- Shiny como reportes dinámicos
- Su utilidad para el análisis exploratorio
- Lógica de servidor- interfaz de usuario
- Inputs- Outputs, funciones reactivas, widgets.

Eje 3. Estadística

clase 7: Estadística descriptiva

- Introducción a probabilidad

- Introducción a distribuciones
- El problema de la inversión
- Estadística
- Población y muestra
- Estimadores puntuales, tests de hipótesis
- Boxplots, histogramas y kernels

clase 8: Correlación y Modelo Lineal

- Análisis de correlación.
- Presentación conceptual del modelo lineal
- El modelo lineal desde una perspectiva computacional
- Supuestos del modelo lineal
- Modelo lineal en R
- Modelo lineal en el tidyverse

Eje 4. Clases temáticas

clase 9: Análisis de encuestas

- Introducción al diseño de encuestas
- Presentación de la Encuesta Permanente de Hogares
- Generación de estadísticos de resumen en muestras estratificadas
- Utilización de los ponderadores

clase 10: Mapas

- Utilización de información geográfica en R
- Elaboración de mapas
- gestión de shapefiles

clase 11: Text Mining

- Introducción al análisis de textos
- Limpieza
- Preprocesamiento
- BoW
- Stopwords
- TF-IDF
- Wordcloud
- Escraqueo de Twitter

Bibliografía de consulta

- GWickham, H., & Grolemond, G. (2016). R for data science: import, tidy, transform, visualize, and model data. " O'Reilly Media, Inc.". <https://es.r4ds.hadley.nz/>
- James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). An introduction to statistical learning. New York: springer. <http://faculty.marshall.usc.edu/gareth-james/ISL/>

- Wickham, Hadley. ggplot2: elegant graphics for data analysis. Springer, 2016. <https://ggplot2-book.org/>

Librerias a instalar

```
install.packages(c("tidyverse","openxlsx","xlsx","ggplot2","GGally","ggridges","treemap"))
```

Chapter 1

Introducción a R

En esta primera clase revisaremos los fundamentos de R base y el entorno de RStudio. El objetivo es poder comenzar a utilizar el programa, abrir archivos y empezar a experimentar para ganar confianza.

- Descripción del programa *R*. Lógica sintáctica del lenguaje y comandos básicos
- Presentación de la plataforma RStudio para trabajar en *R*
- Caracteres especiales en *R*
- Operadores lógicos y aritméticos
- Definición de objetos: valores, vectores y DataFrames
- Tipos de variable (numéricas, de caracteres, lógicas)
- Lectura y escritura de archivos

1.1 Explicación

1.1.1 ¿Qué es R?

- Lenguaje para el procesamiento y análisis estadístico de datos
- Software Libre
- Sintaxis Básica: R base
- Sintaxis incremental¹: El lenguaje se va ampliando por aportes de Universidades, investigadores/as, usuarios/as y empresas privadas, organizados en librerías (o paquetes)
- Comunidad web muy grande para realizar preguntas y despejar dudas. Por ejemplo, en el caso de Buenos Aires contamos con R-Ladies Buenos Aires y RenBaires.
- Gráficos con calidad de publicación

¹Más allá de los comandos elementales, comandos más sofisticados tienen muchas versiones, y algunas quedan en desuso en el tiempo.



Figure 1.1: <https://cran.r-project.org/>

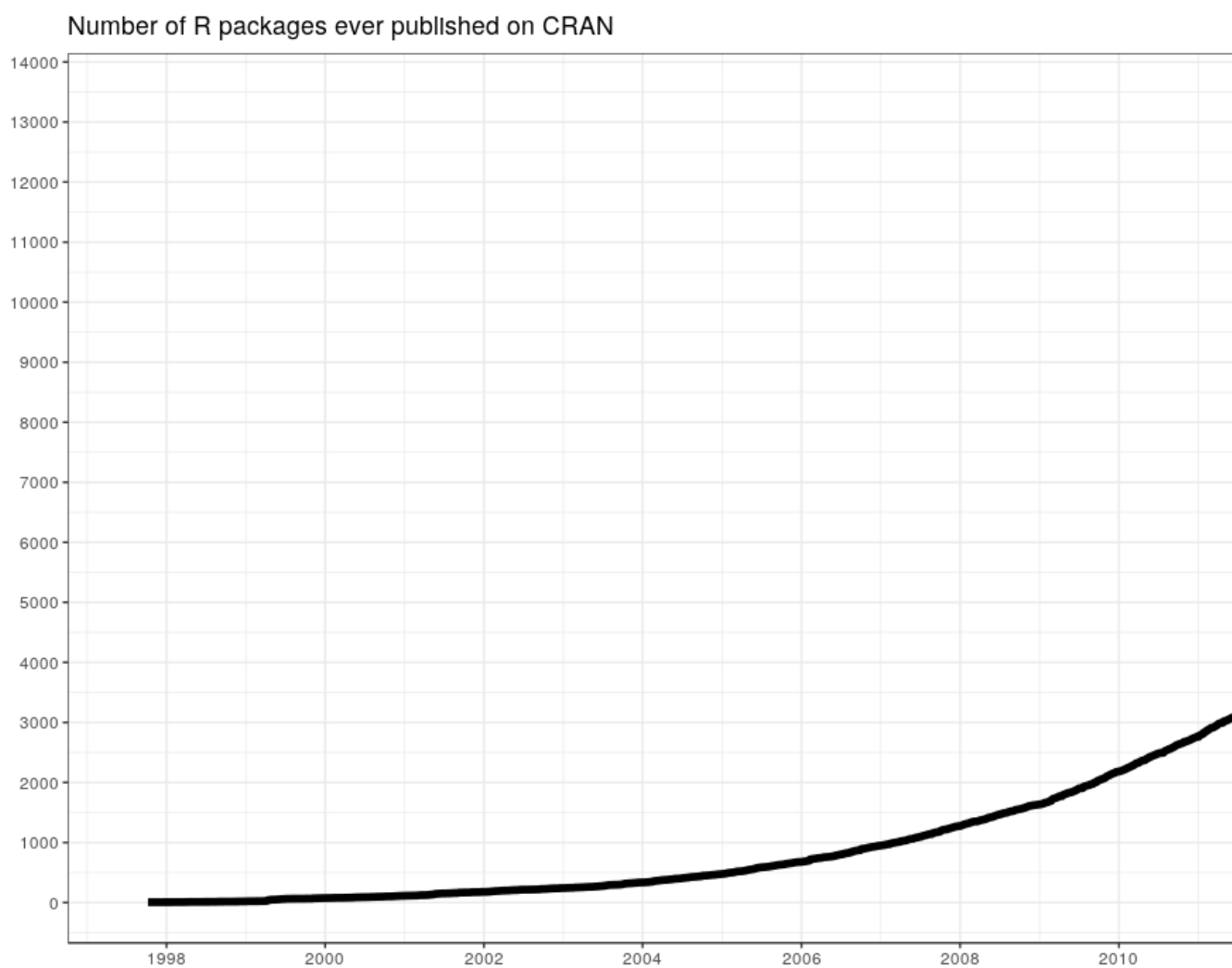


Figure 1.2: fuente: <https://gist.github.com/daroczig/3cf06d6db4be2bbe3368>



Figure 1.3: <https://www.rstudio.com/>

Uno de los *entornos* más cómodos para utilizar el *lenguaje R* es el *programa R studio*.

- Rstudio es una empresa que produce productos asociados al lenguaje R, como el programa sobre el que corremos los comandos, y extensiones del lenguaje (librerías).
- El programa es *gratis* y se puede bajar de la página oficial

1.1.2 Lógica sintáctica.

1.1.2.1 Definición de objetos

Los **Objetos/Elementos** constituyen la categoría esencial del R. De hecho, todo en R es un objeto, y se almacena con un nombre específico que **no debe poseer espacios**. Un número, un vector, una función, la progresión de letras del abecedario, una base de datos, un gráfico, constituyen para R objetos de distinto tipo. Los objetos que vamos creando a medida que trabajamos pueden visualizarse en el panel derecho superior de la pantalla (el *Environment*).

El operador `<-` (**Alt + Guión**) sirve para definir un objeto. **A la izquierda** del `<-` debe ubicarse el nombre que tomará el elemento a crear. **Del lado derecho** debe ir la definición del mismo.

```
A <- 1
```

Por ejemplo, podemos crear el elemento **A**, cuyo valor será 1. Para esto, debemos *correr* el código presionando **Ctrl + Enter**, con el cursor ubicado en

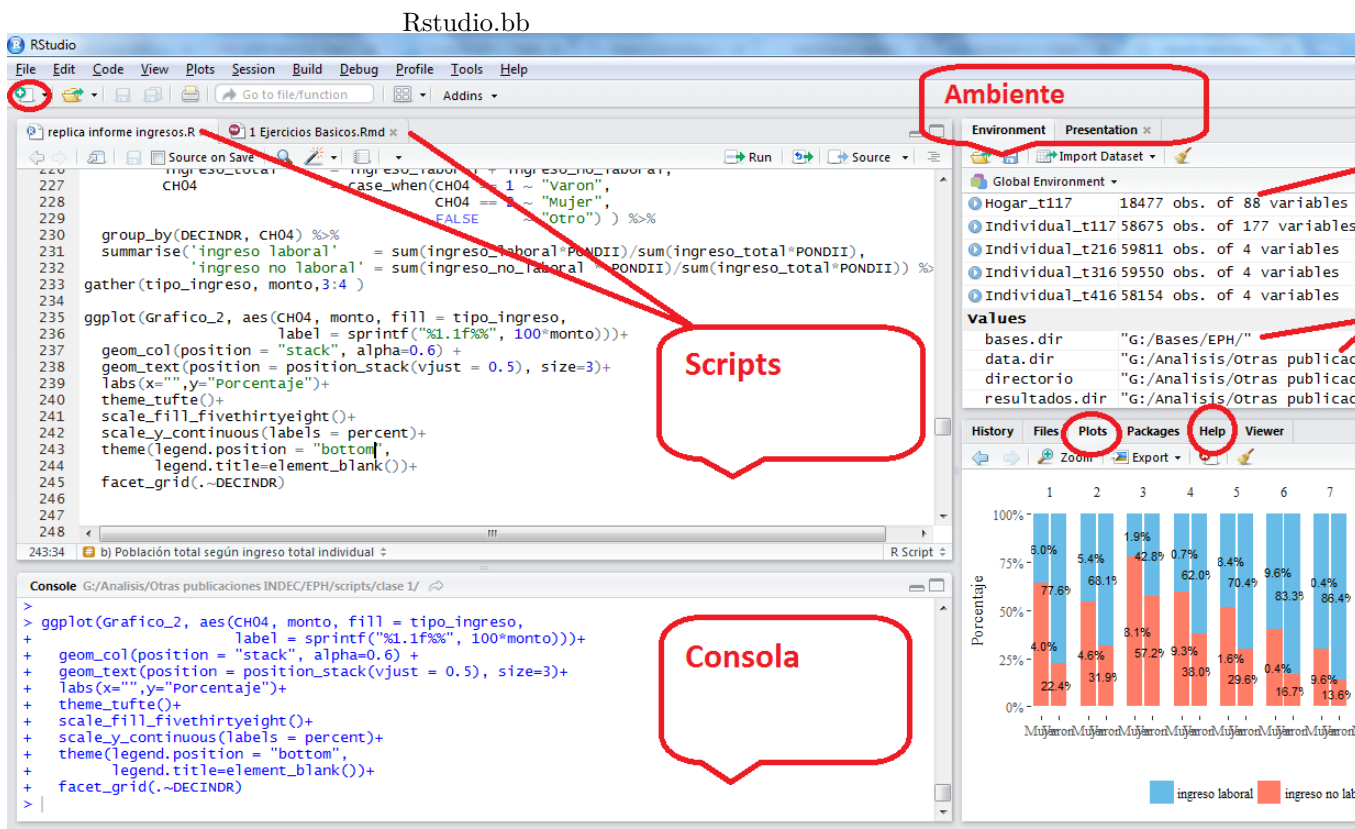


Figure 1.4: Pantalla Rstudio

cualquier parte de la línea. Al definir un elemento, el mismo queda guardado en el ambiente del programa, y podrá ser utilizado posteriormente para observar su contenido o para realizar una operación con el mismo.

```
A
```

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

```
A+6
```

```
## [1] 7
```

Al correr una línea con el nombre del objeto, la consola del programa nos muestra su contenido. Entre corchetes observamos el número de orden del elemento en cuestión. Si corremos una operación, la consola nos muestra el resultado de la misma.

El operador `=` es **equivalente** a `<-`, pero en la práctica no se utiliza para la definición de objetos.

```
B = 2
```

```
B
```

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

`<-` es un operador **Unidireccional**, es decir que:

`A <- B` implica que **A** va tomar como valor el contenido del objeto **B**, y no al revés.

```
A <- B
```

```
A      # Ahora A toma el valor de B, y B continúa conservando el mismo valor
```

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

```
B
```

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

1.1.3 R base

Con *R base* nos referimos a los comandos básicos que vienen incorporados en el R, sin necesidad de cargar librerías.

1.1.3.1 Operadores lógicos:

- `>` (mayor a-)
- `>=` (mayor o igual a-)
- `<` (menor a-)
- `<=` (menor o igual a-)
- `==` (igual a-)
- `!=` (distinto a-)


```
# Redefinimos los valores A y B
A <- 10
B <- 20

# Realizamos comparaciones lógicas
A > B
```

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

```
A >= B
```

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

```
A < B
```

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

```
A <= B
```

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

```
A == B
```

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

```
A != B
```

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

```
C <- A != B
```

```
C
```

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

Como muestra el último ejemplo, el resultado de una operación lógica puede almacenarse como el valor de un objeto.

1.1.3.2 Operadores aritméticos:

```
#suma
A <- 5+6
A
```

```
## [1] 11
```

```
#Resta
B <- 6-8
B
```

```
## [1] -2
```

```
#cociente
C <- 6/2.5
C
```

```
## [1] 2.4
#multiplicacion
D <- 6*2.5
D
```

```
## [1] 15
```

1.1.3.3 Funciones:

Las funciones son series de procedimientos estandarizados, que toman como input determinados argumentos a fijar por el usuario, y devuelven un resultado acorde a la aplicación de dichos procedimientos. Su lógica de funcionamiento es:

```
funcion(argumento1 = arg1, argumento2 = arg2)
```

A lo largo del curso iremos viendo numerosas funciones, según lo requieran los distintos ejercicios. Sin embargo, veamos ahora algunos ejemplos para comprender su funcionamiento:

- `paste()` : concatena una serie de caracteres, pudiendo indicarse cómo separar a cada uno de ellos
- `paste0()`: concatena una serie de caracteres sin separar
- `sum()`: suma de todos los elementos de un vector
- `mean()` promedio aritmético de todos los elementos de un vector

```
paste("Pega", "estas", 4, "palabras", sep = " ")
```

```
## [1] "Pega estas 4 palabras"
```

```
#Puedo concatenar caracteres almacenados en objetos
paste(A, B, C, sep = "**")
```

```
## [1] "11**-2**2.4"
```

```
# Paste0 pega los caracteres sin separador
paste0(A, B, C)
```

```
## [1] "11-22.4"
```

```
1:5
```

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

```
sum(1:5)
```

```
## [1] 15
```

```
mean(1:5, na.rm = TRUE)
```

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

1.1.3.4 Caracteres especiales

- R es sensible a mayúsculas y minúsculas, tanto para los nombres de las variables, como para las funciones y parámetros.
- Los **espacios en blanco** y los **carriage return** (*enter*) no son considerados por el lenguaje. Los podemos aprovechar para emprolijar el código y que la lectura sea más simple².
- El **numeral #** se utiliza para hacer comentarios. Todo lo que se escribe después del **#** no es interpretado por R. Se debe utilizar un **#** por cada línea de código que se desea anular
- Los **corchetes []** se utilizan para acceder a un objeto:
 - en un vector[nº orden]
 - en una tabla[filas, columnas]
 - en una lista[nº elemento]
- el signo **\$** también es un método de acceso. Particularmente, en los dataframes, nos permitira acceder a una determinada columna de una tabla
- Los **paréntesis()** se utilizan en las funciones para definir los parámetros.
- Las **comas ,** se utilizan para separar los parametros al interior de una función.

1.1.4 Objetos:

Existe una gran cantidad de objetos distintos en R, en lo que respecta al curso trabajaremos principalmente con 3 de ellos:

- Valores
- Vectores
- Data Frames
- Listas

1.1.4.1 Valores

Los valores y vectores pueden ser a su vez de distintas *clases*:

Numeric

```
A <- 1
class(A)
```

```
## [1] "numeric"
```

Character

²veremos que existen ciertas excepciones con algunos paquetes más adelante.

```
A <- paste('Soy', 'una', 'concatenación', 'de', 'caracteres', sep = " ")
A
```

```
## [1] "Soy una concatenación de caracteres"
```

```
class(A)
```

```
## [1] "character"
```

Factor

```
A <- factor("Soy un factor, con niveles fijos")
```

```
class(A)
```

```
## [1] "factor"
```

La diferencia entre un *character* y un *factor* es que el último tiene solo algunos valores permitidos (levels), con un orden interno predefinido (el cual, por ejemplo, se respetará a la hora de realizar un gráfico)

1.1.4.2 Vectores

Para crear un **vector** utilizamos el comando `c()`, de combinar.

```
C <- c(1, 3, 4)
```

```
C
```

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

Podemos sumarle 2 a cada elemento del **vector** anterior

```
C <- C + 2
```

```
C
```

```
## [1] 3 5 6
```

O sumarle 1 al primer elemento, 2 al segundo, y 3 al tercer elemento del **vector** anterior

```
D <- C + 1:3 # esto es equivalente a hacer 3+1, 5+2, 6+3
```

```
D
```

```
## [1] 4 7 9
```

`1:3` significa que queremos todos los números enteros desde 1 hasta 3.

Podemos crear un **vector** que contenga las palabras: “Carlos”, “Federico”, “Pedro”

```
E <- c("Carlos", "Federico", "Pedro")
```

```
E
```

```
## [1] "Carlos" "Federico" "Pedro"
```

Para acceder a algún elemento del vector, podemos buscarlo por su número de orden, entre []

```
E[2]
```

```
## [1] "Federico"
```

Si nos interesa almacenar dicho valor, al buscarlo lo asignamos a un nuevo objeto, dándole el nombre que deseemos

```
elemento2 <- E[2]
```

```
elemento2
```

```
## [1] "Federico"
```

Para **borrar** un objeto del ambiente de trabajo, utilizamos el comando *rm()*

```
rm(elemento2)
```

```
elemento2
```

```
## Error in eval(expr, envir, enclos): object 'elemento2' not found
```

También podemos cambiar el texto del segundo elemento de E, por el texto “Pablo”

```
E[2] <- "Pablo"
```

```
E
```

```
## [1] "Carlos" "Pablo" "Pedro"
```

1.1.5 Data Frames

Un Data Frame es una tabla de datos, donde cada columna representa una variable, y cada fila una observación.

Este objeto suele ser central en el proceso de trabajo, y suele ser la forma en que se cargan datos externos para trabajar en el ambiente de R, y en que se exportan los resultados de nuestros trabajo.

También se puede crear como la combinación de N vectores de igual tamaño. Por ejemplo, tomamos algunos valores del Índice de salarios

```
INDICE <- c(100, 100, 100,
            101.8, 101.2, 100.73,
            102.9, 102.4, 103.2)
```

```
FECHA <- c("Oct-16", "Oct-16", "Oct-16",
           "Nov-16", "Nov-16", "Nov-16",
           "Dic-16", "Dic-16", "Dic-16")
```

```
GRUPO <- c("Privado_Registrado", "Público", "Privado_No_Registrado",
           "Privado_Registrado", "Público", "Privado_No_Registrado",
```

```

"Privado_Registrado", "Público", "Privado_No_Registrado")

Datos <- data.frame(INDICE, FECHA, GRUPO)
Datos

##  INDICE  FECHA          GRUPO
## 1 100.00 Oct-16   Privado_Registrado
## 2 100.00 Oct-16         Público
## 3 100.00 Oct-16 Privado_No_Registrado
## 4 101.80 Nov-16   Privado_Registrado
## 5 101.20 Nov-16         Público
## 6 100.73 Nov-16 Privado_No_Registrado
## 7 102.90 Dic-16   Privado_Registrado
## 8 102.40 Dic-16         Público
## 9 103.20 Dic-16 Privado_No_Registrado

```

Tal como en un **vector** se ubica a los elementos mediante `[]`, en un **dataframe** se obtienen sus elementos de la forma `[fila, columna]`.

Otra opción es especificar la columna, mediante el operador `$`, y luego seleccionar dentro de esa columna el registro deseado mediante el número de orden.

```
Datos$FECHA
```

```
## [1] Oct-16 Oct-16 Oct-16 Nov-16 Nov-16 Nov-16 Dic-16 Dic-16 Dic-16
## Levels: Dic-16 Nov-16 Oct-16
```

```
Datos[3,2]
```

```
## [1] Oct-16
## Levels: Dic-16 Nov-16 Oct-16
```

```
Datos$FECHA[3]
```

```
## [1] Oct-16
## Levels: Dic-16 Nov-16 Oct-16
```

¿que pasa si hacemos `Datos$FECHA[3,2]` ?

```
Datos$FECHA[3,2]
```

```
## Error in `[.default`(Datos$FECHA, 3, 2): incorrect number of dimensions
```

Nótese que el último comando tiene un número incorrecto de dimensiones, porque estamos refiriendonos 2 veces a la columna FECHA.

Acorde a lo visto anteriormente, el acceso a los **dataframes** mediante `[]` puede utilizarse para realizar filtros sobre la base, especificando una condición para las filas. Por ejemplo, puedo utilizar los `[]` para conservar del **dataframe** `Datos` unicamente los registros con fecha de Diciembre 2016:

```
Datos[Datos$FECHA=="Dic-16",]
```

```
##  INDICE  FECHA                GRUPO
## 7  102.9 Dic-16      Privado_Registrado
## 8  102.4 Dic-16                Público
## 9  103.2 Dic-16 Privado_No_Registrado
```

La lógica del paso anterior sería: Accedo al dataframe `Datos`, pidiendo únicamente conservar las filas (por eso la condición se ubica a la *izquierda* de la `,`) que cumplan el requisito de pertenecer a la categoría “**Dic-16**” de la variable **FECHA**.

Aún más, podría aplicar el filtro y al mismo tiempo identificar una variable de interés para luego realizar un cálculo sobre aquella. Por ejemplo, podría calcular la media de los índices en el mes de Diciembre.

```
### Por separado
```

```
Indices_Dic <- Datos$INDICE[Datos$FECHA=="Dic-16"]
Indices_Dic
```

```
## [1] 102.9 102.4 103.2
```

```
mean(Indices_Dic)
```

```
## [1] 102.8333
```

```
### Todo junto
```

```
mean(Datos$INDICE[Datos$FECHA=="Dic-16"])
```

```
## [1] 102.8333
```

La lógica de esta sintaxis sería: “Me quedo con la variable **INDICE**, cuando la variable **FECHA** sea igual a “**Dic-16**”, luego calculo la media de dichos valores”.

1.1.6 Listas

Contienen una concatenación de objetos de cualquier tipo. Así como un vector contiene valores, un dataframe contiene vectores, una lista puede contener dataframes, pero también vectores, o valores, y *todo ello a la vez*.

```
superlista <- list(A,B,C,D,E,FECHA, DF = Datos, INDICE, GRUPO)
superlista
```

```
## [[1]]
## [1] Soy un factor, con niveles fijos
## Levels: Soy un factor, con niveles fijos
##
## [[2]]
## [1] -2
##
```

```
## [[3]]
## [1] 3 5 6
##
## [[4]]
## [1] 4 7 9
##
## [[5]]
## [1] "Carlos" "Pablo" "Pedro"
##
## [[6]]
## [1] "Oct-16" "Oct-16" "Oct-16" "Nov-16" "Nov-16" "Nov-16" "Dic-16" "Dic-16"
## [9] "Dic-16"
##
## $DF
##   INDICE  FECHA                GRUPO
## 1 100.00 Oct-16 Privado_Registrado
## 2 100.00 Oct-16      Público
## 3 100.00 Oct-16 Privado_No_Registrado
## 4 101.80 Nov-16 Privado_Registrado
## 5 101.20 Nov-16      Público
## 6 100.73 Nov-16 Privado_No_Registrado
## 7 102.90 Dic-16 Privado_Registrado
## 8 102.40 Dic-16      Público
## 9 103.20 Dic-16 Privado_No_Registrado
##
## [[8]]
## [1] 100.00 100.00 100.00 101.80 101.20 100.73 102.90 102.40 103.20
##
## [[9]]
## [1] "Privado_Registrado" "Público" "Privado_No_Registrado"
## [4] "Privado_Registrado" "Público" "Privado_No_Registrado"
## [7] "Privado_Registrado" "Público" "Privado_No_Registrado"
```

Para acceder un elemento de una lista, podemos utilizar el operador `$`, que se puede usar a su vez de forma iterativa.

```
superlista$DF$FECHA[2]
```

```
## [1] Oct-16
## Levels: Dic-16 Nov-16 Oct-16
```

1.1.7 Ambientes de trabajo

Hay algunas cosas que tenemos que tener en cuenta respecto del orden del ambiente en el que trabajamos:

- Working Directory: Es el directorio de trabajo. Pueden ver el suyo con

`getwd()`, es *hacia donde apunta el código*, por ejemplo, si quieren leer un archivo, la ruta del archivo tiene que estar explicitada como el recorrido desde el Working Directory.

- **Environment:** Esto engloba tanto la información que tenemos cargada en *Data* y *Values*, como las librerías que tenemos cargadas mientras trabajamos.

Es importante que mantengamos bien delimitadas estas cosas entre diferentes trabajos, sino:

1. El directorio queda referido a un lugar específico en nuestra computadora.
 - Si se lo compartimos a otro **se rompe**
 - Si cambiamos de computadora **se rompe**
 - Si lo cambiamos de lugar **se rompe**
 - Si primero abrimos otro script **se rompe**
2. Tenemos mezclados resultados de diferentes trabajos:
 - Nunca sabemos si esa variable/tabla/lista se creo en ese script y no otro
 - Perdemos espacio de la memoria
 - No estamos seguros de que el script cargue todas las librerías que necesita

Rstudio tiene una herramienta muy útil de trabajo que son los **proyectos**. Estos permiten mantener un ambiente de trabajo delimitado por cada uno de nuestros trabajos. Es decir:

- El directorio de trabajo se refiere a donde esta ubicado el archivo .Rproj
- El Environment es específico de nuestro proyecto.

Un proyecto no es un sólo script, sino toda una carpeta de trabajo.

Para crearlo, vamos al logo de nuevo proyecto (Arriba a la derecha de la pantalla), y elegimos la carpeta de trabajo.

1.1.8 Tipos de archivos de R

- **Script:** Es un archivo de texto plano, donde podemos poner el código que utilizamos para preservarlo
- **Rnotebook:** También sirve para guardar el código, pero a diferencia de los scripts, se puede compilar, e intercalar código con resultados
- **Rproject:** Es un archivo que define la metadata del proyecto
- **RDS y Rdata:** Dos formatos de archivos propios de R para guardar datos.

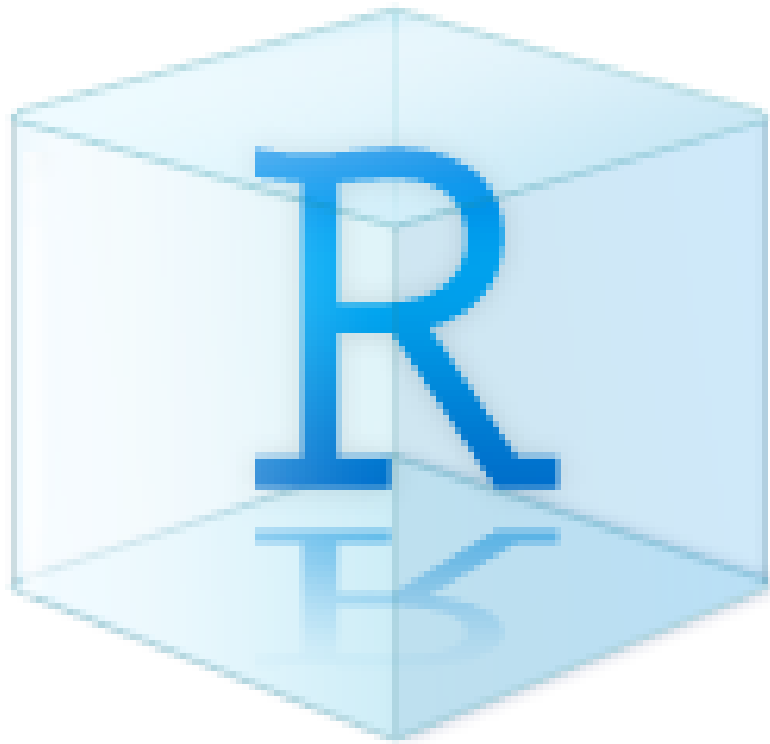


Figure 1.5: logo Rproject

1.2 Práctica Guiada

1.2.1 Instalación de paquetes complementarios al R Base

Hasta aquí hemos visto múltiples funciones que están contenidas dentro del lenguaje básico de R. Ahora bien, al tratarse de un software libre, los usuarios de R con más experiencia contribuyen sistemáticamente a expandir este lenguaje mediante la creación y actualización de **paquetes** complementarios. Lógicamente, los mismos no están incluidos en la instalación inicial del programa, pero podemos descargarlos e instalarlos al mismo tiempo con el siguiente comando:

```
install.packages("nombre_del_paquete")
```

Resulta recomendable **ejecutar este comando desde la consola** ya que sólo necesitaremos correrlo una vez en nuestra computadora. Al ejecutar el mismo, se descargarán de la pagina de CRAN los archivos correspondientes al paquete hacia el directorio en donde hayamos instalado el programa. Típicamente los archivos se encontrarán en **C:\Program Files\R\R-3.5.0\library**, siempre con la versión del programa correspondiente.

Una vez instalado el paquete, cada vez que abramos una nueva sesión de R y querramos utilizar el mismo debemos **cargarlo al ambiente de trabajo** mediante la siguiente función:

```
library(nombre_del_paquete)
```

Nótese que al cargar/activar el paquete no son necesarias las comillas.

1.2.2 Lectura y escritura de archivos

1.2.2.1 .csv y .txt

Hay **muchas** funciones para leer archivos de tipo *.txt* y *.csv*. La mayoría sólo cambia los parámetros que vienen por default.

Es importante tener en cuenta que una base de datos que proviene de archivos *.txt*, o *.csv* puede presentar diferencias en cuanto a los siguientes parámetros:

- encabezado
- delimitador (, , tab, ;)
- separador decimal

```
dataframe <- read.delim(file, header = TRUE, sep = "\t", quote = "\"", dec = ".", fill = TRUE, co
```

Ejemplo. Levantar la base de sueldos de funcionarios

En el parametro **file** tengo que especificar el nombre completo del archivo, incluyendo el directorio donde se encuentra. Lo más sencillo es abrir comillas, apretar Tab y se despliega el menú de las cosas que tenemos en el directorio de trabajo. Si queremos movernos hacia arriba, agregamos `../`

```
suealdos_funcionarios <- read.table(file = 'fuentes/sueldo_funcionarios_2019.csv', sep="
suealdos_funcionarios[1:10,]
```

```
##          cuil anio mes funcionario_apellido funcionario_nombre
## 1 20-17692128-6 2019 1 RODRIGUEZ LARRETA HORACIO ANTONIO
## 2 20-17735449-0 2019 1 SANTILLI DIEGO CESAR
## 3 27-24483014-0 2019 1 ACUÑA MARIA SOLEDAD
## 4 20-13872301-2 2019 1 ASTARLOA GABRIEL MARIA
## 5 20-25641207-2 2019 1 AVOGADRO ENRIQUE LUIS
## 6 27-13221055-7 2019 1 BOU PEREZ ANA MARIA
## 7 27-13092400-5 2019 1 FREDA MONICA BEATRIZ
## 8 20-17110752-1 2019 1 MACCHIAVELLI EDUARDO ALBERTO
## 9 20-22293873-3 2019 1 MIGUEL FELIPE OSCAR
## 10 20-14699669-9 2019 1 MOCCIA FRANCO
##          repartición asignacion_por_cargo_i
## 1 Jefe de Gobierno 197745.8
## 2 Vicejefatura de Gobierno 197745.8
## 3 Ministerio de Educación e Innovación 224516.6
## 4 Procuración General de la Ciudad de Buenos Aires 224516.6
## 5 Ministerio de Cultura 224516.6
## 6 Ministerio de Salud 224516.6
## 7 Sindicatura General de la Ciudad de Buenos Aires 224516.6
## 8 Ministerio de Ambiente y Espacio Público 224516.6
## 9 Jefatura de Gabinete de Ministros 224516.6
## 10 Ministerio de Desarrollo Urbano y Transporte 224516.6
## aguinaldo_ii total_salario_bruto_i_._ii observaciones
## 1 0 197745.8
## 2 0 197745.8
## 3 0 224516.6
## 4 0 224516.6
## 5 0 224516.6
## 6 0 224516.6
## 7 0 224516.6
## 8 0 224516.6
## 9 0 224516.6
## 10 0 224516.6
```

Como puede observarse aquí, la base cuenta con 94 registros y 10 variables. Al trabajar con bases de microdatos, resulta conveniente contar con algunos comandos para tener una mirada rápida de la base, antes de comenzar a realizar los procesamientos que deseemos.

Veamos algunos de ellos:

```
#view(suealdos_funcionarios)
names(suealdos_funcionarios)
```

```
## [1] "cuil" "anio"
## [3] "mes" "funcionario_apellido"
## [5] "funcionario_nombre" "repartición"
## [7] "asignacion_por_cargo_i" "aguinaldo_ii"
## [9] "total_salario_bruto_i._ii" "observaciones"
```

```
summary(sueldos_funcionarios)
```

```
##          cuil          anio          mes  funcionario_apellido
## 20-13872301-2: 3  Min.   :2019  Min.   :1.00  ACUÑA       : 3
## 20-14699669-9: 3  1st Qu.:2019  1st Qu.:2.00  ASTARLOA     : 3
## 20-16891528-5: 3  Median :2019  Median :3.00  AVELLANEDA  : 3
## 20-16891539-0: 3  Mean    :2019  Mean    :3.34  AVOGADRO    : 3
## 20-17110752-1: 3  3rd Qu.:2019  3rd Qu.:5.00  BENEGAS     : 3
## 20-17692128-6: 3  Max.    :2019  Max.    :6.00  BOU PEREZ   : 3
## (Other)       :76  (Other)   :76
##          funcionario_nombre
## ANA MARIA      : 3
## BRUNO GUIDO    : 3
## CHRISTIAN      : 3
## DIEGO CESAR    : 3
## DIEGO HERNAN   : 3
## EDUARDO ALBERTO: 3
## (Other)        :76
##                                     repartición
## Consejo de los Derechos de Niñas, Niños y Adoles - Presidencia: 3
## Ente de Turismo Ley N° 2627                                     : 3
## Jefatura de Gabinete de Ministros                               : 3
## Jefe de Gobierno                                                : 3
## Ministerio de Ambiente y Espacio Público                         : 3
## Ministerio de Cultura                                           : 3
## (Other)                                                         :76
## asignacion_por_cargo_i  aguinaldo_ii  total_salario_bruto_i._ii
## Min.   :197746          Min.   :    0  Min.   :197746
## 1st Qu.:217520          1st Qu.:    0  1st Qu.:217805
## Median :226866          Median :    0  Median :226866
## Mean   :224718          Mean   : 14843  Mean   :239560
## 3rd Qu.:231168          3rd Qu.:    0  3rd Qu.:248033
## Max.   :249662          Max.   :113433  Max.   :340300
##
##          observaciones
##                   :93
## baja 28/2/2019: 1
##
##
##
```

```
##
##
```

```
head(sueldos_funcionarios)[,1:5]
```

```
##           cuil anio mes funcionario_apellido funcionario_nombre
## 1 20-17692128-6 2019   1   RODRIGUEZ LARRETA   HORACIO ANTONIO
## 2 20-17735449-0 2019   1           SANTILLI     DIEGO CESAR
## 3 27-24483014-0 2019   1           ACUÑA       MARIA SOLEDAD
## 4 20-13872301-2 2019   1         ASTARLOA     GABRIEL MARIA
## 5 20-25641207-2 2019   1         AVOGADRO     ENRIQUE LUIS
## 6 27-13221055-7 2019   1          BOU PEREZ     ANA MARIA
```

1.2.2.2 Excel

Para leer y escribir archivos excel podemos utilizar los comandos que vienen con la librería *openxlsx*

```
# install.packages("openxlsx") # por única vez
library(openxlsx) #activamos la librería

# creamos una tabla cualquiera de prueba
x <- 1:10
y <- 11:20
tabla_de_R <- data.frame(x,y)

# escribimos el archivo
write.xlsx(x = tabla_de_R, file = "resultados/archivo.xlsx", row.names = FALSE)
# Donde lo guardó? Hay un directorio por default en caso de que no hayamos definido al.

# getwd()

# Si queremos exportar multiples dataframes a un Excel, debemos armar previamente una
Lista_a_exportar <- list("sueldos funcionarios" = sueldos_funcionarios,
                        "Tabla Numeros" = tabla_de_R)

write.xlsx(x = Lista_a_exportar, file = "resultados/archivo_2_hojas.xlsx", row.names =

# leemos el archivo especificando la ruta (o el directorio por default) y el nombre de
Indices_Salario <- read.xlsx(xlsxFile = "resultados/archivo_2_hojas.xlsx", sheet = "sueldos funcionarios")

# alternativamente podemos especificar el número de orden de la hoja que deseamos leer
Indices_Salario <- read.xlsx(xlsxFile = "resultados/archivo_2_hojas.xlsx", sheet = 1)
Indices_Salario[1:10,]
```

```
##           cuil anio mes funcionario_apellido funcionario_nombre
## 1 20-17692128-6 2019   1   RODRIGUEZ LARRETA   HORACIO ANTONIO
```

## 2	20-17735449-0	2019	1	SANTILLI	DIEGO CESAR
## 3	27-24483014-0	2019	1	ACUÑA	MARIA SOLEDAD
## 4	20-13872301-2	2019	1	ASTARLOA	GABRIEL MARIA
## 5	20-25641207-2	2019	1	AVOGADRO	ENRIQUE LUIS
## 6	27-13221055-7	2019	1	BOU PEREZ	ANA MARIA
## 7	27-13092400-5	2019	1	FREDA	MONICA BEATRIZ
## 8	20-17110752-1	2019	1	MACCHIAVELLI	EDUARDO ALBERTO
## 9	20-22293873-3	2019	1	MIGUEL	FELIPE OSCAR
## 10	20-14699669-9	2019	1	MOCCIA	FRANCO
##	repartición asignacion_por_cargo_i				
## 1	Jefe de Gobierno				197745.8
## 2	Vicejefatura de Gobierno				197745.8
## 3	Ministerio de Educación e Innovación				224516.6
## 4	Procuración General de la Ciudad de Buenos Aires				224516.6
## 5	Ministerio de Cultura				224516.6
## 6	Ministerio de Salud				224516.6
## 7	Sindicatura General de la Ciudad de Buenos Aires				224516.6
## 8	Ministerio de Ambiente y Espacio Público				224516.6
## 9	Jefatura de Gabinete de Ministros				224516.6
## 10	Ministerio de Desarrollo Urbano y Transporte				224516.6
##	aguinaldo_ii	total_salario_bruto_i_.	ii	observaciones	
## 1	0			197745.8	
## 2	0			197745.8	
## 3	0			224516.6	
## 4	0			224516.6	
## 5	0			224516.6	
## 6	0			224516.6	
## 7	0			224516.6	
## 8	0			224516.6	
## 9	0			224516.6	
## 10	0			224516.6	

Chapter 2

Tidyverse

- Limpieza de Base de datos: Renombrar y recodificar variables, tratamiento de valores faltantes (missing values/ NA's)
- Seleccionar variables, ordenar y agrupar la base de datos para realizar cálculos
- Creación de nuevas variables
- Aplicar filtros sobre la base de datos
- Construir medidas de resumen de la información
- Tratamiento de variables numéricas (edad, ingresos, horas de trabajo, cantidad de hijos / componentes del hogar, entre otras).

2.1 Explicación

A lo largo de esta clase, trabajaremos con el paquete **tidyverse**. El mismo agrupa una serie de paquetes que tienen una misma lógica en su diseño y por ende funcionan en armonía.

Entre ellos, usaremos principalmente **dplyr** y **tidyr** para realizar transformaciones sobre nuestro set de datos. En una futura clase utilizaremos **ggplot** para realizar gráficos.

A continuación cargamos la librería a nuestro ambiente. Para ello debe estar previamente instalada en nuestra pc.

```
library(tidyverse)
```

Para mostrar el funcionamiento básico de tidyverse utilizaremos a modo de ejemplo datos del Informe de Mercado de Trabajo del INDEC.

```
INDICADOR <- c("Tasa de Actividad", "Tasa de Empleo", "Tasa de Desocupación",  
               "Tasa de Actividad", "Tasa de Empleo", "Tasa de Desocupación",  
               "Tasa de Actividad", "Tasa de Empleo", "Tasa de Desocupación")
```

```
FECHA <- c("2018.3T", "2018.3T", "2018.3T",
           "2018.4T", "2018.4T", "2018.4T",
           "2019.1T", "2019.1T", "2019.1T")

TASA <- c(46.7, 42.5, 9,
          46.5, 42.2, 9.1,
          47, 42.3, 10.1)

Datos <- data.frame(INDICADOR, FECHA, TASA)
Datos
```

```
##           INDICADOR  FECHA TASA
## 1 Tasa de Actividad 2018.3T 46.7
## 2 Tasa de Empleo   2018.3T 42.5
## 3 Tasa de Desocupación 2018.3T 9.0
## 4 Tasa de Actividad 2018.4T 46.5
## 5 Tasa de Empleo   2018.4T 42.2
## 6 Tasa de Desocupación 2018.4T 9.1
## 7 Tasa de Actividad 2019.1T 47.0
## 8 Tasa de Empleo   2019.1T 42.3
## 9 Tasa de Desocupación 2019.1T 10.1
```

2.1.1 Dplyr

El caracter principal para utilizar este paquete es `%>%`, *pipe* (de tubería).

Los `%>%` toman el set de datos a su izquierda, y los transforman mediante los comandos a su derecha, en los cuales los elementos de la izquierda están implícitos. En otros términos:

$f(x, y)$ es equivalente a $x \%>\% f(., y)$

Veamos las principales funciones que pueden utilizarse con la lógica de este paquete:

2.1.1.1 glimpse

Permite ver la estructura de la tabla. Nos muestra:

- número de filas
- número de columnas
- nombre de las columnas
- tipo de dato de cada columna
- las primeras observaciones de la tabla

```
glimpse(Datos)
```

```

Observations: 9
Variables: 3
$ INDICADOR <fct> Tasa de Actividad, Tasa de Empleo, Tasa de Desocupación, Tasa de Actividad, Tasa de Empleo, Tasa de Desocupación, Tasa de Actividad, Tasa de E...
$ FECHA <fct> 2018.3T, 2018.3T, 2018.3T, 2018.4T, 2018.4T, 2018.4T, 2019.1T, 2019.1T, 2019.1T
$ TASA <dbl> 46.7, 42.5, 9.0, 46.5, 42.2, 9.1, 47.0, 42.3, 10.1

```

2.1.1.2 filter

Permite filtrar la tabla de acuerdo al cumplimiento de condiciones lógicas.

Datos %>%

```
filter(TASA > 10 , INDICADOR == "Tasa de Desocupación")
```

```
##                INDICADOR  FECHA TASA
## 1 Tasa de Desocupación 2019.1T 10.1
```

Nótese que en este caso al separar con una `,` las condiciones se exige el cumplimiento de ambas. En caso de desear que se cumpla alguna de las condiciones debe utilizarse el caracter `|`.

Datos %>%

```
filter(TASA > 10 | INDICADOR == "Tasa de Desocupación")
```

```
##                INDICADOR  FECHA TASA
## 1 Tasa de Actividad 2018.3T 46.7
## 2 Tasa de Empleo 2018.3T 42.5
## 3 Tasa de Desocupación 2018.3T 9.0
## 4 Tasa de Actividad 2018.4T 46.5
## 5 Tasa de Empleo 2018.4T 42.2
## 6 Tasa de Desocupación 2018.4T 9.1
## 7 Tasa de Actividad 2019.1T 47.0
## 8 Tasa de Empleo 2019.1T 42.3
## 9 Tasa de Desocupación 2019.1T 10.1
```

2.1.1.3 rename

Permite renombrar una columna de la tabla. Funciona de la siguiente manera:

```
Data %>% rename(nuevo_nombre = viejo_nombre)
```

Datos %>%

```
rename(Periodo = FECHA)
```

```
##                INDICADOR Periodo TASA
## 1 Tasa de Actividad 2018.3T 46.7
## 2 Tasa de Empleo 2018.3T 42.5
## 3 Tasa de Desocupación 2018.3T 9.0
## 4 Tasa de Actividad 2018.4T 46.5
## 5 Tasa de Empleo 2018.4T 42.2
## 6 Tasa de Desocupación 2018.4T 9.1
## 7 Tasa de Actividad 2019.1T 47.0
```

```
## 8      Tasa de Empleo 2019.1T 42.3
## 9 Tasa de Desocupación 2019.1T 10.1
```

Nótese que, a diferencia del ejemplo de la función **filter** donde utilizábamos `==` para comprobar una condición lógica, en este caso se utiliza sólo un `=` ya que lo estamos haciendo es *asignar* un nombre.

2.1.1.4 mutate

Permite agregar una variable a la tabla (especificando el nombre que tomará ésta), que puede ser el resultado de operaciones sobre otras variables de la misma tabla.

En caso de especificar el nombre de una columna existente, el resultado de la operación realizada “sobre-escribirá” la información de la columna con dicho nombre.

```
Datos <- Datos %>%
  mutate(PROPORCION = TASA / 100)
```

Datos

##	INDICADOR	FECHA	TASA	PROPORCION
## 1	Tasa de Actividad	2018.3T	46.7	0.467
## 2	Tasa de Empleo	2018.3T	42.5	0.425
## 3	Tasa de Desocupación	2018.3T	9.0	0.090
## 4	Tasa de Actividad	2018.4T	46.5	0.465
## 5	Tasa de Empleo	2018.4T	42.2	0.422
## 6	Tasa de Desocupación	2018.4T	9.1	0.091
## 7	Tasa de Actividad	2019.1T	47.0	0.470
## 8	Tasa de Empleo	2019.1T	42.3	0.423
## 9	Tasa de Desocupación	2019.1T	10.1	0.101

2.1.1.5 case_when

Permite definir una variable, de forma tal que tome un valor particular para cada condición establecida. En caso de no cumplir con ninguna de las condiciones establecidas, la variable tomará valor **NA**.

La sintaxis de la función es:

```
case_when(condicion lógica1 ~ valor asignado1)
```

```
Datos <- Datos %>%
  mutate(CODIGO = case_when(INDICADOR == "Tasa de Actividad" ~ "ACT",
                             INDICADOR == "Tasa de Empleo" ~ "EMP",
                             INDICADOR == "Tasa de Desocupación" ~ "DES"))
```

Datos

##	INDICADOR	FECHA	TASA	PROPORCION	CODIGO
----	-----------	-------	------	------------	--------

```
## 1    Tasa de Actividad 2018.3T 46.7      0.467    ACT
## 2      Tasa de Empleo 2018.3T 42.5      0.425    EMP
## 3 Tasa de Desocupación 2018.3T  9.0      0.090    DES
## 4      Tasa de Actividad 2018.4T 46.5      0.465    ACT
## 5      Tasa de Empleo 2018.4T 42.2      0.422    EMP
## 6 Tasa de Desocupación 2018.4T  9.1      0.091    DES
## 7      Tasa de Actividad 2019.1T 47.0      0.470    ACT
## 8      Tasa de Empleo 2019.1T 42.3      0.423    EMP
## 9 Tasa de Desocupación 2019.1T 10.1      0.101    DES
```

Si queremos asignar un valor a todo lo que no cumple ninguna de las condiciones anteriores, podemos poner `TRUE ~ valor`

2.1.1.6 select

Permite especificar la serie de columnas que se desea conservar de un `DataFrame`. También pueden especificarse las columnas que se desean descartar (agregándoles un `-` adelante). Muy útil para agilizar el trabajo en bases de datos de gran tamaño.

```
Datos2 <- Datos %>%
  select(CODIGO, FECHA, PROPORCION)
Datos2
```

```
##  CODIGO  FECHA  PROPORCION
## 1    ACT 2018.3T    0.467
## 2    EMP 2018.3T    0.425
## 3    DES 2018.3T    0.090
## 4    ACT 2018.4T    0.465
## 5    EMP 2018.4T    0.422
## 6    DES 2018.4T    0.091
## 7    ACT 2019.1T    0.470
## 8    EMP 2019.1T    0.423
## 9    DES 2019.1T    0.101
```

```
Datos <- Datos %>%
  select(-c(PROPORCION, CODIGO))
Datos
```

```
##          INDICADOR  FECHA TASA
## 1    Tasa de Actividad 2018.3T 46.7
## 2      Tasa de Empleo 2018.3T 42.5
## 3 Tasa de Desocupación 2018.3T  9.0
## 4    Tasa de Actividad 2018.4T 46.5
## 5      Tasa de Empleo 2018.4T 42.2
## 6 Tasa de Desocupación 2018.4T  9.1
## 7    Tasa de Actividad 2019.1T 47.0
## 8      Tasa de Empleo 2019.1T 42.3
```

```
## 9 Tasa de Desocupación 2019.1T 10.1
```

2.1.1.7 arrange

Permite ordenar la tabla según los valores de determinada/s variable/s. Es útil cuando luego deben hacerse otras operaciones que requieran del ordenamiento de la tabla, o para mostrar resultados de forma ordenada.

```
Datos <- Datos %>%
  arrange(INDICADOR, FECHA)
```

```
Datos
```

```
##           INDICADOR  FECHA TASA
## 1  Tasa de Actividad 2018.3T 46.7
## 2  Tasa de Actividad 2018.4T 46.5
## 3  Tasa de Actividad 2019.1T 47.0
## 4 Tasa de Desocupación 2018.3T  9.0
## 5 Tasa de Desocupación 2018.4T  9.1
## 6 Tasa de Desocupación 2019.1T 10.1
## 7      Tasa de Empleo 2018.3T 42.5
## 8      Tasa de Empleo 2018.4T 42.2
## 9      Tasa de Empleo 2019.1T 42.3
```

2.1.1.8 summarise

Crea una nueva tabla que resuma la información original. Para ello, definimos las variables de resumen y las formas de agregación.

```
Datos %>%
  filter(INDICADOR == "Tasa de Desocupación") %>%
  summarise(INDICE_MAX = max(TASA),
            INDICE_MIN = min(TASA),
            INDICE_PROM = mean(TASA))
```

```
##  INDICE_MAX INDICE_MIN INDICE_PROM
## 1         10.1         9         9.4
```

2.1.1.9 group_by

Esta función permite realizar operaciones de forma agrupada. Lo que hace la función es “separar” a la tabla según los valores de la variable indicada y realizar las operaciones que se especifican a continuación, de manera independiente para cada una de las “subtablas”. En nuestro ejemplo, podría ser útil para calcular el promedio de las tasas por *INDICADOR*.

```
Datos %>%
  group_by(INDICADOR) %>%
  summarise(INDICE_PROM = mean(TASA))
```

```
## # A tibble: 3 x 2
##   INDICADOR      INDICE_PROM
##   <fct>          <dbl>
## 1 Tasa de Actividad      46.7
## 2 Tasa de Desocupación    9.4
## 3 Tasa de Empleo        42.3
```

2.1.2 Joins

Otra implementación muy importante del paquete dplyr son las funciones para unir tablas (joins).

2.1.2.1 left_join

Veamos un ejemplo de la función `left_join` (una de las más utilizadas en la práctica).

Para ello crearemos previamente un Dataframe que contenga las cantidades de población total y población económicamente activa para cada uno de los períodos del Dataframe *Datos*.

```
Poblaciones <- data.frame(FECHA = c("2018.3T", "2018.4T", "2019.1T"),
                          POBLACION_miles = c(27842, 27914, 28261),
                          PEA_miles = c(12990, 12979, 13285))
```

Poblaciones

```
##   FECHA POBLACION_miles PEA_miles
## 1 2018.3T      27842      12990
## 2 2018.4T      27914      12979
## 3 2019.1T      28261      13285
```

Unimos nuestras dos tablas. La siguiente forma de realizarlo es equivalente a:

```
Datos_join <- left_join(Datos, Poblaciones, by = "FECHA")
```

```
Datos_join <- Datos %>%
  left_join(Poblaciones, by = "FECHA")
```

Datos_join

```
##           INDICADOR  FECHA TASA POBLACION_miles PEA_miles
## 1   Tasa de Actividad 2018.3T  46.7      27842      12990
## 2   Tasa de Actividad 2018.4T  46.5      27914      12979
## 3   Tasa de Actividad 2019.1T  47.0      28261      13285
## 4 Tasa de Desocupación 2018.3T   9.0      27842      12990
## 5 Tasa de Desocupación 2018.4T   9.1      27914      12979
## 6 Tasa de Desocupación 2019.1T  10.1      28261      13285
## 7      Tasa de Empleo 2018.3T  42.5      27842      12990
## 8      Tasa de Empleo 2018.4T  42.2      27914      12979
```

dplyr *joins*

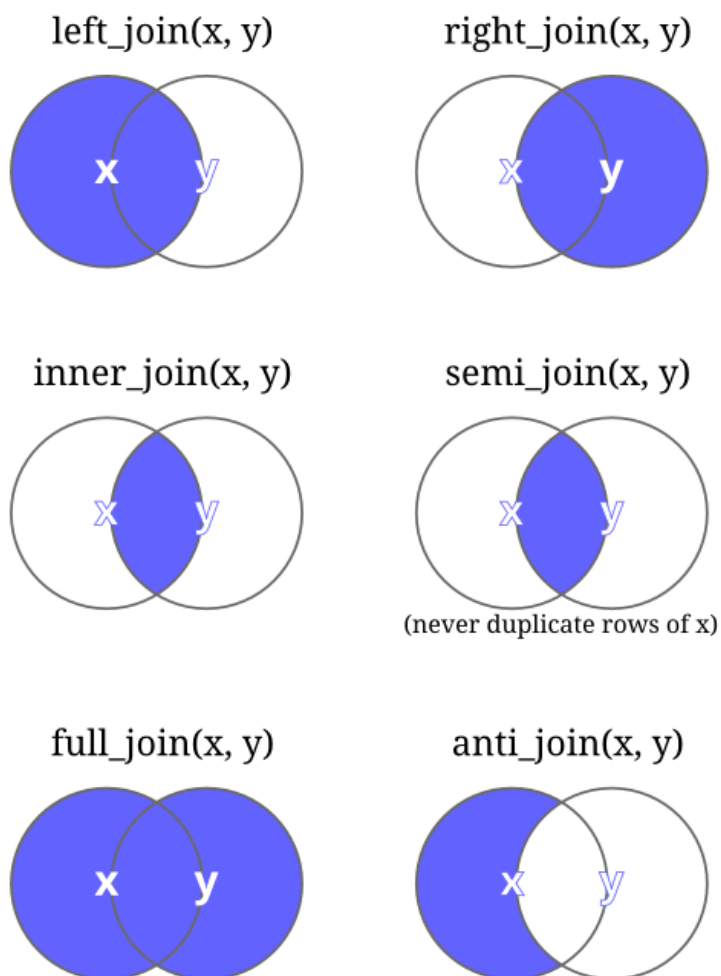


Figure 2.1: fuente: http://rstudio-pubs-static.s3.amazonaws.com/227171_618ebdce0b9d44f3af65700e833593db.html

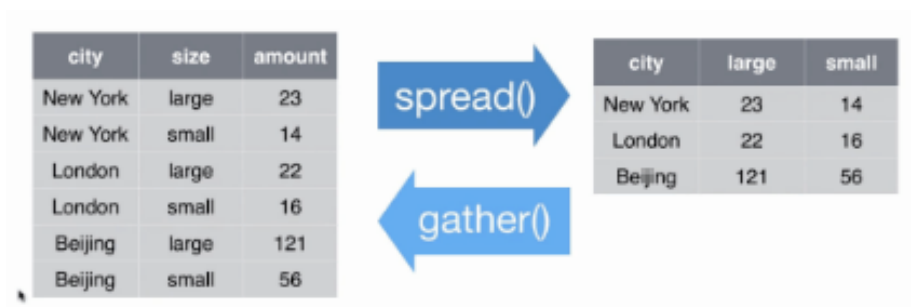


Figure 2.2: fuente: <http://www.gis-blog.com/data-management-with-r-tidyr-part-1/>

```
## 9      Tasa de Empleo 2019.1T 42.3      28261      13285
```

Finalmente, podemos calcular la cantidad de personas desocupadas en cada uno de los períodos con los que contamos.

```
Datos_join %>%
  filter(INDICADOR == "Tasa de Desocupación") %>%
  group_by(FECHA) %>%
  summarise(DESOCUP_miles = round(TASA/100 * PEA_miles, 0))
```

```
## # A tibble: 3 x 2
##   FECHA   DESOCUP_miles
##   <fct>         <dbl>
## 1 2018.3T         1169
## 2 2018.4T         1181
## 3 2019.1T         1342
```

2.1.3 Tidyr

El paquete tidyr está pensado para facilitar el emprolijamiento de los datos.

Gather es una función que nos permite pasar los datos de forma horizontal a una forma vertical.

spread es una función que nos permite pasar los datos de forma vertical a una forma horizontal.

```
# Utilizamos un conjunto de datos que viene con la librería datasets
library(datasets)
```

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1         5.1         3.5         1.4         0.2   setosa
## 2         4.9         3.0         1.4         0.2   setosa
```

```
## 3      4.7      3.2      1.3      0.2 setosa
## 4      4.6      3.1      1.5      0.2 setosa
## 5      5.0      3.6      1.4      0.2 setosa
## 6      5.4      3.9      1.7      0.4 setosa
```

```
iris <- iris %>%
  mutate(id = 1:nrow()) %>% # le agrego un ID
  select(id, everything())   # lo acomodo para que el id este primero.

head(iris)
```

```
##   id Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1  1          5.1          3.5          1.4          0.2 setosa
## 2  2          4.9          3.0          1.4          0.2 setosa
## 3  3          4.7          3.2          1.3          0.2 setosa
## 4  4          4.6          3.1          1.5          0.2 setosa
## 5  5          5.0          3.6          1.4          0.2 setosa
## 6  6          5.4          3.9          1.7          0.4 setosa
```

2.1.3.1 Gather y Spread

```
iris_vertical <- iris %>% gather(., # el . llama a lo que esta atras del %>%
                                key  = Variables,
                                value = Valores,
                                2:5) #le indico qué columnas juntar

head(iris_vertical)
```

```
##   id Species    Variables Valores
## 1  1 setosa Sepal.Length      5.1
## 2  2 setosa Sepal.Length      4.9
## 3  3 setosa Sepal.Length      4.7
## 4  4 setosa Sepal.Length      4.6
## 5  5 setosa Sepal.Length      5.0
## 6  6 setosa Sepal.Length      5.4
```

Podemos deshacer el **gather** con un **Spread**

```
iris_horizontal <- iris_vertical %>%
  spread(. ,
         key  = Variables, # la llave es la variable que va a dar los nombres de columnas
         value = Valores) # los valores con que se llenan las celdas

head(iris_horizontal)
```

```
##   id Species Petal.Length Petal.Width Sepal.Length Sepal.Width
## 1  1 setosa          1.4          0.2          5.1          3.5
## 2  2 setosa          1.4          0.2          4.9          3.0
## 3  3 setosa          1.3          0.2          4.7          3.2
```

```
## 4 4 setosa      1.5      0.2      4.6      3.1
## 5 5 setosa      1.4      0.2      5.0      3.6
## 6 6 setosa      1.7      0.4      5.4      3.9
```

2.1.4 Lubridate

El paquete lubridate está pensado para trabajar con los datos tipo fecha (date) o fecha-hora (datetime) para cambiarles el formato, realizar operaciones y extraer información

```
library(lubridate)
```

2.1.4.1 Cambio de formato

Existe una gran cantidad de funciones para realizar esto. La idea general es poder llevar los objetos datetime a un formato común compuesto de los elementos: año, mes, día, hora, minuto y segundo (también se puede setear el huso horario)

```
fecha <- "04/12/92 17:35:16"
fecha
```

```
## [1] "04/12/92 17:35:16"
```

Con la función `dmy_hms` podemos convertir este string a una fecha: estamos indicando que el formato de la fecha es día(d), mes(m), año(y), hora(h), minuto(m) y segundo(s).

```
fecha <- dmy_hms(fecha)
fecha
```

```
## [1] "1992-12-04 17:35:16 UTC"
```

Muchas funciones de lubridate operan con esta misma lógica.

Otra función para realizar un cambio de formato es `parse_date_time`. Permite construir objetos datetime a partir de datos más complejos, como por ejemplo cuando aparece el nombre del mes y el año.

En el parámetro `x` pasamos el dato de la fecha y en el parámetro `orders` especificamos el orden en el cual se encuentra la información de la fecha.

```
fecha2 <- "Dec-92"
fecha2 <- parse_date_time(fecha2, orders = 'my')
fecha2
```

```
## [1] "1992-12-01 UTC"
```

2.1.4.2 Extracción de información

Existen muchas funciones muy sencillas para extraer información de un objeto `datetime`. Algunas son:

```
year(fecha) # Obtener el año
```

```
## [1] 1992
```

```
month(fecha) # Obtener el mes
```

```
## [1] 12
```

```
day(fecha) # Obtener el día
```

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

```
wday(fecha, label = TRUE) # Obtener el nombre del día
```

```
## [1] vie
```

```
## Levels: dom < lun < mar < mié < jue < vie < sáb
```

```
hour(fecha) # Obtener la hora
```

```
## [1] 17
```

2.1.4.3 Operaciones

Podemos sumar o restarle cualquier período de tiempo a un objeto `datetime`

```
# Sumo dos días  
fecha + days(2)
```

```
## [1] "1992-12-06 17:35:16 UTC"
```

```
# Resto 1 semana y dos horas  
fecha - (weeks(1) + hours(2))
```

```
## [1] "1992-11-27 15:35:16 UTC"
```

2.2 Práctica Guiada

En esta ocasión utilizaremos los datos de la librería `gapminder` para utilizar todo lo que aprendimos sobre el `tidyverse`.

```
library(tidyverse)  
library(gapminder)
```

```
glimpse(gapminder)
```

```
## Observations: 1,704
```

```
## Variables: 6
```

```
## $ country <fct> Afghanistan, Afghanistan, Afghanistan, Afghanistan, ...
## $ continent <fct> Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia...
## $ year <int> 1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992...
## $ lifeExp <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, 38.438, 39.8...
## $ pop <int> 8425333, 9240934, 10267083, 11537966, 13079460, 1488...
## $ gdpPercap <dbl> 779.4453, 820.8530, 853.1007, 836.1971, 739.9811, 78...
```

2.2.1 Ejemplo 1

Calcular el promedio, el máximo y el mínimo de la esperanza de vida de cada continente en el año 2007. Presentar los datos ordenados según la esperanza de vida promedio.

Necesitamos filtrar los datos tal que sólo queden aquellos correspondientes a 2007. Luego, agrupamos los casos de acuerdo a su *continente*, y calculamos los indicadores agregados solicitados. Luego, ordenamos los resultados.

```
ejercicio1 <- gapminder %>%
  filter(year == 2007) %>%
  group_by(continent) %>%
  summarise(esp_vida_prom = mean(lifeExp),
            esp_vida_max = max(lifeExp),
            esp_vida_min = min(lifeExp)) %>%
  arrange(esp_vida_prom)

ejercicio1
```

```
## # A tibble: 5 x 4
##   continent esp_vida_prom esp_vida_max esp_vida_min
##   <fct>         <dbl>         <dbl>         <dbl>
## 1 Africa          54.8           76.4           39.6
## 2 Asia            70.7           82.6           43.8
## 3 Americas       73.6           80.7           60.9
## 4 Europe         77.6           81.8           71.8
## 5 Oceania        80.7           81.2           80.2
```

2.2.2 Ejemplo 2

Construir una nueva variable en el dataset que contenga una estimación del PBI. Estimar la mediana del PBI, y construir otra variable que tome valor “ALTO” cuando el PBI supera ese valor, y “BAJO” cuando no.

Calculamos el PBI como el producto entre la población y el PBI per cápita para cada uno de los países y años. A continuación, guardamos el cálculo de la mediana del PBI en un valor llamado *mediana_GDP*. Por último, utilizamos la función `case_when` para poder construir la variable de nivel de PBI de acuerdo

a la condición lógica solicitada. Nótese que el dataframe `ejercicio2` ha sido re-escrito.

```
ejercicio2 <- gapminder %>%
  mutate(GDP = pop * gdpPercap)

mediana_GDP <- median(ejercicio2$GDP)

ejercicio2 <- ejercicio2 %>%
  mutate(GDP_level = case_when(GDP > mediana_GDP ~ "ALTO",
                               GDP < mediana_GDP ~ "BAJO"))

head(ejercicio2)
```

```
## # A tibble: 6 x 8
##   country    continent year lifeExp      pop gdpPercap      GDP GDP_level
##   <fct>      <fct>    <int>   <dbl>   <int>    <dbl>    <dbl> <chr>
## 1 Afghanist~ Asia      1952   28.8  8.43e6    779.   6.57e 9 BAJO
## 2 Afghanist~ Asia      1957   30.3  9.24e6    821.   7.59e 9 BAJO
## 3 Afghanist~ Asia      1962   32.0  1.03e7    853.   8.76e 9 BAJO
## 4 Afghanist~ Asia      1967   34.0  1.15e7    836.   9.65e 9 BAJO
## 5 Afghanist~ Asia      1972   36.1  1.31e7    740.   9.68e 9 BAJO
## 6 Afghanist~ Asia      1977   38.4  1.49e7    786.  1.17e10 BAJO
```

2.2.3 Ejemplo 3

Crear una copia de la base donde sólo se conserven las variables *country*, *year* y *lifeExp*, pero con los nombres *pais*, *anio* y *espVida*.

Utilizamos `select()` para quedarnos con las columnas solicitadas, y `rename()` para cambiar sus nombres.

```
ejercicio3 <- gapminder %>%
  select(country, year, lifeExp) %>%
  rename(pais = country,
         anio = year,
         espVida = lifeExp)

head(ejercicio3)
```

```
## # A tibble: 6 x 3
##   pais      anio espVida
##   <fct>    <int>   <dbl>
## 1 Afghanistan 1952   28.8
## 2 Afghanistan 1957   30.3
## 3 Afghanistan 1962   32.0
## 4 Afghanistan 1967   34.0
## 5 Afghanistan 1972   36.1
```

```
## 6 Afghanistan 1977 38.4
```

2.2.4 Ejemplo 4

Crear una copia de la base donde sólo se conserven las variables *country*, *year* y *gdpPercap*, pero con los nombres *pais*, *anio* y *pbiPercap*.

```
ejercicio4 <- gapminder %>%
  select(country, year, gdpPercap) %>%
  rename(pais = country,
         anio = year,
         pbiPercap = gdpPercap)

head(ejercicio4)
```

```
## # A tibble: 6 x 3
##   pais      anio pbiPercap
##   <fct>    <int>    <dbl>
## 1 Afghanistan 1952      779.
## 2 Afghanistan 1957      821.
## 3 Afghanistan 1962      853.
## 4 Afghanistan 1967      836.
## 5 Afghanistan 1972      740.
## 6 Afghanistan 1977      786.
```

2.2.5 Ejemplo 5

Crear una nueva tabla que contenga los datos de las tablas *ejercicio3* y *ejercicio4*. Deben unirse de acuerdo al *pais* y al *anio*.

Podemos utilizar la función `left_join()`.

```
ejercicio5 <- left_join(ejercicio3, ejercicio4, by = c("pais", "anio"))

head(ejercicio5)
```

```
## # A tibble: 6 x 4
##   pais      anio espVida pbiPercap
##   <fct>    <int>    <dbl>    <dbl>
## 1 Afghanistan 1952     28.8      779.
## 2 Afghanistan 1957     30.3      821.
## 3 Afghanistan 1962     32.0      853.
## 4 Afghanistan 1967     34.0      836.
## 5 Afghanistan 1972     36.1      740.
## 6 Afghanistan 1977     38.4      786.
```

2.2.6 Ejemplo 6

Presentar los datos de la tabla `ejercicio1` de forma tal que `esp_vida_prom`, `esp_vida_max` y `esp_vida_min` sean valores de una variable llamada *indicador*, y los valores se encuentren en la variable *valor*.

Utilizamos `gather()`, porque queremos transformar los datos de un formato “horizontal” a uno “vertical”.

```
ejercicio6 <- ejercicio1 %>%  
  gather(., key = indicador, value = valor, 2:4)  
  
head(ejercicio6)
```

```
## # A tibble: 6 x 3  
##   continent indicador      valor  
##   <fct>      <chr>      <dbl>  
## 1 Africa    esp_vida_prom  54.8  
## 2 Asia      esp_vida_prom  70.7  
## 3 Americas  esp_vida_prom  73.6  
## 4 Europe    esp_vida_prom  77.6  
## 5 Oceania   esp_vida_prom  80.7  
## 6 Africa    esp_vida_max   76.4
```


Chapter 3

Programacion Funcional

El objetivo de esta clase es introducir a los alumnos en el uso de la programación funcional. Es decir, en la utilización de funciones y el uso de controles de flujo de la información para la organización de su código.

- Estructuras de código condicionales
- Loops
- Creación de funciones a medida del usuario
- Librería purrr para programación funcional

3.1 Explicación

```
library(tidyverse)
```

3.1.1 Loops

Un **loop** es una estructura de código que nos permite aplicar iterativamente un mismo conjunto de comandos, variando el valor de una variable. Por ejemplo:

```
for(i in 1:10){  
  print(i^2)  
}
```

```
## [1] 1  
## [1] 4  
## [1] 9  
## [1] 16  
## [1] 25  
## [1] 36  
## [1] 49
```

```
## [1] 64
## [1] 81
## [1] 100
```

Esto se lee como : “Recorre cada uno de los valores (i) del vector numérico 1 a 10, y para cada uno de ellos imprimí el cuadrado (i^2)”.

Uno puede especificar la palabra que desee que tomé cada uno de los valores que debe tomar. En el ejemplo anterior fue **i**, pero bien podría ser la “**Valores**”

```
for(Valores in 1:10){
  print(Valores^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
## [1] 64
## [1] 81
## [1] 100
```

Un loop puede iterar sobre cualquier tipo de vector, independientemente de lo que contenga.

Los loops son una estructura básica que existen en cualquier lenguaje de programación. En R no recomendamos abusar de ellos porque hacen que el código sea más lento.

3.1.2 Estructuras Condicionales

Las **estructuras condicionales** nos permiten ejecutar una porción de código en caso de que cumplan una condición lógica

3.1.2.1 if

Su funcionamiento es el siguiente:

```
if(condicion){codigo a ejecutar si se cumple la condición}
```

```
if( 2+2 == 4){
  print("Menos Mal")
}
```

```
## [1] "Menos Mal"
```

```
if( 2+2 == 148.24){
  print("R, tenemos un problema")
}
```

3.1.2.2 ifelse

La función `if_else()` sirve para crear o modificar dicotómicamente un objeto/variable/vector a partir del cumplimiento de una o más condiciones lógicas.

Su funcionamiento es el siguiente:

`if_else(condición, función a aplicar si se cumple la condición, función a aplicar si no se cumple la condición)`

```
if_else(2+2==4, true = "Joya", false = "Error")
```

```
## [1] "Joya"
```

3.1.3 Funciones

La creación de **funciones** propias nos permite automatizar todas aquellas partes del código que se repiten mucho. Una vez diseñadas, funcionan igual que cualquier comando.

Por ejemplo, podemos definir la suma de dos elementos como

```
suma <- function(valor1, valor2) {
  valor1+valor2
}
```

```
suma(5,6)
```

```
## [1] 11
```

Obviamente las funciones no son sólo para variables numéricas. Por ejemplo, podemos pegar dos strings con una flecha en el medio

```
funcion_prueba <- function(parametro1,parametro2) {
  paste(parametro1, parametro2, sep = " <--> ")
}
```

```
funcion_prueba(parametro1 = "A ver", parametro2 = "Que pasa")
```

```
## [1] "A ver <--> Que pasa"
```

También podemos asignar un valor por default para los parametros en caso de que el usuario no defina su valor al utilizar la función.

```
Otra_funcion_prueba <- function(parametro1 ,parametro2 = "String default") {
  paste(parametro1, parametro2, sep = " <--> ")
}
```

```
}
Otra_funcion_prueba(parametro1 = "Valor 1 ")
```

```
## [1] "Valor 1  <--> String default"
```

Las funciones que creamos nosotros permanecen en el ambiente de R temporariamente. Cuando removemos los objetos del ambiente, la función deja de existir. Por ende, debemos incorporarla en cada uno de los scripts en la cual la necesitamos. Una buena práctica, es incorporar nuestras funciones útiles al comienzo de cada script junto a la carga de las librerías.

Vale mencionar que **lo que ocurre en una función, queda en la función** excepto que explícitamente pidamos que devuelva el resultado, con el comando `print()`.

Las funciones siempre devuelven el último objeto que se crea en ellas, o si explícitamente se utiliza el comando `return()`

3.1.4 PURRR¹

MAP es la forma *tidy* de hacer loops. Además de ser más prolijo el código, es mucho más eficiente.

La función **map** toma un input, una función para aplicar, y alguna otra cosa (por ejemplo parametros que necesite la función)

- `map(.x, .f, ...)`
- `map(VECTOR_O_LIST_INPUT, FUNCTION_A_APLICAR, OTROS OPCIONALES)`

Usamos **map2** cuando tenemos que pasar dos input, que se aplican sobre una función:

- `map2(.x, .y, .f, ...)`
- `map2(INPUT_UNO, INPUT_DOS, FUNCTION_A_APLICAR, OTROS OPCIONALES)`

Si tenemos más de dos...

- `pmap(.l, .f, ...)`
- `pmap(VECTOR_O_LIST_INPUT, FUNCTION_A_APLICAR, OTROS OPCIONALES)`

Por ejemplo. Si queremos utilizar la función prueba sobre los datos del dataframe ABC_123

```
ABC_123 <- data.frame(Letras = LETTERS[1:20], Num = 1:20)
funcion_prueba
```

¹basado en https://jennybc.github.io/purrr-tutorial/ls03_map-function-syntax.html

```
## function(parametro1,parametro2) {
##   paste(parametro1, parametro2, sep = " <--> ")
## }
```

Si el resultado que queremos es que junte cada fila, necesitamos pasarle dos parámetros: utilizamos `map2()`

```
resultado <- map2(ABC_123$Letras,ABC_123$Num,funcion_prueba)
resultado[1:3]
```

```
## [[1]]
## [1] "A <--> 1"
##
## [[2]]
## [1] "B <--> 2"
##
## [[3]]
## [1] "C <--> 3"
```

La salida de los `map()` es una **lista**, no un vector, por lo que si lo metemos dentro de un dataframe se vería así:

```
ABC_123 %>%
  mutate(resultado= map2(Letras,Num,funcion_prueba))
```

```
##   Letras Num resultado
## 1      A   1 A <--> 1
## 2      B   2 B <--> 2
## 3      C   3 C <--> 3
## 4      D   4 D <--> 4
## 5      E   5 E <--> 5
## 6      F   6 F <--> 6
## 7      G   7 G <--> 7
## 8      H   8 H <--> 8
## 9      I   9 I <--> 9
## 10     J  10 J <--> 10
## 11     K  11 K <--> 11
## 12     L  12 L <--> 12
## 13     M  13 M <--> 13
## 14     N  14 N <--> 14
## 15     O  15 O <--> 15
## 16     P  16 P <--> 16
## 17     Q  17 Q <--> 17
## 18     R  18 R <--> 18
## 19     S  19 S <--> 19
## 20     T  20 T <--> 20
```

al ponerlo dentro del dataframe desarma la lista y guarda cada elemento por separado. La magia de eso es que podemos **guardar cualquier cosa en el**

dataframe no sólo valores, sino también listas, funciones, dataframes, etc.

Si queremos recuperar los valores originales en este caso podemos usar `unlist()`

```
resultado[1:3] %>% unlist()
```

```
## [1] "A <--> 1" "B <--> 2" "C <--> 3"
```

```
ABC_123 %>%
```

```
  mutate(resultado= unlist(map2(Letras,Num,funcion_prueba)))
```

```
##      Letras Num resultado
## 1      A     1 A <--> 1
## 2      B     2 B <--> 2
## 3      C     3 C <--> 3
## 4      D     4 D <--> 4
## 5      E     5 E <--> 5
## 6      F     6 F <--> 6
## 7      G     7 G <--> 7
## 8      H     8 H <--> 8
## 9      I     9 I <--> 9
## 10     J    10 J <--> 10
## 11     K    11 K <--> 11
## 12     L    12 L <--> 12
## 13     M    13 M <--> 13
## 14     N    14 N <--> 14
## 15     O    15 O <--> 15
## 16     P    16 P <--> 16
## 17     Q    17 Q <--> 17
## 18     R    18 R <--> 18
## 19     S    19 S <--> 19
## 20     T    20 T <--> 20
```

Si lo que queríamos era que la función nos haga todas las combinaciones de letras y número, entonces lo que necesitamos es pasarle el segundo parametro como algo *fijo*, poniendolo después de la función.

```
map(ABC_123$Letras,funcion_prueba,ABC_123$Num)[1:2]
```

```
## [[1]]
## [1] "A <--> 1" "A <--> 2" "A <--> 3" "A <--> 4" "A <--> 5"
## [6] "A <--> 6" "A <--> 7" "A <--> 8" "A <--> 9" "A <--> 10"
## [11] "A <--> 11" "A <--> 12" "A <--> 13" "A <--> 14" "A <--> 15"
## [16] "A <--> 16" "A <--> 17" "A <--> 18" "A <--> 19" "A <--> 20"
##
## [[2]]
## [1] "B <--> 1" "B <--> 2" "B <--> 3" "B <--> 4" "B <--> 5"
## [6] "B <--> 6" "B <--> 7" "B <--> 8" "B <--> 9" "B <--> 10"
## [11] "B <--> 11" "B <--> 12" "B <--> 13" "B <--> 14" "B <--> 15"
```

```
## [16] "B <--> 16" "B <--> 17" "B <--> 18" "B <--> 19" "B <--> 20"
```

En este caso, el map itera sobre cada elemento de `letras`, y para cada elemento i hace `funcion_prueba(i,ABC$Num)` y guarda el resultado en la lista

si lo queremos meter en el dataframe

```
ABC_123 %>%
  mutate(resultado= map(Letras,funcion_prueba,Num))
```

```
##   Letras Num
## 1      A   1
## 2      B   2
## 3      C   3
## 4      D   4
## 5      E   5
## 6      F   6
## 7      G   7
## 8      H   8
## 9      I   9
## 10     J  10
## 11     K  11
## 12     L  12
## 13     M  13
## 14     N  14
## 15     O  15
## 16     P  16
## 17     Q  17
## 18     R  18
## 19     S  19
## 20     T  20
##
```

```
## 1 A <--> 1, A <--> 2, A <--> 3, A <--> 4, A <--> 5, A <--> 6, A <--> 7, A <--> 8, A <--> 9, A <--> 10, A <--> 11, A <--> 12, A <--> 13, A <--> 14, A <--> 15, A <--> 16, A <--> 17, A <--> 18, A <--> 19, A <--> 20
## 2 B <--> 1, B <--> 2, B <--> 3, B <--> 4, B <--> 5, B <--> 6, B <--> 7, B <--> 8, B <--> 9, B <--> 10, B <--> 11, B <--> 12, B <--> 13, B <--> 14, B <--> 15, B <--> 16, B <--> 17, B <--> 18, B <--> 19, B <--> 20
## 3 C <--> 1, C <--> 2, C <--> 3, C <--> 4, C <--> 5, C <--> 6, C <--> 7, C <--> 8, C <--> 9, C <--> 10, C <--> 11, C <--> 12, C <--> 13, C <--> 14, C <--> 15, C <--> 16, C <--> 17, C <--> 18, C <--> 19, C <--> 20
## 4 D <--> 1, D <--> 2, D <--> 3, D <--> 4, D <--> 5, D <--> 6, D <--> 7, D <--> 8, D <--> 9, D <--> 10, D <--> 11, D <--> 12, D <--> 13, D <--> 14, D <--> 15, D <--> 16, D <--> 17, D <--> 18, D <--> 19, D <--> 20
## 5 E <--> 1, E <--> 2, E <--> 3, E <--> 4, E <--> 5, E <--> 6, E <--> 7, E <--> 8, E <--> 9, E <--> 10, E <--> 11, E <--> 12, E <--> 13, E <--> 14, E <--> 15, E <--> 16, E <--> 17, E <--> 18, E <--> 19, E <--> 20
## 6 F <--> 1, F <--> 2, F <--> 3, F <--> 4, F <--> 5, F <--> 6, F <--> 7, F <--> 8, F <--> 9, F <--> 10, F <--> 11, F <--> 12, F <--> 13, F <--> 14, F <--> 15, F <--> 16, F <--> 17, F <--> 18, F <--> 19, F <--> 20
## 7 G <--> 1, G <--> 2, G <--> 3, G <--> 4, G <--> 5, G <--> 6, G <--> 7, G <--> 8, G <--> 9, G <--> 10, G <--> 11, G <--> 12, G <--> 13, G <--> 14, G <--> 15, G <--> 16, G <--> 17, G <--> 18, G <--> 19, G <--> 20
## 8 H <--> 1, H <--> 2, H <--> 3, H <--> 4, H <--> 5, H <--> 6, H <--> 7, H <--> 8, H <--> 9, H <--> 10, H <--> 11, H <--> 12, H <--> 13, H <--> 14, H <--> 15, H <--> 16, H <--> 17, H <--> 18, H <--> 19, H <--> 20
## 9 I <--> 1, I <--> 2, I <--> 3, I <--> 4, I <--> 5, I <--> 6, I <--> 7, I <--> 8, I <--> 9, I <--> 10, I <--> 11, I <--> 12, I <--> 13, I <--> 14, I <--> 15, I <--> 16, I <--> 17, I <--> 18, I <--> 19, I <--> 20
## 10 J <--> 1, J <--> 2, J <--> 3, J <--> 4, J <--> 5, J <--> 6, J <--> 7, J <--> 8, J <--> 9, J <--> 10, J <--> 11, J <--> 12, J <--> 13, J <--> 14, J <--> 15, J <--> 16, J <--> 17, J <--> 18, J <--> 19, J <--> 20
## 11 K <--> 1, K <--> 2, K <--> 3, K <--> 4, K <--> 5, K <--> 6, K <--> 7, K <--> 8, K <--> 9, K <--> 10, K <--> 11, K <--> 12, K <--> 13, K <--> 14, K <--> 15, K <--> 16, K <--> 17, K <--> 18, K <--> 19, K <--> 20
## 12 L <--> 1, L <--> 2, L <--> 3, L <--> 4, L <--> 5, L <--> 6, L <--> 7, L <--> 8, L <--> 9, L <--> 10, L <--> 11, L <--> 12, L <--> 13, L <--> 14, L <--> 15, L <--> 16, L <--> 17, L <--> 18, L <--> 19, L <--> 20
## 13 M <--> 1, M <--> 2, M <--> 3, M <--> 4, M <--> 5, M <--> 6, M <--> 7, M <--> 8, M <--> 9, M <--> 10, M <--> 11, M <--> 12, M <--> 13, M <--> 14, M <--> 15, M <--> 16, M <--> 17, M <--> 18, M <--> 19, M <--> 20
## 14 N <--> 1, N <--> 2, N <--> 3, N <--> 4, N <--> 5, N <--> 6, N <--> 7, N <--> 8, N <--> 9, N <--> 10, N <--> 11, N <--> 12, N <--> 13, N <--> 14, N <--> 15, N <--> 16, N <--> 17, N <--> 18, N <--> 19, N <--> 20
## 15 O <--> 1, O <--> 2, O <--> 3, O <--> 4, O <--> 5, O <--> 6, O <--> 7, O <--> 8, O <--> 9, O <--> 10, O <--> 11, O <--> 12, O <--> 13, O <--> 14, O <--> 15, O <--> 16, O <--> 17, O <--> 18, O <--> 19, O <--> 20
```

```
## 16 P <--> 1, P <--> 2, P <--> 3, P <--> 4, P <--> 5, P <--> 6, P <--> 7, P <--> 8, P <--> 9, P <--> 10,
## 17 Q <--> 1, Q <--> 2, Q <--> 3, Q <--> 4, Q <--> 5, Q <--> 6, Q <--> 7, Q <--> 8, Q <--> 9, Q <--> 10,
## 18 R <--> 1, R <--> 2, R <--> 3, R <--> 4, R <--> 5, R <--> 6, R <--> 7, R <--> 8, R <--> 9, R <--> 10,
## 19 S <--> 1, S <--> 2, S <--> 3, S <--> 4, S <--> 5, S <--> 6, S <--> 7, S <--> 8, S <--> 9, S <--> 10,
## 20 T <--> 1, T <--> 2, T <--> 3, T <--> 4, T <--> 5, T <--> 6, T <--> 7, T <--> 8, T <--> 9, T <--> 10,
```

Ahora cada fila tiene un vector de 20 elementos guardado en la columna resultado

3.1.5 Funciones implícitas

no es necesario que definamos la función de antemano. Podemos usar *funciones implícitas*

```
map_dbl(c(1:10), function(x) x^2)
```

```
## [1] 1 4 9 16 25 36 49 64 81 100
```

```
map2_dbl(c(1:10),c(11:20), function(x,y) x*y)
```

```
## [1] 11 24 39 56 75 96 119 144 171 200
```

3.1.6 Funciones lambda

incluso más conciso que las funciones implícitas son las **funciones lambda** donde definimos las variables como *.x* *.y*, etc. La flexibilidad de estas expresiones es limitada, pero puede ser útil en algunos casos.

```
map_dbl(c(1:10), ~.x^2)
```

```
## [1] 1 4 9 16 25 36 49 64 81 100
```

```
map2_dbl(c(1:10),c(11:20), ~.x*.y)
```

```
## [1] 11 24 39 56 75 96 119 144 171 200
```

3.1.7 Walk

Las funciones Walk Tienen la misma forma que los map, pero se usan cuando lo que queremos iterar no genera una salida, sino que nos interesan los efectos secundarios que generan.

```
map2(ABC_123$Letras,ABC_123$Num,function_prueba)[1:3]
```

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

```
## [1] "A <--> 1"
```

```
##
```

```
## [[2]]
```

```
## [1] "B <--> 2"
```

```
##
```



```
## [[3]]
## [1] "C <--> 3"

walk2(ABC_123$Letras,ABC_123$Num,funcion_prueba)

imprimir_salida <- function(x,y){
  print(funcion_prueba(x,y))
}

walk2(ABC_123$Letras,ABC_123$Num,imprimir_salida)

## [1] "A <--> 1"
## [1] "B <--> 2"
## [1] "C <--> 3"
## [1] "D <--> 4"
## [1] "E <--> 5"
## [1] "F <--> 6"
## [1] "G <--> 7"
## [1] "H <--> 8"
## [1] "I <--> 9"
## [1] "J <--> 10"
## [1] "K <--> 11"
## [1] "L <--> 12"
## [1] "M <--> 13"
## [1] "N <--> 14"
## [1] "O <--> 15"
## [1] "P <--> 16"
## [1] "Q <--> 17"
## [1] "R <--> 18"
## [1] "S <--> 19"
## [1] "T <--> 20"
```

Eso que vemos es el efecto secundario dentro de la función (imprimir)

3.1.8 Cuando usar estas herramientas?

A lo largo del curso vimos diferentes técnicas para manipulación de datos. En particular, la librería `dplyr` nos permitía fácilmente modificar y crear nuevas variables, agrupando. Cuando usamos `dplyr` y cuando usamos `purrr`.

- Si trabajamos sobre un DF simple, sin variables anidadas (lo que conocíamos hasta hoy) podemos usar `dplyr`
- Si queremos trabajar con DF anidados, con cosas que no son DF, o si el resultado de la operación que vamos a realizar a nivel file es algo distinto a un valor único, nos conviene usar `map` y `purrr`
- Las funciones `walk` son útiles por ejemplo para escribir archivos en disco de forma iterativa. Algo que no genera una salida

3.2 Práctica Guiada

```
library(fs)
library(tidyverse)
library(openxlsx)
library(glue)
```

3.2.1 Ejemplo 1: Iterando en la EPH

Lo primero que necesitamos es definir un vector o lista sobre el que iterar.

Por ejemplo, podemos armar un vector con los path a las bases individuales, con el comando `fs::dir_ls`

```
bases_individuales_path <- dir_ls(path = 'fuentes/', regexp= 'individual')
bases_individuales_path
```

```
## fuentes/usu_individual_t119.txt fuentes/usu_individual_t418.txt
```

Luego, como en la función que usamos para leer las bases definimos muchos parametros, nos podemos armar una función *wrapper* que sólo necesite un parámetro, y que simplifique la escritura del map

```
leer_base_eph <- function(path) {
  read.table(path, sep=";", dec=".", header = TRUE, fill = TRUE) %>%
    select(ANO4, TRIMESTRE, REGION, P21, CH04, CH06)
}
```

```
bases_df <- tibble(bases_individuales_path) %>%
  mutate(base = map(bases_individuales_path, leer_base_eph))
```

```
bases_df
```

```
## # A tibble: 2 x 2
##   bases_individuales_path      base
##   <fs::path>                <list>
## 1 fuentes/usu_individual_t119.txt <df[,6] [59,369 x 6]>
## 2 fuentes/usu_individual_t418.txt <df[,6] [57,418 x 6]>
```

El resultado es un DF donde la columna **base** tiene en cada fila, otro DF con la base de la EPH de ese período. Esto es lo que llamamos un *nested DF* o dataframe nestado pa les pibes.

Si queremos juntar todo, podemos usar `unnest()`

```
bases_df <- bases_df %>% unnest()
bases_df
```

```
## # A tibble: 116,787 x 7
##   bases_individuales_path      ANO4 TRIMESTRE REGION  P21  CH04  CH06
```

```
##      <fs::path>                <int>      <int> <int> <int> <int> <int>
## 1 fuentes/usu_individual_t119.txt 2019        1    41    0    2    28
## 2 fuentes/usu_individual_t119.txt 2019        1    41    0    2    13
## 3 fuentes/usu_individual_t119.txt 2019        1    41    0    1     1
## 4 fuentes/usu_individual_t119.txt 2019        1    41 5000    2    41
## 5 fuentes/usu_individual_t119.txt 2019        1    41    0    2     9
## 6 fuentes/usu_individual_t119.txt 2019        1    41 8000    1    51
## 7 fuentes/usu_individual_t119.txt 2019        1    41    0    1    63
## 8 fuentes/usu_individual_t119.txt 2019        1    41    0    2    62
## 9 fuentes/usu_individual_t119.txt 2019        1    41    0    2    24
## 10 fuentes/usu_individual_t119.txt 2019        1    41 3000    1    74
## # ... with 116,777 more rows
```

¿Qué pasa si los DF que tenemos nesteados no tienen la misma cantidad de columnas?

Esto mismo lo podemos usar para fragmentar el dataset por alguna variable, con el `group_by()`

```
bases_df %>%
  group_by(REGION) %>%
  nest()
```

```
## # A tibble: 6 x 2
##   REGION data
##   <int> <list>
## 1     41 <tibble [11,509 x 6]>
## 2     44 <tibble [14,204 x 6]>
## 3     42 <tibble [11,150 x 6]>
## 4     43 <tibble [34,702 x 6]>
## 5     40 <tibble [24,432 x 6]>
## 6      1 <tibble [20,790 x 6]>
```

Así, para cada región tenemos un DF.

¿De qué sirve todo esto?

No todo en la vida es un Dataframe. Hay estructuras de datos que no se pueden normalizar a filas y columnas. En esos casos recurríamos tradicionalmente a los loops. Con MAP podemos tener los elementos agrupados en un sólo objeto y aún conservar sus formas diferentes.

3.2.2 Ejemplo 2. Regresión lineal

Si bien no nos vamos a meter en el detalle del modelo lineal hoy, es útil usarlo como ejemplo de lo que podemos hacer con MAP.

Planteamos el modelo

$$P21 = \beta_0 + \beta_1 * CH04 + \beta_2 * CH06$$

Osea, un modleo que explica el ingreso según sexo y edad

```
lmfit <- lm(P21~factor(CH04)+CH06,data = bases_df)

summary(lmfit)

##
## Call:
## lm(formula = P21 ~ factor(CH04) + CH06, data = bases_df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -15472  -6606  -3367   2148  590198
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   4853.196     74.509   65.14  <2e-16 ***
## factor(CH04)2 -4063.112     72.200  -56.27  <2e-16 ***
## CH06           103.095      1.612   63.97  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 12300 on 116784 degrees of freedom
## Multiple R-squared:  0.05511,    Adjusted R-squared:  0.0551
## F-statistic: 3406 on 2 and 116784 DF,  p-value: < 2.2e-16
```

(al final de la clase podemos charlar sobre los resultados, si hay interés :-)

De forma Tidy, la librería broom nos da los resultados en un DF.

```
broom::tidy(lmfit)

## # A tibble: 3 x 5
##   term          estimate std.error statistic p.value
##   <chr>          <dbl>     <dbl>     <dbl>   <dbl>
## 1 (Intercept)    4853.      74.5       65.1     0
## 2 factor(CH04)2 -4063.      72.2      -56.3     0
## 3 CH06           103.       1.61      64.0     0
```

Si lo queremos hacer por region

3.2.2.1 Loopeando

```
resultados <- tibble()

for (region in unique(bases_df$REGION)) {

  data <- bases_df %>%
```

```

    filter(REGION==region)

    lmfit <- lm(P21~factor(CH04)+CH06,data = data)

    lmtidy <- broom::tidy(lmfit)
    lmtidy$region <- region
    resultados <- bind_rows(resultados,lmtidy)
  }

resultados

## # A tibble: 18 x 6
##   term                estimate std.error statistic    p.value region
##   <chr>                <dbl>    <dbl>    <dbl>    <dbl>    <int>
## 1 (Intercept)         3768.      185.      20.3 3.15e- 90     41
## 2 factor(CH04)2      -3814.      180.     -21.2 6.00e- 98     41
## 3 CH06                 106.       4.18      25.3 1.12e-137    41
## 4 (Intercept)         7156.      291.      24.6 1.09e-130    44
## 5 factor(CH04)2     -5938.      278.     -21.4 1.42e- 99     44
## 6 CH06                 145.       6.32      23.0 1.40e-114    44
## 7 (Intercept)         4930.      231.      21.4 2.15e- 99     42
## 8 factor(CH04)2     -4007.      224.     -17.9 1.71e- 70     42
## 9 CH06                 97.8       4.95      19.7 2.68e- 85     42
## 10 (Intercept)        5107.      131.      39.0 0.         43
## 11 factor(CH04)2     -3949.      127.     -31.1 5.02e-209    43
## 12 CH06                 83.5       2.78      30.0 3.87e-195    43
## 13 (Intercept)        3329.      128.      26.0 4.12e-147    40
## 14 factor(CH04)2     -3239.      125.     -25.9 3.74e-146    40
## 15 CH06                 122.       2.89      42.2 0.         40
## 16 (Intercept)        5196.      197.      26.4 3.45e-151     1
## 17 factor(CH04)2     -4051.      189.     -21.4 1.80e-100     1
## 18 CH06                 88.2       4.12      21.4 1.98e-100     1

```

3.2.2.2 Usando MAP

Primero me armo una funcion que me simplifica el codigo

```

fun<-function(porcion,grupo) {  broom::tidy(lm(P21~factor(CH04)+CH06,data = porcion))}

bases_df_lm <- bases_df %>%
  group_by(REGION) %>%
  nest() %>%
  mutate(lm = map(data,fun))
bases_df_lm

## # A tibble: 6 x 3

```

```
## REGION data          lm
##   <int> <list>        <list>
## 1     41 <tibble [11,509 x 6]> <tibble [3 x 5]>
## 2     44 <tibble [14,204 x 6]> <tibble [3 x 5]>
## 3     42 <tibble [11,150 x 6]> <tibble [3 x 5]>
## 4     43 <tibble [34,702 x 6]> <tibble [3 x 5]>
## 5     40 <tibble [24,432 x 6]> <tibble [3 x 5]>
## 6      1 <tibble [20,790 x 6]> <tibble [3 x 5]>
```

```
bases_df_lm %>%
  unnest(lm)
```

```
## # A tibble: 18 x 6
##   REGION term          estimate std.error statistic  p.value
##   <int> <chr>          <dbl>    <dbl>    <dbl>    <dbl>
## 1     41 (Intercept)    3768.    185.     20.3 3.15e- 90
## 2     41 factor(CH04)2 -3814.    180.    -21.2 6.00e- 98
## 3     41 CH06           106.     4.18     25.3 1.12e-137
## 4     44 (Intercept)    7156.    291.     24.6 1.09e-130
## 5     44 factor(CH04)2 -5938.    278.    -21.4 1.42e- 99
## 6     44 CH06           145.     6.32     23.0 1.40e-114
## 7     42 (Intercept)    4930.    231.     21.4 2.15e- 99
## 8     42 factor(CH04)2 -4007.    224.    -17.9 1.71e- 70
## 9     42 CH06           97.8     4.95     19.7 2.68e- 85
## 10    43 (Intercept)    5107.    131.     39.0 0.
## 11    43 factor(CH04)2 -3949.    127.    -31.1 5.02e-209
## 12    43 CH06            83.5     2.78     30.0 3.87e-195
## 13    40 (Intercept)    3329.    128.     26.0 4.12e-147
## 14    40 factor(CH04)2 -3239.    125.    -25.9 3.74e-146
## 15    40 CH06            122.     2.89     42.2 0.
## 16     1 (Intercept)    5196.    197.     26.4 3.45e-151
## 17     1 factor(CH04)2 -4051.    189.    -21.4 1.80e-100
## 18     1 CH06            88.2     4.12     21.4 1.98e-100
```

O incluso más facil, utilizando `group_modify` (que es un atajo que solo acepta DF)

```
bases_df %>%
  group_by(REGION) %>%
  group_modify(fun)
```

```
## # A tibble: 18 x 6
## # Groups:   REGION [6]
##   REGION term          estimate std.error statistic  p.value
##   <int> <chr>          <dbl>    <dbl>    <dbl>    <dbl>
## 1      1 (Intercept)    5196.    197.     26.4 3.45e-151
## 2      1 factor(CH04)2 -4051.    189.    -21.4 1.80e-100
## 3      1 CH06            88.2     4.12     21.4 1.98e-100
```

```
## 4    40 (Intercept)      3329.    128.    26.0 4.12e-147
## 5    40 factor(CH04)2 -3239.    125.   -25.9 3.74e-146
## 6    40 CH06           122.     2.89    42.2 0.
## 7    41 (Intercept)      3768.    185.    20.3 3.15e- 90
## 8    41 factor(CH04)2 -3814.    180.   -21.2 6.00e- 98
## 9    41 CH06           106.     4.18    25.3 1.12e-137
## 10   42 (Intercept)      4930.    231.    21.4 2.15e- 99
## 11   42 factor(CH04)2 -4007.    224.   -17.9 1.71e- 70
## 12   42 CH06            97.8     4.95    19.7 2.68e- 85
## 13   43 (Intercept)      5107.    131.    39.0 0.
## 14   43 factor(CH04)2 -3949.    127.   -31.1 5.02e-209
## 15   43 CH06            83.5     2.78    30.0 3.87e-195
## 16   44 (Intercept)      7156.    291.    24.6 1.09e-130
## 17   44 factor(CH04)2 -5938.    278.   -21.4 1.42e- 99
## 18   44 CH06           145.     6.32    23.0 1.40e-114
```

Pero MAP sirve para operar con cualquier objeto de R.

Por ejemplo podemos guardar el **objeto** `S3:lm` que es la regresión lineal entrenada. Ese objeto no es ni un vector, ni una lista, ni un DF. No es una estructura de datos, sino que es algo distinto, con *propiedades* como `predict()` para predecir, el `summary()` que vimos, etc.

```
fun<-function(porcion,grupo) {  lm(P21~factor(CH04)+CH06,data = porcion)}

bases_df %>%
  group_by(REGION) %>%
  nest() %>%
  mutate(lm = map(data,fun))
```

```
## # A tibble: 6 x 3
##   REGION data          lm
##   <int> <list>         <list>
## 1    41 <tibble [11,509 x 6]> <lm>
## 2    44 <tibble [14,204 x 6]> <lm>
## 3    42 <tibble [11,150 x 6]> <lm>
## 4    43 <tibble [34,702 x 6]> <lm>
## 5    40 <tibble [24,432 x 6]> <lm>
## 6     1 <tibble [20,790 x 6]> <lm>
```

3.2.3 Ejemplo 3: Gráficos en serie

Veamos un tercer ejemplo con otra base de datos que ya conocemos: Gapminder, que muestra algunos datos sobre la población de los países por año.

El objetivo de este ejercicio es hacer un gráfico por país de forma automática.

- Primero veamos los datos

```
library(gapminder)
```

```
gapminder_unfiltered %>%
  sample_n(10)
```

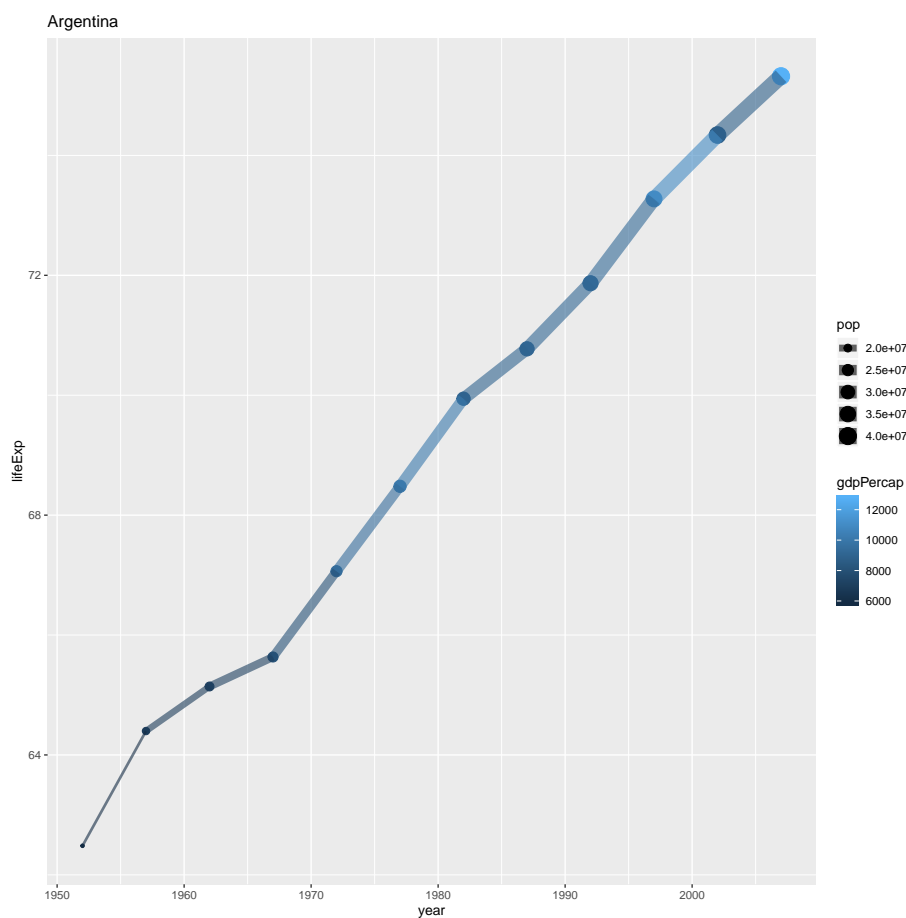
```
## # A tibble: 10 x 6
##   country      continent year lifeExp      pop gdpPercap
##   <fct>        <fct>    <int>   <dbl>    <int>    <dbl>
## 1 Malta        Europe    2007   79.4    401880   21712.
## 2 Denmark      Europe    1972   73.5   4991596   18866.
## 3 Mauritania   Africa    1977   50.9   1456688    1497.
## 4 Japan        Asia     1989   79.0  123107500  24705.
## 5 Guyana       Americas  2002   63.6   759104    3238.
## 6 West Bank and Gaza Asia     1967   51.6   1142636    2650.
## 7 Poland       Europe    1981   71.3  35901961   8607.
## 8 Sweden       Europe    1954   72.4   7213490   9225.
## 9 Guinea-Bissau Africa    1957   33.5   601095     432.
## 10 Kenya      Africa    1957   44.7   7454779    944.
```

la base tiene la siguiente info:

- country: Nombre del país
- continent: Nombre del continente
- year: año
- lifeExp: Esperanza de vida al nacer
- pop: Población
- gdpPercap
- Vamos a hacer un gráfico sencillo para Argentina

```
data_argentina <- gapminder_unfiltered %>%
  filter(country=='Argentina')
```

```
ggplot(data_argentina, aes(year, lifeExp, size= pop, color=gdpPercap))+
  geom_point()+
  geom_line(alpha=0.6)+
  labs(title = unique(data_argentina$country))
```

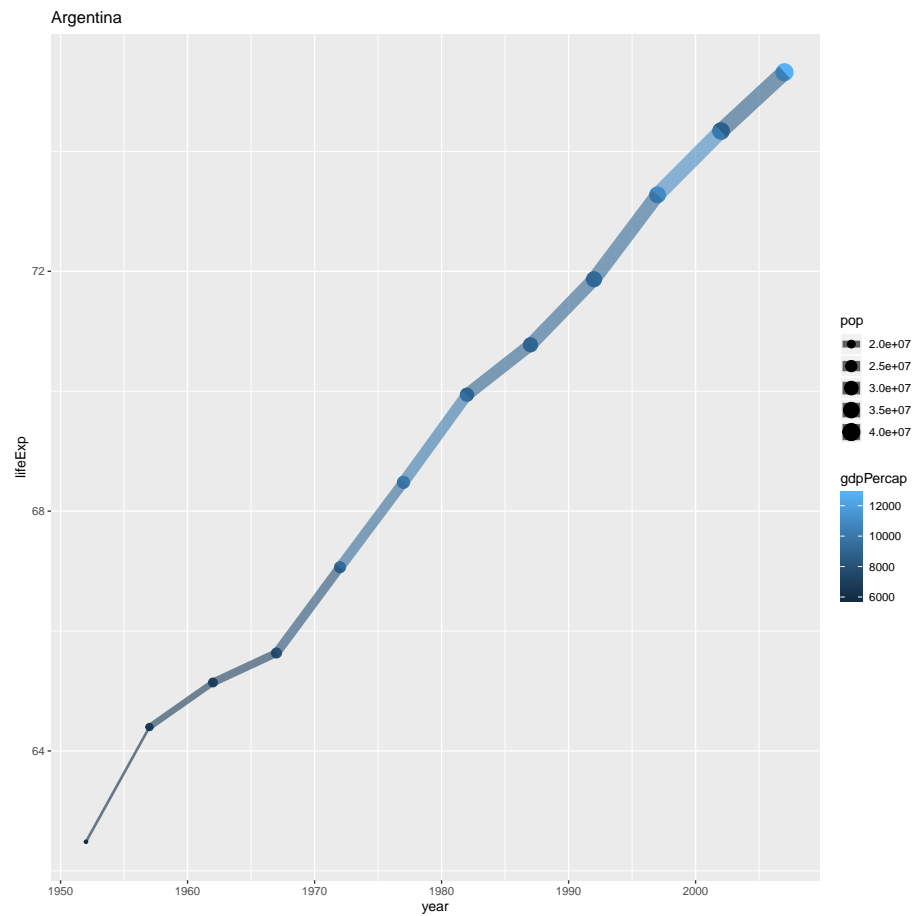
- Ahora que tenemos una idea de lo que queremos graficar lo podemos poner adentro de una función que grafique.

```
# definimos la función
graficar_pais <- function(data, pais){

  ggplot(data, aes(year, lifeExp, size= pop, color=gdpPercap))+
    geom_point()+
    geom_line(alpha=0.6)+
    labs(title = pais)
}
```

probamos la función para un caso

```
graficar_pais(data_argentina, 'Argentina')
```



- Nos armamos un dataset nestado

```
gapminder_nest <- gapminder_unfiltered %>%
  group_by(country) %>%
  nest()
```

```
gapminder_nest %>%
  sample_n(10)
```

```
## # A tibble: 10 x 2
##   country    data
##   <fct>      <list>
## 1 Kenya    <tibble [12 x 5]>
## 2 Hungary    <tibble [57 x 5]>
## 3 Belarus    <tibble [18 x 5]>
## 4 Sudan      <tibble [12 x 5]>
## 5 Ethiopia   <tibble [12 x 5]>
```

```
## 6 Afghanistan <tibble [12 x 5]>
## 7 Korea, Rep. <tibble [12 x 5]>
## 8 Chad <tibble [12 x 5]>
## 9 Timor-Leste <tibble [4 x 5]>
## 10 Samoa <tibble [7 x 5]>
```

- Ahora podemos crear una nueva columna que contenga los gráficos

```
gapminder_nest <- gapminder_nest %>%
  mutate(grafico= map2(.x = data, .y = country, .f = graficar_pais))

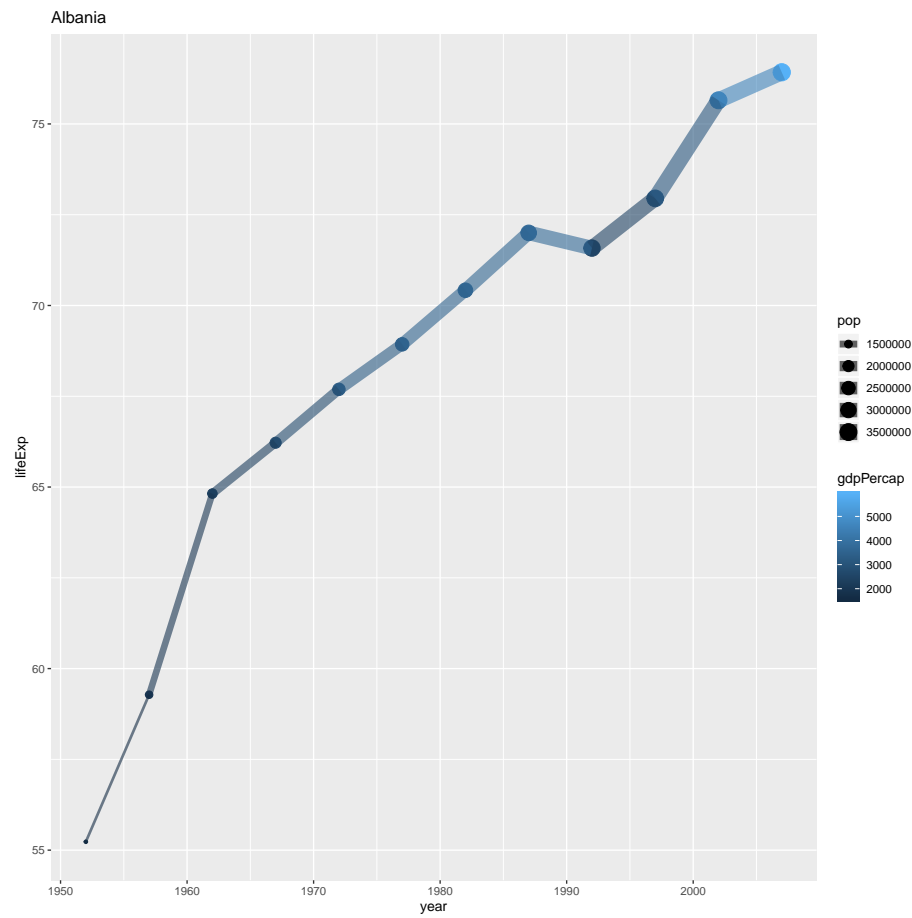
gapminder_nest %>%
  sample_n(10)
```

```
## # A tibble: 10 x 3
##   country          data          grafico
##   <fct>          <list>          <list>
## 1 Philippines    <tibble [12 x 5]> <gg>
## 2 China          <tibble [36 x 5]> <gg>
## 3 Italy          <tibble [56 x 5]> <gg>
## 4 Angola         <tibble [12 x 5]> <gg>
## 5 Cote d'Ivoire  <tibble [12 x 5]> <gg>
## 6 Guyana         <tibble [10 x 5]> <gg>
## 7 Morocco       <tibble [12 x 5]> <gg>
## 8 Martinique     <tibble [1 x 5]>  <gg>
## 9 Burundi        <tibble [12 x 5]> <gg>
## 10 Micronesia, Fed. Sts. <tibble [8 x 5]> <gg>
```

Veamos un ejemplo

```
gapminder_nest$grafico[2]
```

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



Ahora podemos guardar todos los gráficos en un archivo PDF

```
pdf('resultados/graficos_gapminder.pdf')
gapminder_nest$grafico
dev.off()
```

Chapter 4

Visualización de la información

En esta clase veremos como realizar gráficos en R, tanto los comandos básicos como utilizando la librería GGLOT.

- Gráficos básicos de R (función “plot”): Comandos para la visualización ágil de la información
- Gráficos elaborados en R (función “ggplot”):
- Gráficos de línea, barras, Boxplots y distribuciones de densidad
- Parámetros de los gráficos: Leyendas, ejes, títulos, notas, colores
- Gráficos con múltiples cruces de variables.

4.1 Explicación

4.1.1 Gráficos Básicos en R

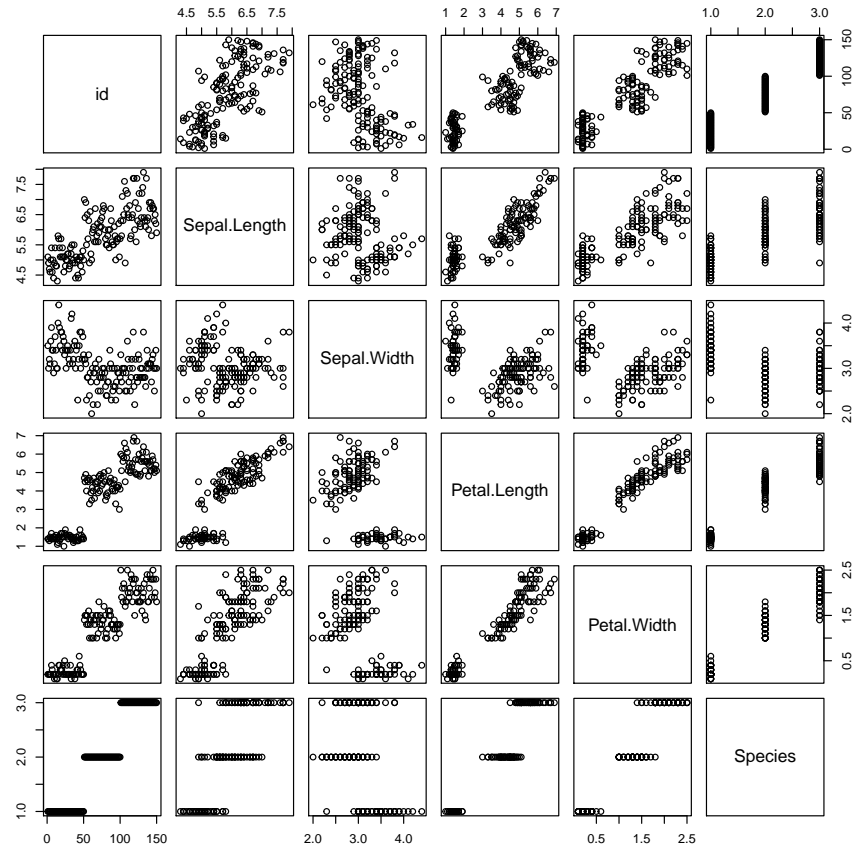
Rbase tiene algunos comandos genéricos para realizar gráficos, que se adaptan al tipo de información que se le pide graficar, por ejemplo:

- `plot()`
- `hist()`

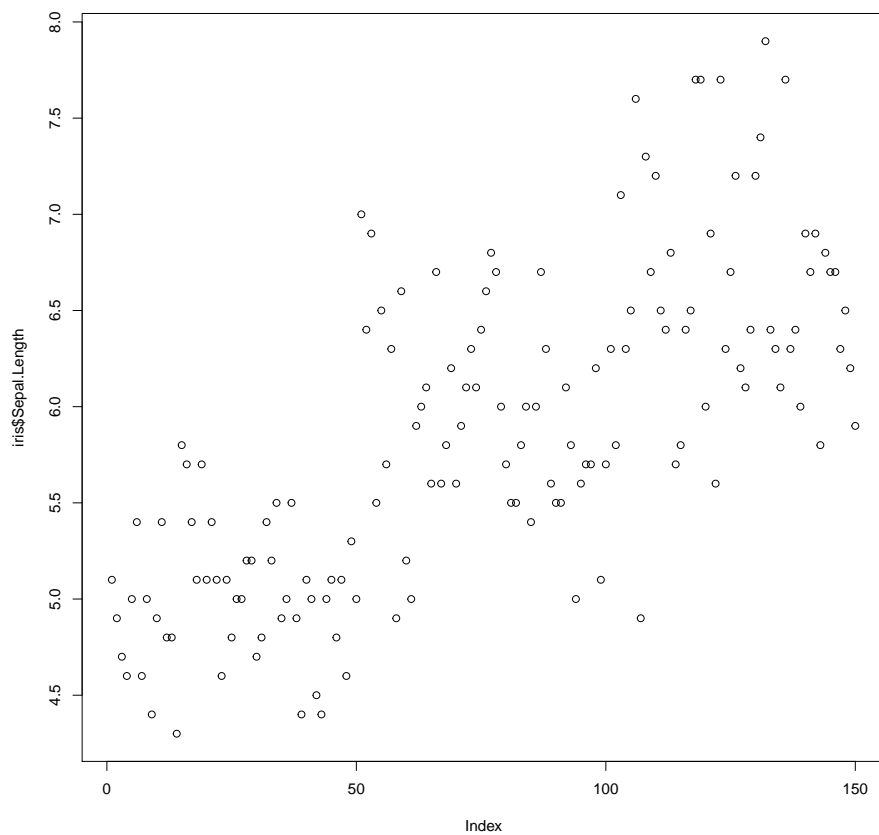
```
# iris es un set de datos clásico, que ya viene incorporado en R
iris[10,]
```

```
##      id Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 10  10           4.9           3.1           1.5           0.1   setosa

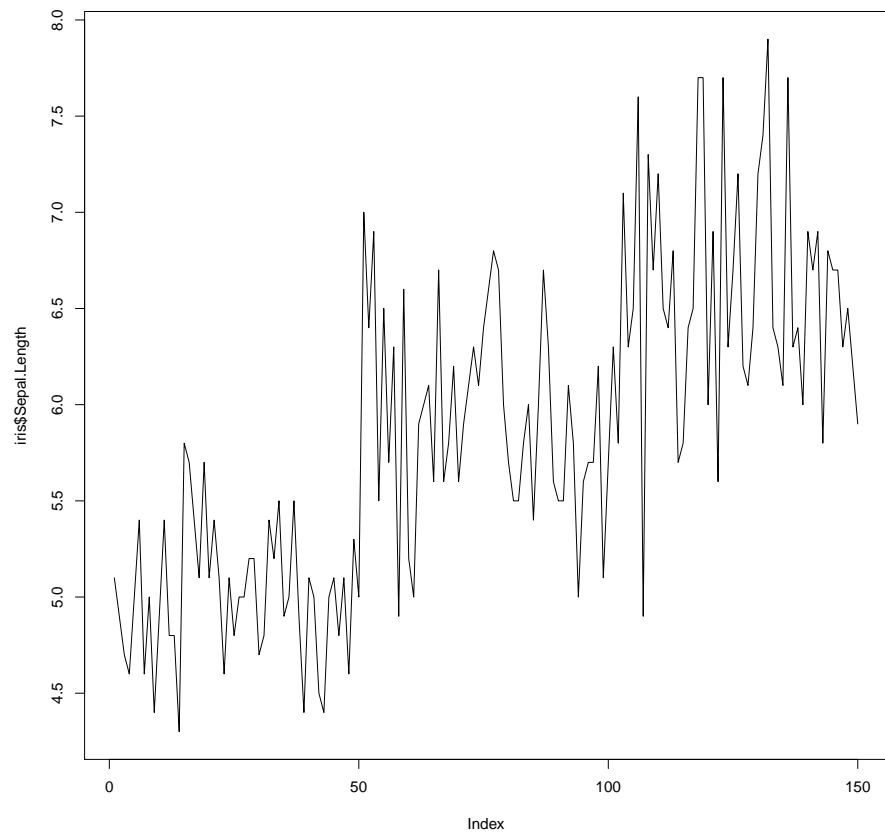
plot(iris)
```



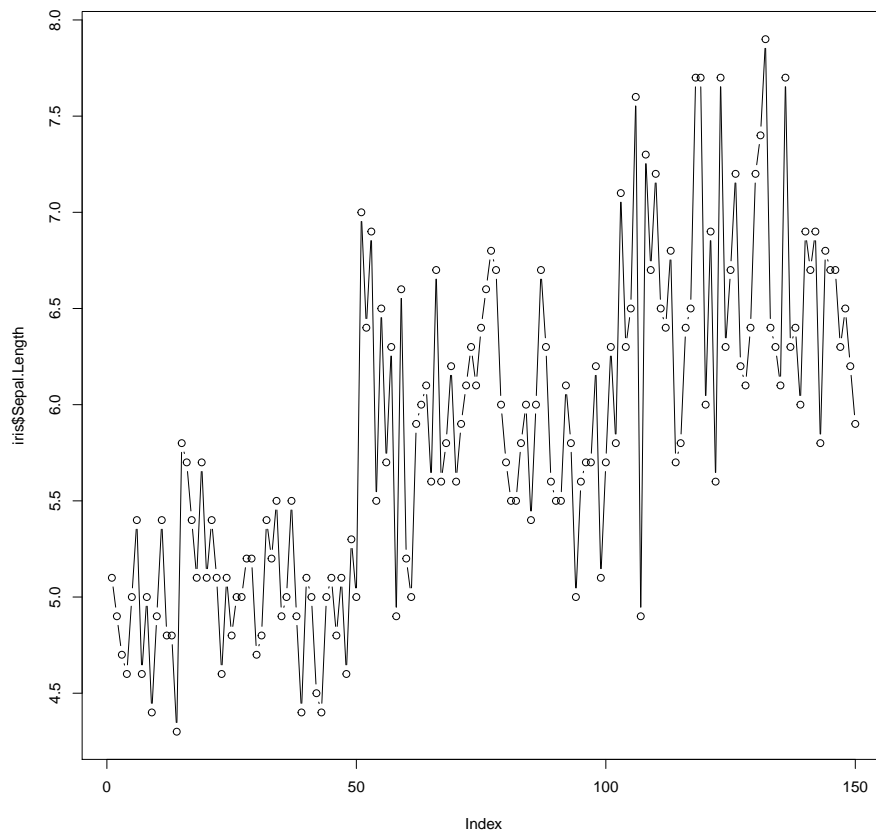
#Al especificar una variable, puedo ver el valor que toma cada uno de sus registros (Iris)
`plot(iris$Sepal.Length,type = "p")` *# Un punto por cada valor*



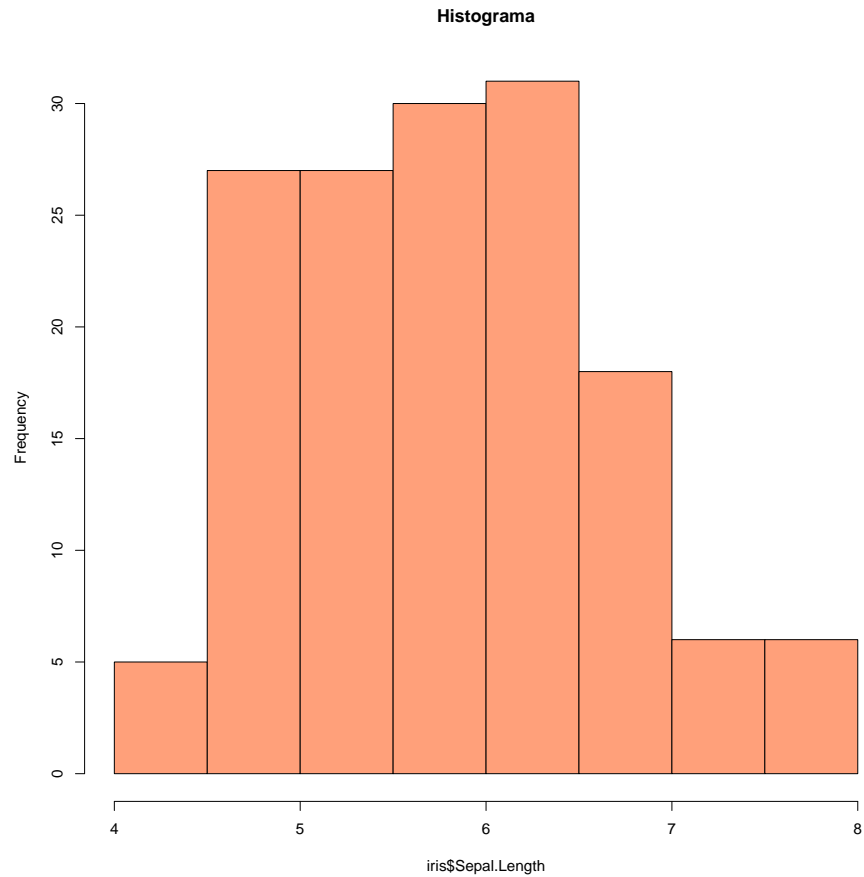
```
plot(iris$Sepal.Length,type = "l") # Una línea que una cada valor
```



```
plot(iris$Sepal.Length,type = "b") #Ambas
```

```
hist(iris$Sepal.Length, col = "lightsalmon1", main = "Histograma")
```



4.1.1.1 png

La función `png()` nos permite grabar una imagen en el disco. Lleva como argumento principal la ruta completa a donde se desea guardar la misma, incluyendo el nombre que queremos dar al archivo. A su vez pueden especificarse otros argumentos como el ancho y largo de la imagen, entre otros.

```
ruta_archivo <- "resultados/grafico1.PNG"
ruta_archivo
```

```
## [1] "resultados/grafico1.PNG"
```

```
png(ruta_archivo)
plot(iris$Sepal.Length,type = "b")
dev.off()
```

```
## pdf
```

```
## 2
```

La función `png()` *abre el dispositivo de imagen* en el directorio especificado. Luego creamos el gráfico que deseamos (o llamamos a uno previamente construido), el cual se desplegará en la ventana inferior derecha de la pantalla de Rstudio. Finalmente con `dev.off()` se *cierra el dispositivo* y se graban los gráficos.

Los gráficos del R base son útiles para escribir de forma rápida y obtener alguna información mientras trabajamos. Muchos paquetes estadísticos permiten mostrar los resultados de forma gráfica con el comando `plot` (por ejemplo, las regresiones lineales `lm()`).

Sin embargo, existen librerías mucho mejores para crear gráficos de nivel de publicación. La más importante es **ggplot2**, que a su vez tiene extensiones mediante otras librerías.

4.1.2 Ggplot2

ggplot tiene su sintaxis propia. La idea central es pensar los gráficos como una sucesión de capas, que se construyen una a la vez.

- El operador `+` nos permite incorporar nuevas capas al gráfico.
- El comando `ggplot()` nos permite definir la fuente de **datos** y las **variables** que determinaran los ejes del gráfico (x,y), así como el color y la forma de las líneas o puntos, etc.
- Las sucesivas capas nos permiten definir:
 - Uno o más tipos de gráficos (de columnas, `geom_col()`, de línea, `geom_line()`, de puntos, `geom_point()`, `boxplot`, `geom_boxplot()`)
 - Títulos `labs()`
 - Estilo del gráfico `theme()`
 - Escalas de los ejes `scale_y_continuous`, `scale_x_discrete`
 - División en subconjuntos `facet_wrap()`, `facet_grid()`

ggplot tiene **muchos** comandos, y no tiene sentido saberlos de memoria, es siempre útil reutilizar gráficos viejos y tener a mano el machete.

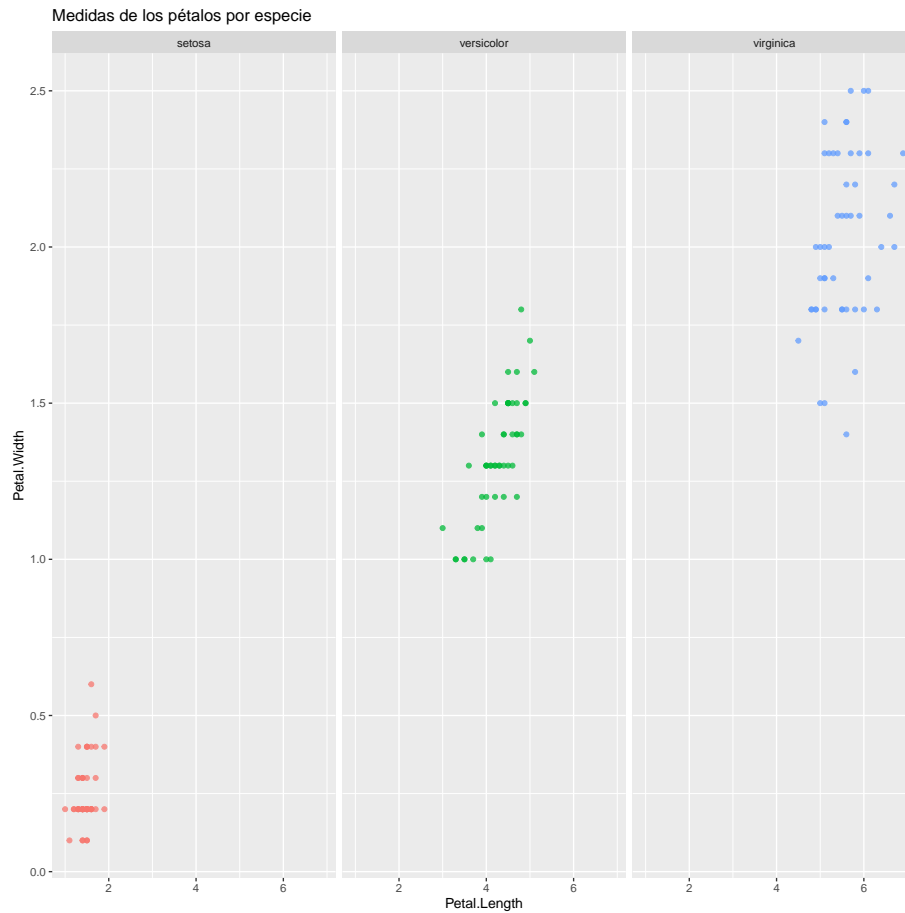
4.1.2.1 Gráfico de Puntos

A continuación se muestra un gráfico de varias capas de construcción, con su correspondiente porción de código. En el mismo se buscará visualizar, a partir de la base de datos **iris** la relación entre el ancho y el largo de los pétalos, mediante un gráfico de puntos.

```
library(tidyverse) # cargamos la librería

ggplot(data = iris, aes(x = Petal.Length, Petal.Width, color = Species))+
  geom_point(alpha=0.75)+
  labs(title = "Medidas de los pétalos por especie")+
```

```
theme(legend.position = 'none') +  
facet_wrap(~Species)
```

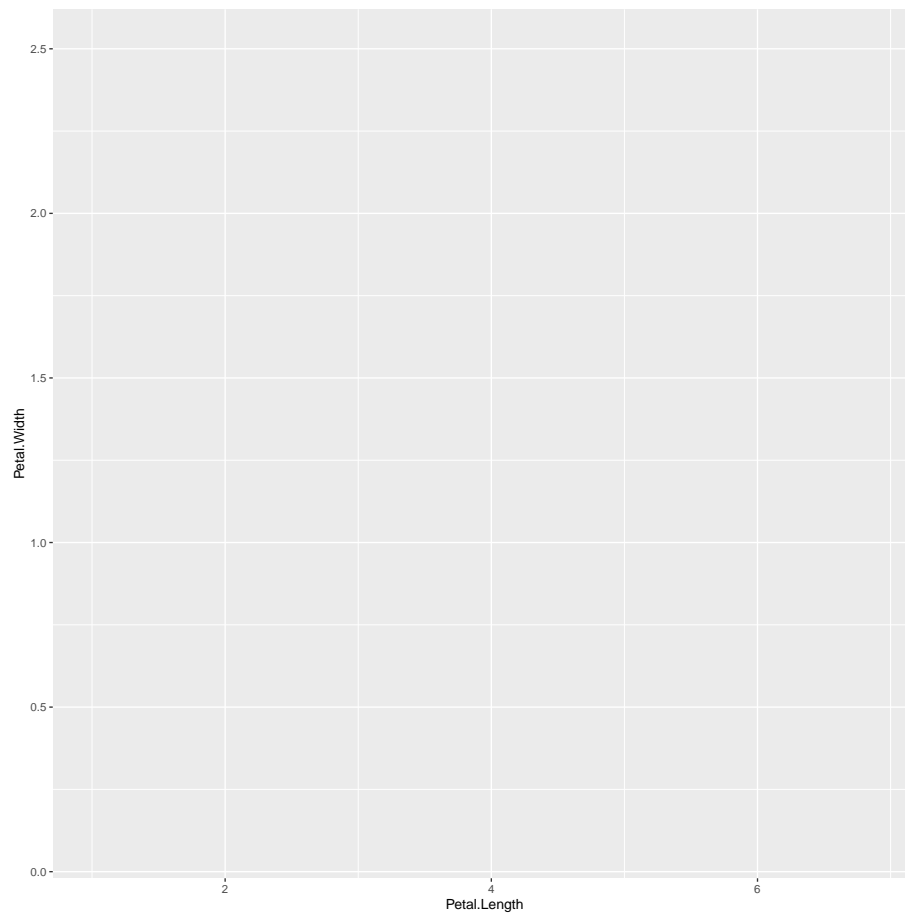


4.1.2.2 Capas del Gráfico

Veamos ahora, el “paso a paso” del armado del mismo.

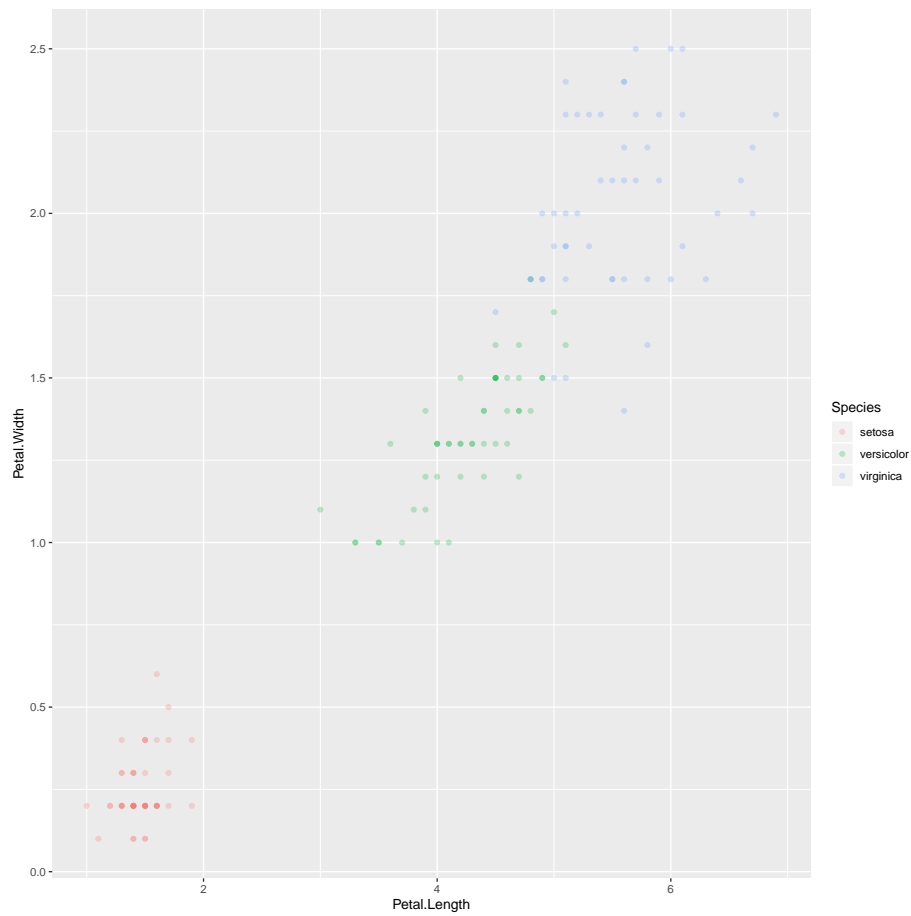
En primera instancia solo defino los ejes. Y en este caso un color particular para cada Especie.

```
g <- ggplot(data = iris, aes(x = Petal.Length, Petal.Width, color = Species))  
g
```



Luego, defino el tipo de gráfico. El *alpha* me permite definir la intensidad de los puntos

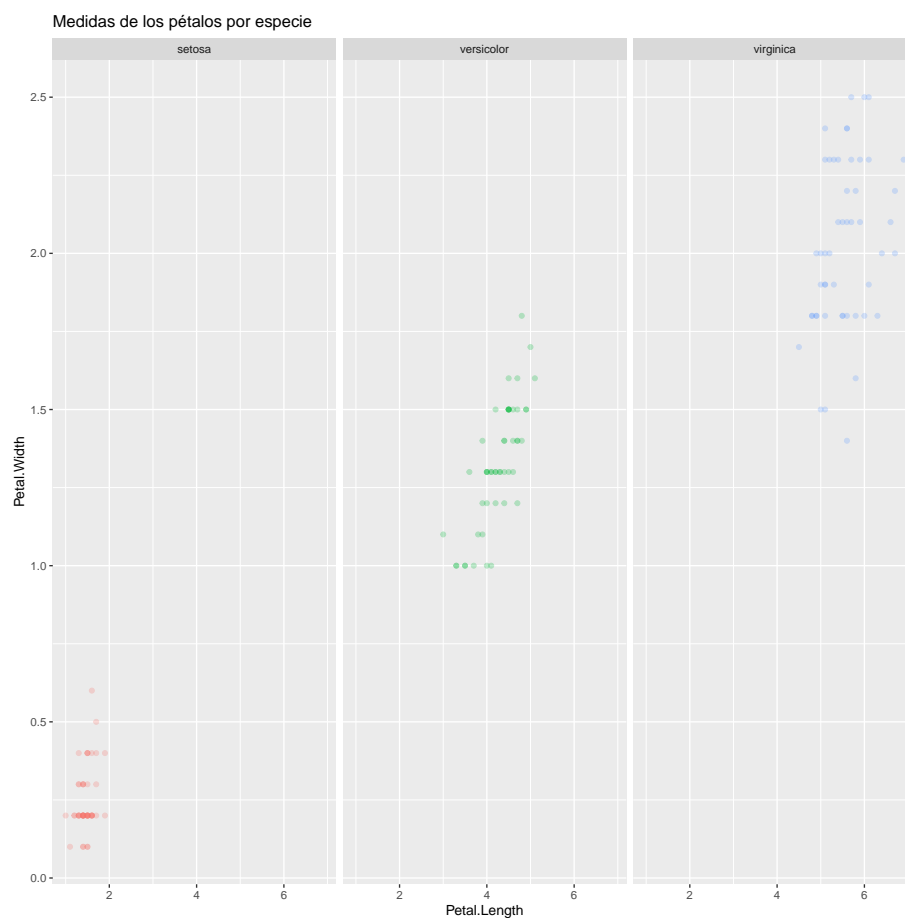
```
g <- g + geom_point(alpha=0.25)
g
```



Las siguientes tres capas me permiten respectivamente:

- Definir el título del gráfico
- Quitar la leyenda
- Abrir el gráfico en tres fragmentos, uno para cada especie

```
g <- g +
  labs(title = "Medidas de los pétalos por especie")+
  theme(legend.position = 'none')+
  facet_wrap(~Species)
g
```



4.1.2.3 Extensiones de GGplot.

La librería GGplot tiene a su vez muchas otras librerías que extienden sus potencialidades. Entre nuestras favoritas están:

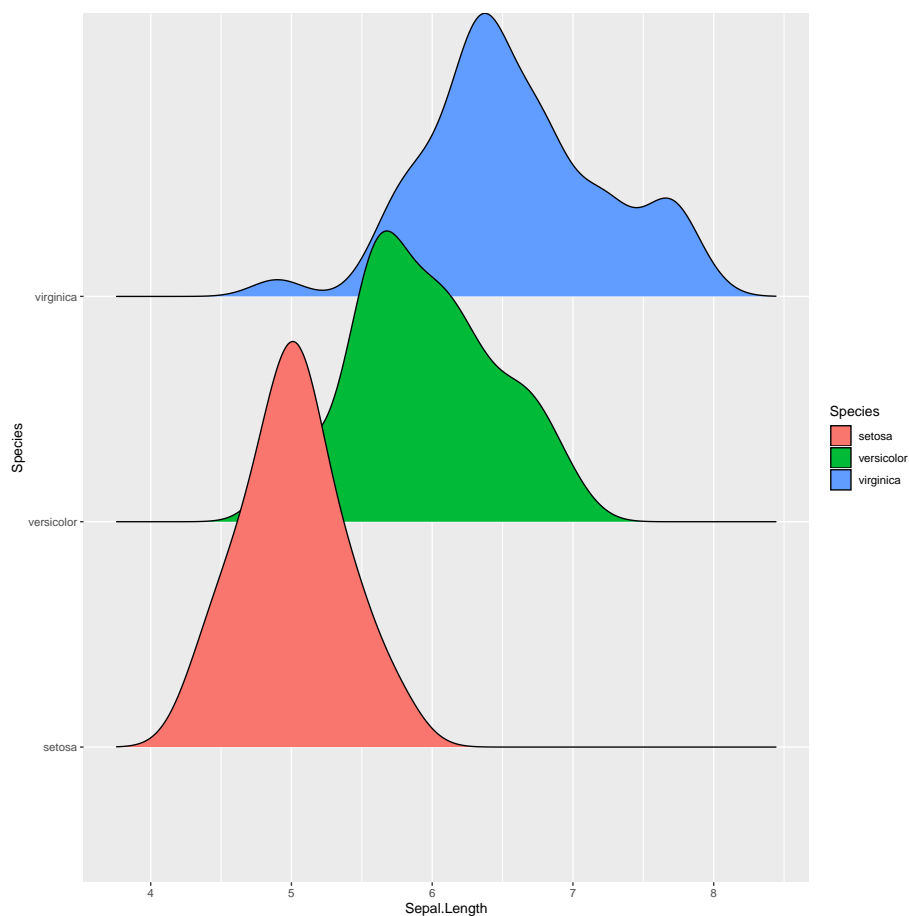
- `gganimate`: Para hacer gráficos animados.
- `ggridge`: Para hacer gráficos de densidad faceteados
- `ggally`: Para hacer varios gráficos juntos. [^]

```
library(GGally)
```

```
ggpairs(iris, mapping = aes(color = Species))
```



```
library(ggribes)
ggplot(iris, aes(x = Sepal.Length, y = Species, fill=Species)) +
  geom_density_ridges()
```

También hay extensiones que te ayudan a escribir el código, como `esquisse`

```
iris <- iris
#Correr en la consola
esquisse::esquisser()
```

4.1.3 Dimensiones del gráfico

Esta forma de pensar los gráficos nos permite repensar los distintos atributos como potenciales aliados a la hora de mostrar información multidimensional. Por ejemplo:

- `color` `color` =
- `rellenofill` =
- `forma` `shape` =
- `tamaño` `size` =

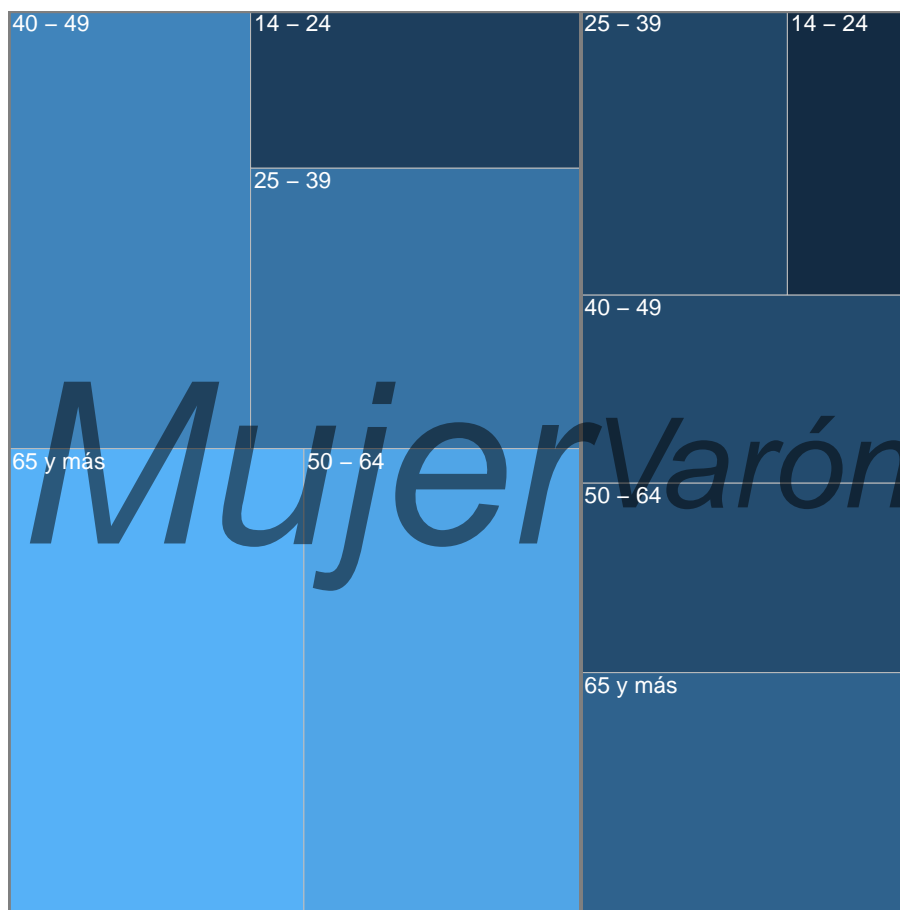
- **transparencia** `alpha =`
- Abrir un mismo gráfico según alguna variable discreta: `facet_wrap()`
- Los atributos que queremos que *mapeen* una variable, deben ir **dentro** del `aes()`, `aes(... color = variable)`
- Cuando queremos simplemente mejorar el diseño (es fijo), se asigna por fuera, o dentro de cada tipo de gráficos, `geom_col(color = 'green')`.

```
library(treemapify)
```

Trabajo doméstico no remunerado

```
trabajo_no_remunerado <- read_csv('fuentes/prom_t_simul_dom_16_sexo__annio__g_edad_lim
```

```
trabajo_no_remunerado %>%
  filter(sexo != 'TOTAL', grupo_edad != 'TOTAL') %>%
  mutate(promedio_hs_diarias = as.numeric(promedio_hs_diarias),
         sexo = case_when(sexo=='m'~'Mujer',
                          sexo=='v'~'Varón')) %>%
  ggplot(., aes(area = promedio_hs_diarias, fill = promedio_hs_diarias, label = grupo_
               subgroup = sexo)) +
  geom_treemap() +
  geom_treemap_subgroup_border() +
  geom_treemap_subgroup_text(place = "centre", grow = T, alpha = 0.5, colour =
                           "black", fontface = "italic", min.size = 0) +
  geom_treemap_text(colour = "white", place = "topleft", reflow = T)+
  theme(legend.position = 'none')
```



```
trabajo_no_remunerado %>%
  filter(sexo != 'TOTAL', grupo_edad != 'TOTAL') %>%
  mutate(promedio_hs_diarias = as.numeric(promedio_hs_diarias)) %>%
  ggplot(., aes(area=promedio_hs_diarias, fill=sexo, label=sexo))+
  geom_treemap() +
  geom_treemap_text(colour = "white", place = "topleft", reflow = T)+
  facet_wrap(., ~grupo_edad, ncol = 1)
```



4.2 Práctica Guiada

4.2.1 Graficos Ingresos - EPH

Para esta práctica utilizaremos las variables de ingresos captadas por la Encuesta Permanente de Hogares

A continuación utilizaremos los conceptos abordados, para realizar gráficos a partir de las variables de ingresos.

#Cargamos las librerías a utilizar

```
library(tidyverse) # tiene ggplot, dplyr, tidyr, y otros
library(ggthemes)  # estilos de gráficos
library(ggrepel)   # etiquetas de texto más prolijas que las de ggplot
```

```
Individual_t119 <- read.table("fuentes/usu_individual_t119.txt",
                             sep=";", dec=",", header = TRUE, fill = TRUE)
```

4.2.1.1 Boxplot de ingresos de la ocupación principal, según nivel educativo

Hacemos un procesamiento simple: Sacamos los ingresos iguales a cero y las no respuestas de nivel educativo.

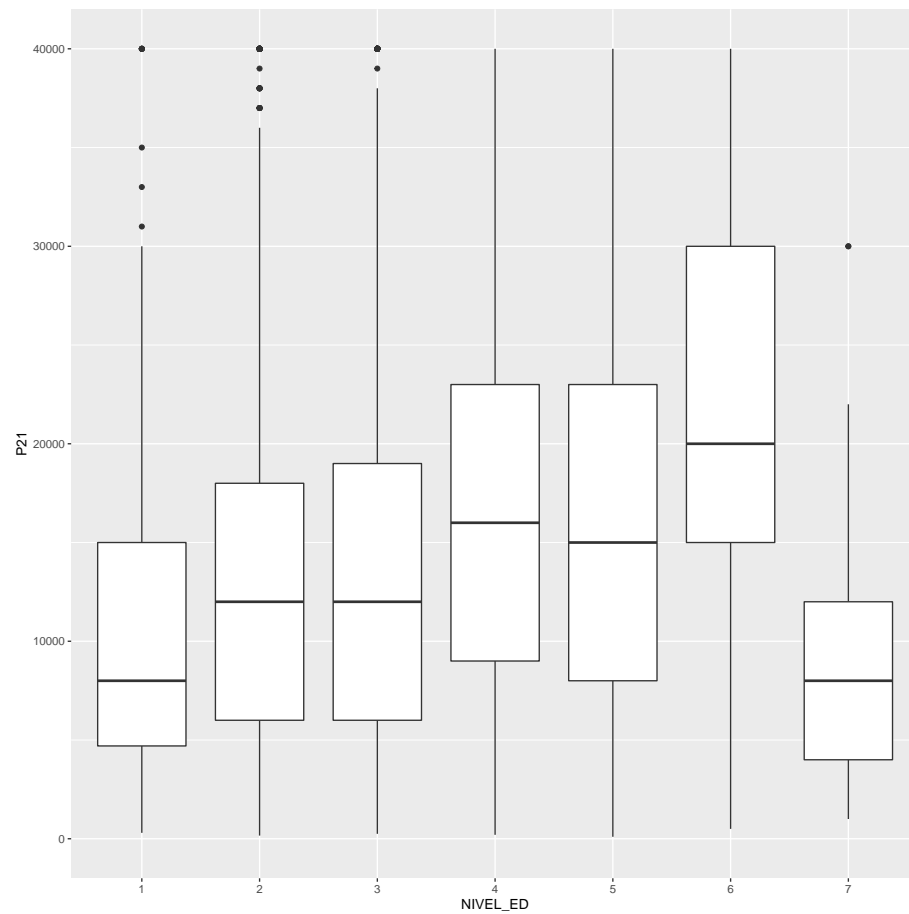
Es importante que las variables sean del tipo que conceptualmente les corresponde (el nivel educativo es una variable categórica, no continua), para que el ggplot pueda graficarlo correctamente.

```
# Las variables sexo( CH04 ) y Nivel educativo están codificadas como números, y el R las entiende como números
class(Individual_t119$NIVEL_ED)
```

```
## [1] "integer"
class(Individual_t119$CH04)
```

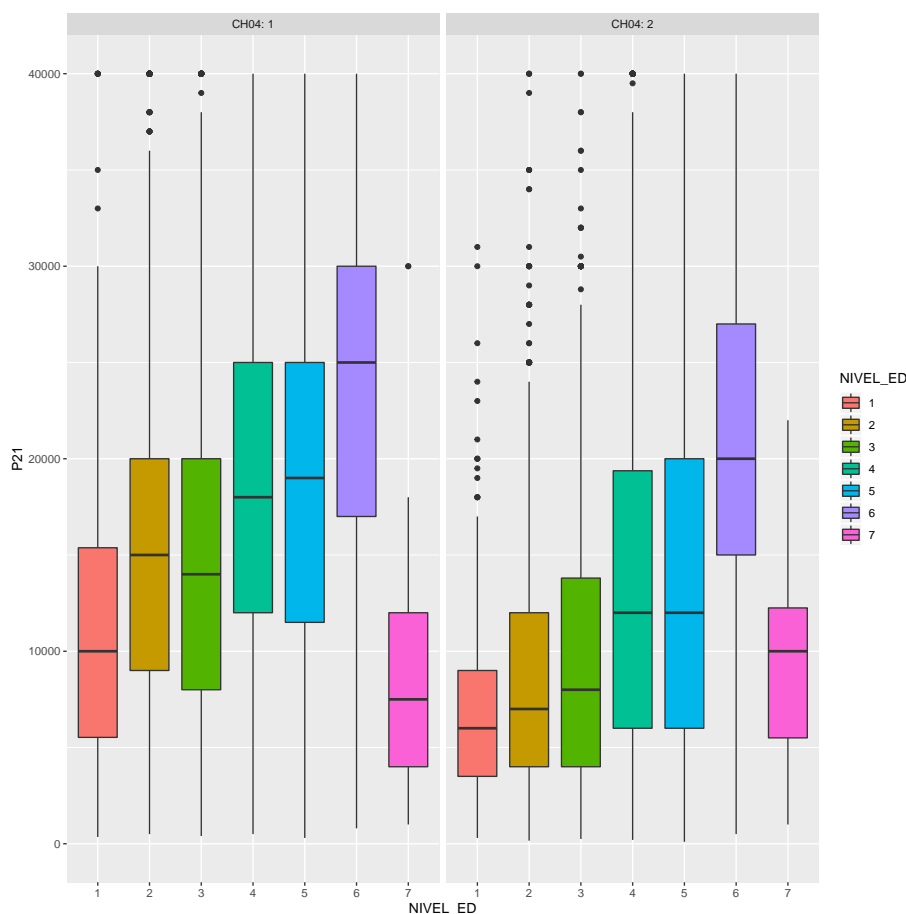
```
## [1] "integer"
ggdata <- Individual_t119 %>%
  filter(P21>0, !is.na(NIVEL_ED)) %>%
  mutate(NIVEL_ED = as.factor(NIVEL_ED),
         CH04      = as.factor(CH04))

ggplot(ggdata, aes(x = NIVEL_ED, y = P21)) +
  geom_boxplot() +
  scale_y_continuous(limits = c(0, 40000)) #Restrinjo el gráfico hasta ingresos de $40000
```



Si queremos agregar la dimensión *sexo*, podemos hacer un `facet_wrap()`

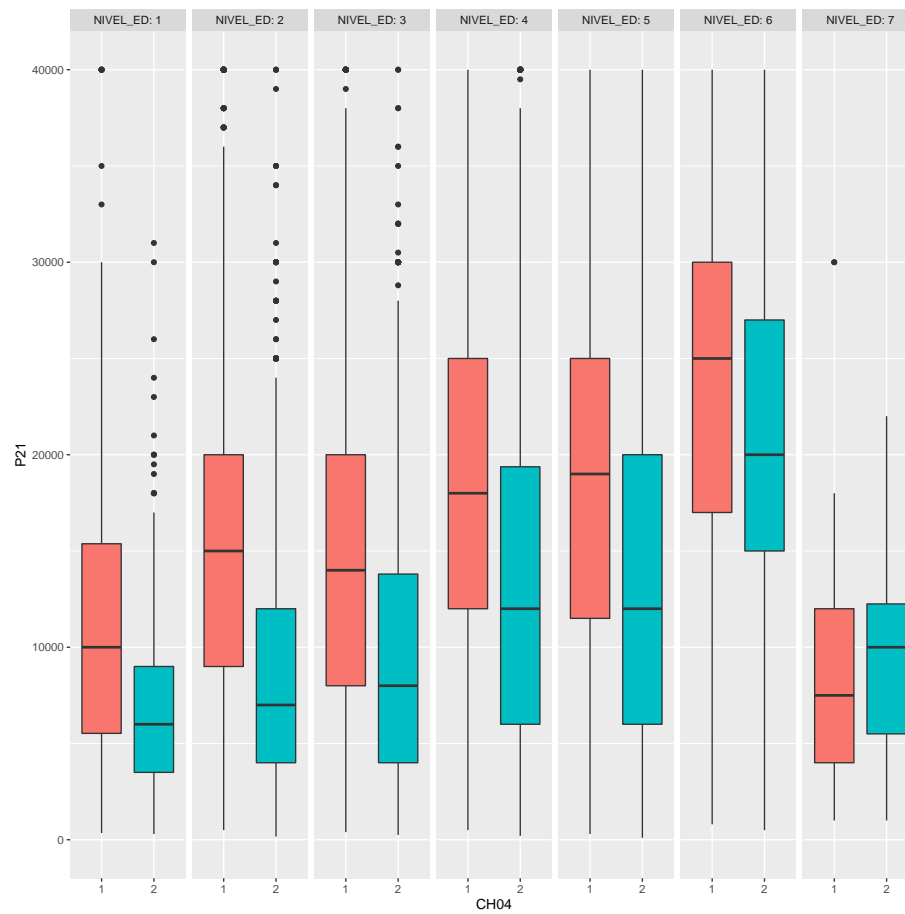
```
ggplot(ggdata, aes(x= NIVEL_ED, y = P21, group = NIVEL_ED, fill = NIVEL_ED )) +
  geom_boxplot()+
  scale_y_continuous(limits = c(0, 40000))+
  facet_wrap(~ CH04, labeller = "label_both")
```



Por la forma en que está presentado el gráfico, el foco de atención sigue puesto en las diferencias de ingresos entre niveles educativo. Simplemente se agrega un corte por la variable de sexo.

Si lo que queremos hacer es poner el foco de atención en las diferencias por sexo, simplemente basta con invertir la variable x especificada con la variable utilizada en el `facet_wrap`

```
ggplot(ggdata, aes(x= CH04, y = P21, group = CH04, fill = CH04 )) +
  geom_boxplot()+
  scale_y_continuous(limits = c(0, 40000))+
  facet_grid(~ NIVEL_ED, labeller = "label_both") +
  theme(legend.position = "none")
```

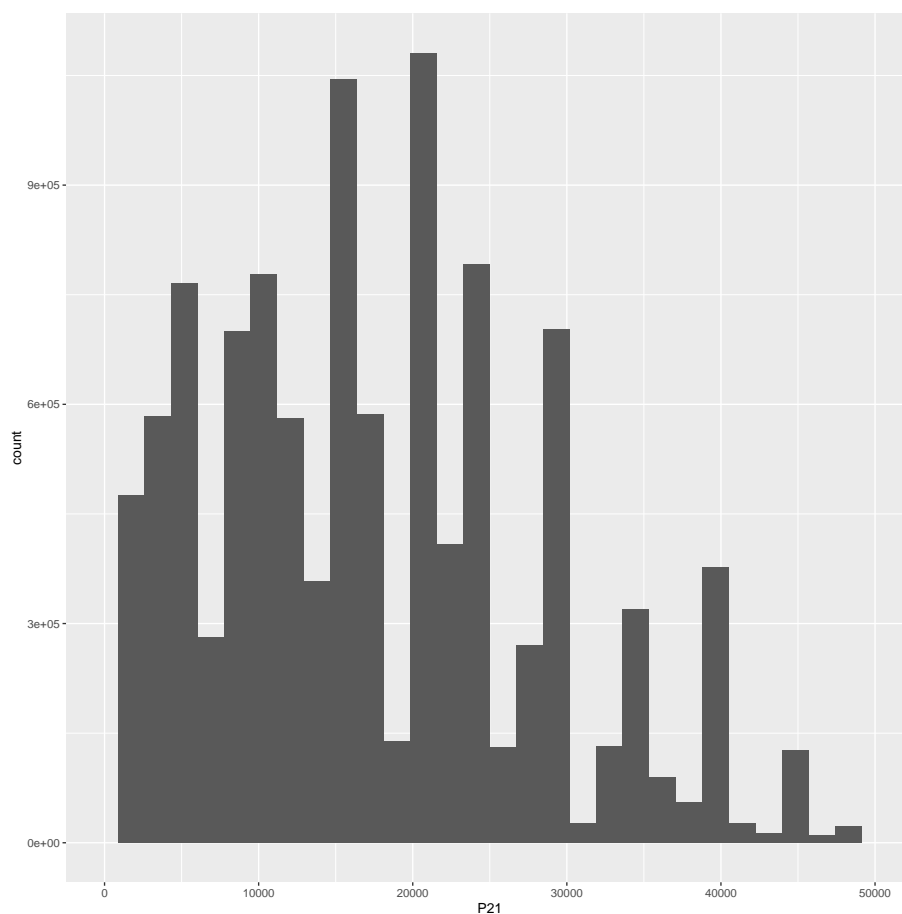


4.2.2 Histogramas

Por ejemplo, si observamos el ingreso de la ocupación principal:

```
hist_data <- Individual_t119 %>%
  filter(P21>0)

ggplot(hist_data, aes(x = P21, weights = PONDII0)) +
  geom_histogram() +
  scale_x_continuous(limits = c(0,50000))
```

En este gráfico, los posibles valores de p21 se dividen en 30 **bins** consecutivos y el gráfico muestra cuantas observaciones caen en cada uno de ellos

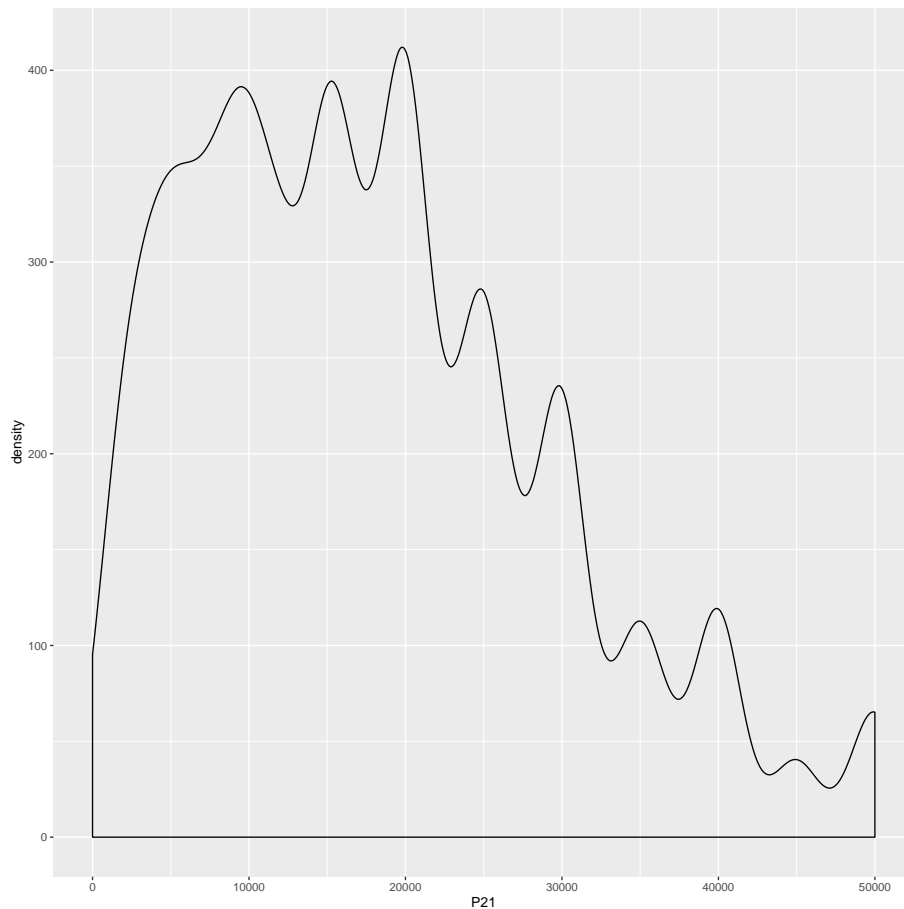
4.2.3 Kernels

La función `geom_density()` nos permite construir **kernels** de la distribución. Es particularmente útil cuando tenemos una variable continua, dado que los histogramas rompen esa sensación de continuidad.

Veamos un ejemplo sencillo con los ingresos de la ocupación principal. Luego iremos complejizándolo

```
kernel_data <- Individual_t119 %>%  
  filter(P21>0)  
  
ggplot(kernel_data, aes(x = P21, weights = PONDIIIO))+  
  geom_density()+
```

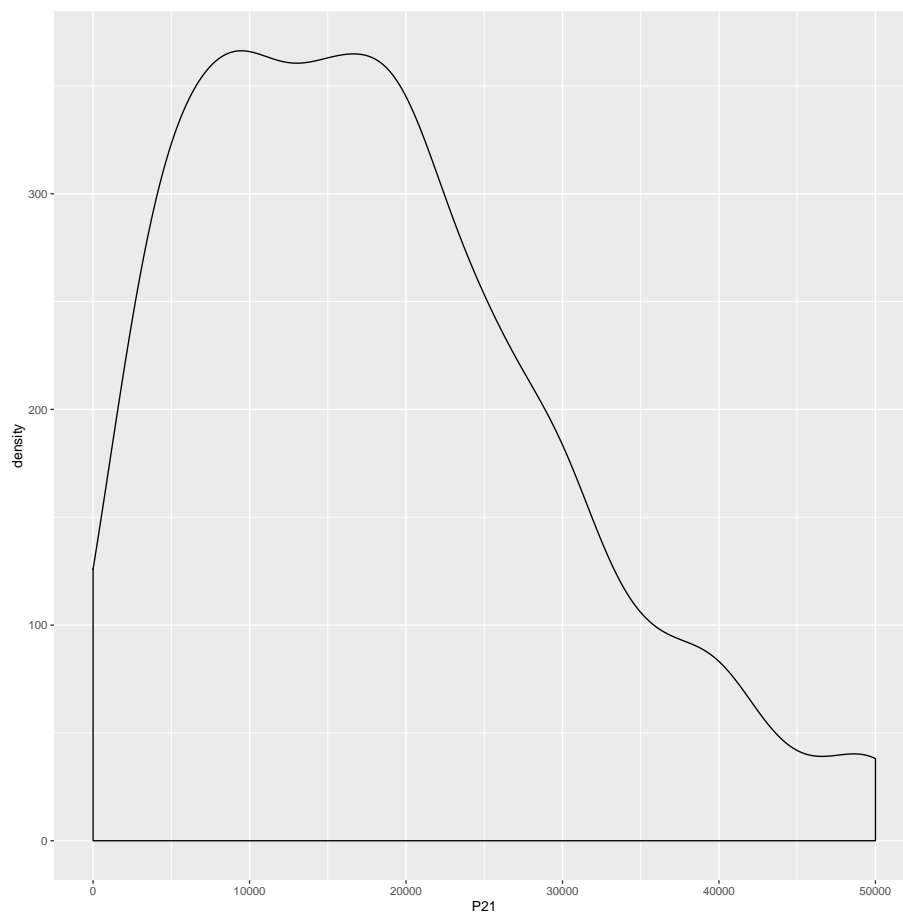
```
scale_x_continuous(limits = c(0,50000))
```



El eje y no tiene demasiada interpretabilidad en los Kernel, porque hace a la forma en que se construyen las distribuciones.

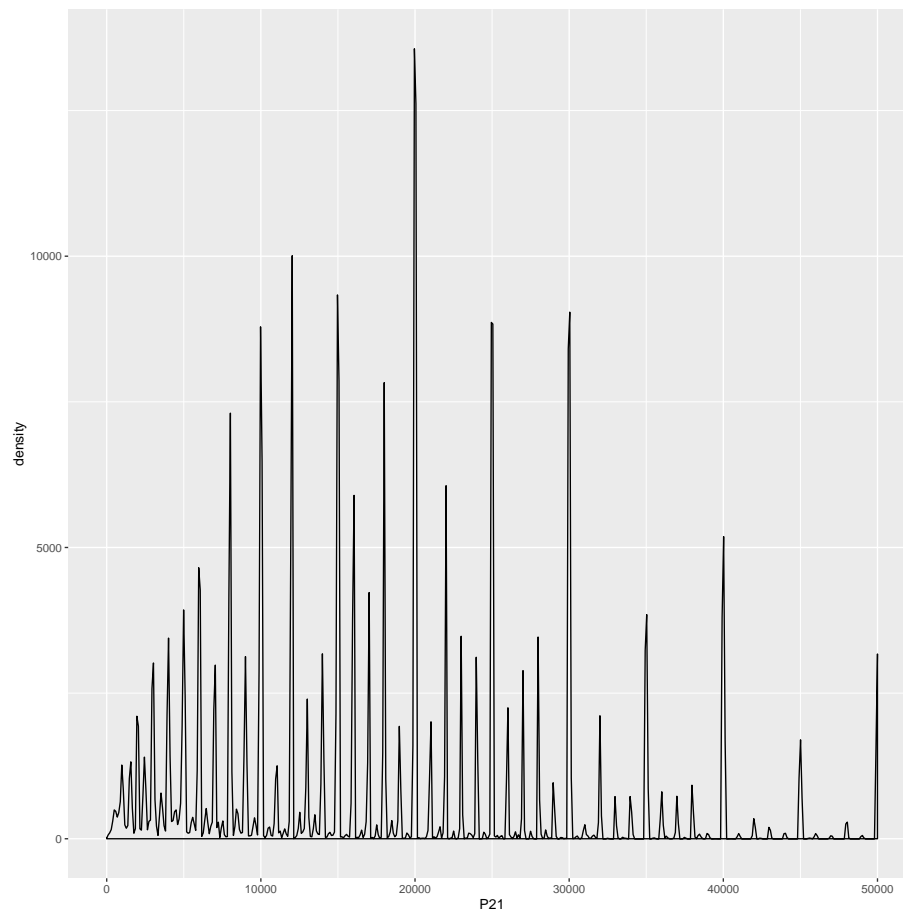
El parametro `adjust`, dentro de la función `geom_density` nos permite reducir o ampliar el rango de suavizado de la distribución. Su valor por default es 1. Veamos que sucede si lo seteamos en 2

```
ggplot(kernel_data, aes(x = P21, weights = PONDII0)) +  
  geom_density(adjust = 2) +  
  scale_x_continuous(limits = c(0,50000))
```



Como es esperable, la distribución del ingreso tiene “picos” en los valores redondos, ya que la gente suele declarar un valor aproximado al ingreso efectivo que percibe. Nadie declara ingresos de 30001. Al suavizar la serie con un kernel, eliminamos ese efecto. Si seteamos el rango para el suavizado en valores menores a 1, podemos observar estos picos.

```
ggplot(kernel_data, aes(x = P21, weights = PONDIIIO)) +  
  geom_density(adjust = 0.01) +  
  scale_x_continuous(limits = c(0, 50000))
```

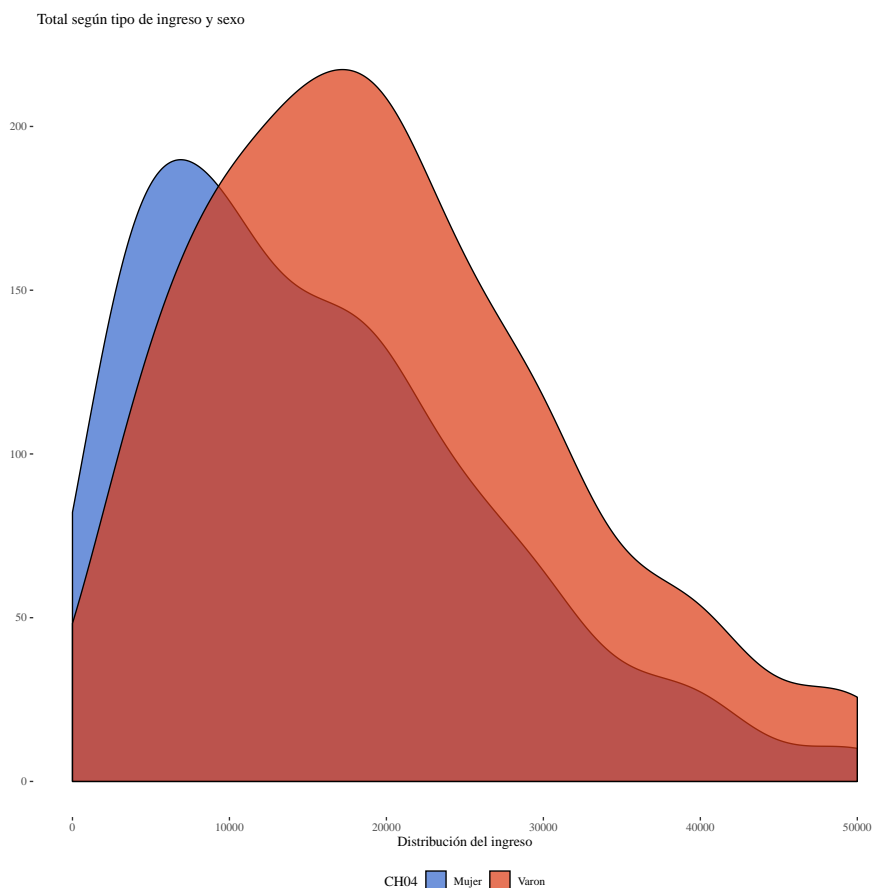


Ahora bien, como en todo grafico de R, podemos seguir agregando dimensiones para enriquecer el análisis.

```
kernel_data_2 <- kernel_data %>%
  mutate(CH04= case_when(CH04 == 1 ~ "Varon",
                        CH04 == 2 ~ "Mujer"))

ggplot(kernel_data_2, aes(x = P21,
  weights = PONDII0,
  group = CH04,
  fill = CH04)) +
  geom_density(alpha=0.7,adjust =2)+
  labs(x="Distribución del ingreso", y="",
       title=" Total según tipo de ingreso y sexo",
       caption = "Fuente: Encuesta Permanente de Hogares")+
  scale_x_continuous(limits = c(0,50000))+
  theme_tufte()+
```

```
scale_fill_gdocs()+
theme(legend.position = "bottom",
      plot.title      = element_text(size=12))
```



Fuente: Encuesta Permanente de Hogares

```
ggsave(filename = "resultados/Kernel_1.png", scale = 2)
```

Podemos agregar aún la dimensión de ingreso laboral respecto del no laboral

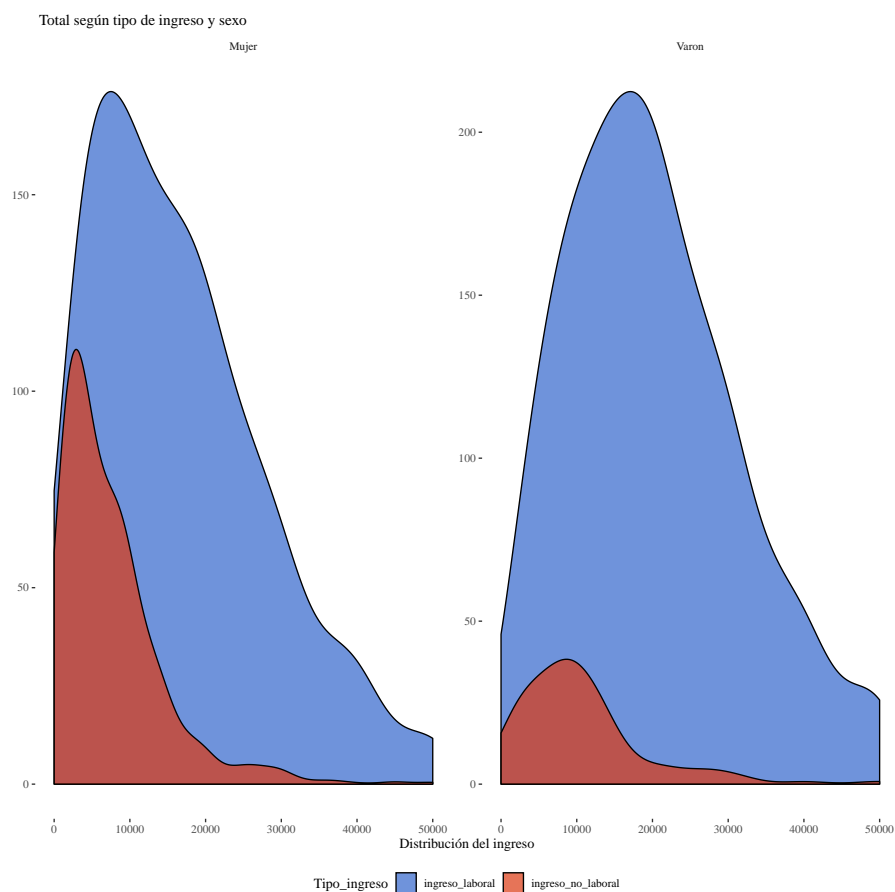
```
kernel_data_3 <- kernel_data_2 %>%
  select(REGION, P47T, T_VI, TOT_P12, P21, PONDII, CH04) %>%
  filter(!is.na(P47T), P47T > 0) %>%
  mutate(ingreso_laboral = TOT_P12 + P21,
         ingreso_no_laboral = T_VI) %>%
  gather(., key = Tipo_ingreso, Ingreso, c((ncol(.)-1):ncol(.))) %>%
  filter(Ingreso != 0) # Para este gráfico, quiero eliminar los ingresos = 0

kernel_data_3 %>%
```

```
sample_n(10)
```

```
##      REGION  P47T  T_VI TOT_P12   P21 PONDII  CH04      Tipo_ingreso
## 1         1 20000      0      0 20000  1502 Mujer ingreso_laboral
## 2        40  5500  2500      0  3000   139 Mujer ingreso_no_laboral
## 3         1 13500  1500      0 12000  2271 Mujer ingreso_laboral
## 4        40 35000      0      0 23000   226 Varon ingreso_laboral
## 5        43 22000      0      0 22000   111 Varon ingreso_laboral
## 6        44 41000      0      0 27000   117 Mujer ingreso_laboral
## 7        40 19500      0  5000 14500   133 Varon ingreso_laboral
## 8        43 18000      0  3000 15000   681 Mujer ingreso_laboral
## 9        43 23000      0      0 23000   590 Varon ingreso_laboral
## 10       1 24000 12000      0 12000   728 Mujer ingreso_laboral
##      Ingreso
## 1      20000
## 2       2500
## 3      12000
## 4      23000
## 5      22000
## 6      27000
## 7      19500
## 8      18000
## 9      23000
## 10     12000
```

```
ggplot(kernel_data_3, aes(
  x = Ingreso,
  weights = PONDII,
  group = Tipo_ingreso,
  fill = Tipo_ingreso)) +
  geom_density(alpha=0.7,adjust =2)+
  labs(x="Distribución del ingreso", y="",
       title=" Total según tipo de ingreso y sexo",
       caption = "Fuente: Encuesta Permanente de Hogares")+
  scale_x_continuous(limits = c(0,50000))+
  theme_tufte()+
  scale_fill_gdocs()+
  theme(legend.position = "bottom",
        plot.title      = element_text(size=12))+
  facet_wrap(~ CH04, scales = "free")
```



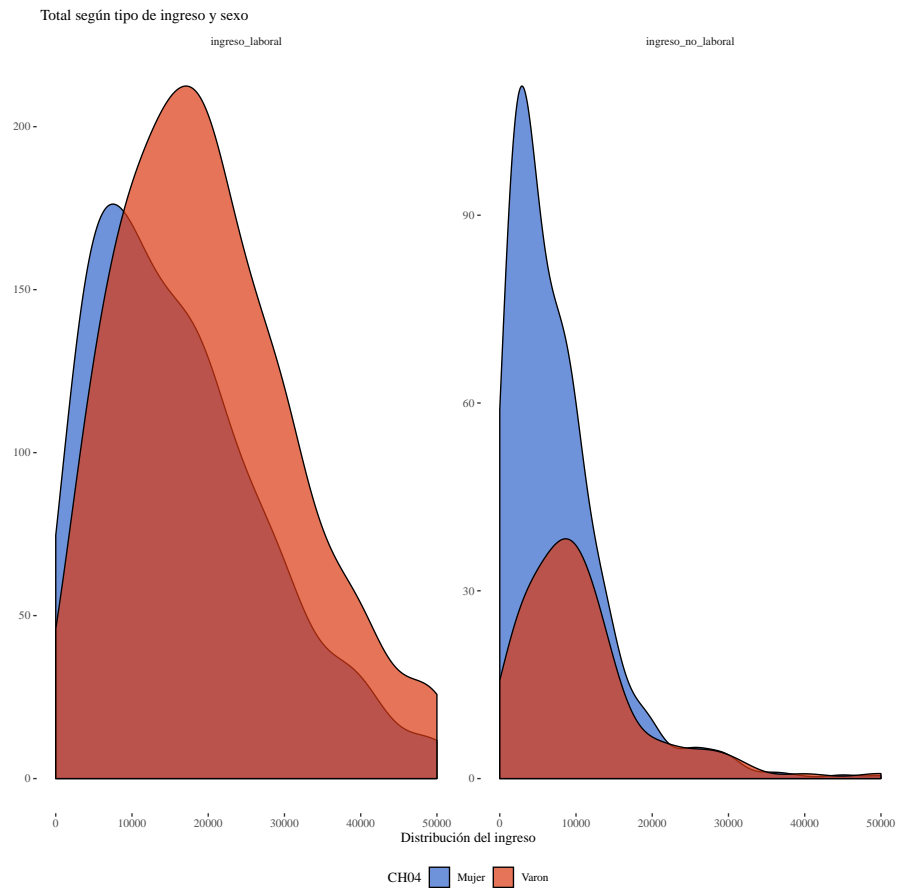
Fuente: Encuesta Permanente de Hogares

```
ggsave(filename = "resultados/Kernel_2.png", scale = 2)
```

En este tipo de gráficos, importa mucho qué variable se utiliza para *facetear* y qué variable para agrupar, ya que la construcción de la distribución es diferente.

```
ggplot(kernel_data_3, aes(
  x = Ingreso,
  weights = PONDII,
  group = CH04,
  fill = CH04)) +
  geom_density(alpha=0.7, adjust =2)+
  labs(x="Distribución del ingreso", y="",
       title=" Total según tipo de ingreso y sexo",
       caption = "Fuente: Encuesta Permanente de Hogares")+
  scale_x_continuous(limits = c(0,50000))+
  theme_tufte()+
  scale_fill_gdocs()+
```

```
theme(legend.position = "bottom",
      plot.title       = element_text(size=12)) +
facet_wrap(~Tipo_ingreso, scales = "free")
```



Fuente: Encuesta Permanente de Hogares

```
ggsave(filename = "resultados/Kernel_3.png", scale = 2)
```


Chapter 5

Documentación en R

5.1 Explicación

5.1.1 R Markdown: Introducción

Los archivos R Markdown nos permiten combinar código, resultados y texto. El objetivo de esta clase es aprender a trabajar bajo dicho entorno para facilitar 3 aplicaciones:

- Documentar el trabajo que realizamos, incluyendo comentarios sobre los procedimientos.
- Compartir código y resultados con gente que también trabaja en R.
- Compartir resultados con gente que no trabaja en R, y simplemente necesita enfocarse en las conclusiones.

Las presentes notas de clase están basadas en el libro R4DS y las cheatsheets. También se recomienda el libro R Markdown: The Definitive Guide.

5.1.2 Requisitos

Necesitamos instalar y cargar el paquete **rmarkdown**, pero por lo general no hace falta hacerlo explícitamente porque RStudio realiza esto automáticamente cuando es necesario.

5.1.3 Markdown básico

Se trata de un archivo de extensión **.Rmd**. Contiene en su estructura tres tipos importantes de contenido:

- Un encabezado YAML (“Yet another markup language”) rodeado de - - -

```
title: "El título de nuestro informe"
```

```
date: Septiembre 2019
output: html_document
---
```

- Bloques de código de R rodeado de “`”.

```
`` `{r echo=TRUE, message=FALSE, warning=FALSE}
library(tidyverse)

tabla <- datos %>%
  filter(condicion.logica == TRUE) %>%
  mutate(variable = variable * 100)
`` `
```

- Texto con formato (que veremos en unos minutos)

Cuando abrimos un archivo **.Rmd**, obtenemos una interfaz de notebook donde el código y el output se encuentran intercalados (en lugar de aparecer el output sólo en la consola, panel de Plots y/o modificaciones en el entorno de trabajo).

Los bloques de código se pueden ejecutar haciendo click en el ícono **ejecutar** (el botón de *Play* en la parte superior/derecha del bloque), o presionando **Cmd/Ctrl + Shift + Enter**. RStudio ejecuta el código y muestra los resultados incrustados en el código.

Para producir un reporte completo que contenga todo el texto, código y resultados, podemos clicar en **Knit** o presionar **Cmd/Ctrl + Shift + K**. Esto mostrará el reporte en el panel *Viewer* y creará un archivo HTML independiente que podremos compartir con otros.

5.1.4 Formateo de texto

La prosa en los archivos **.Rmd** está escrita en Markdown, una colección simple de convenciones para dar formato a archivos de texto plano. Markdown está diseñado para ser fácil de leer y fácil de escribir, siendo también muy fácil de aprender. Del *Cheatsheet*:

sintaxis

```

Texto plano
Termina línea con dos espacios para nuevo párrafo.
*cursivo* y _cursivo_
**negrita** y __negrita__
superíndice^2^
~~tachado~~
[eslabón](www.rstudio.com)

# Encabezado 1

## Encabezado 2

### Encabezado 3

#### Encabezado 4

##### Encabezado 5

##### Encabezado 6

raya em: --
raya em: ---
elipsis: ...
ecuación en línea: $A = \pi * r^{2}$

imagen: 

regla horizontal (o nueva diapositiva):

***

> cita en bloque

* lista sin orden
* elemento 2
  + sub-elemento 1
  + sub-elemento 2

1. lista ordenada
2. elemento 2
  + sub-elemento 1
  + sub-elemento 2

Encabezado Tabla | Segundo Encabezado
-----|-----
Celda de tabla | Celda 2
Celda 3 | Celda 4

```

resulta en

Texto plano
Termina línea con dos espacios para nuevo párrafo.
cursivo y cursivo
negrita y negrita
superíndice²
tachado
eslabón

Encabezado 1

Encabezado 2


Encabezado 3

Encabezado 4

Encabezado 5

Encabezado 6

raya em: –
raya em: —
elipsis: ...
ecuación en línea: $A = \pi * r^2$

imagen: 

regla horizontal (o nueva diapositiva):

cita en bloque

- lista sin orden
 - elemento 2
 - sub-elemento 1
 - sub-elemento 2
1. lista ordenada
 2. elemento 2
 - sub-elemento 1
 - sub-elemento 2

Encabezado Tabla	Segundo Encabezado
Celda de tabla	Celda 2
Celda 3	Celda 4

5.1.5 Bloques de código

Como ya mencionamos, para ejecutar código dentro de un documento R Markdown, necesitamos insertar un bloque (*Chunk*). Hay tres maneras para hacerlo:

- El atajo de teclado **Cmd/Ctrl + Alt + I**
- El icono “Insertar” en la barra de edición (**Insert > R**)
- Tipear manualmente los delimitadores de bloque ````${r}```` y `````.

Obviamente, recomendamos usar el atajo de teclado porque, a largo plazo, ahorra mucho tiempo. El código se puede seguir corriendo con **Cmd/Ctrl + Enter** línea a línea. Sin embargo, los bloques de código tienen otro atajo de teclado: **Cmd/Ctrl + Shift + Enter**, que ejecuta todo el código en el bloque.

Un bloque debería ser relativamente autónomo, y enfocado alrededor de una sola tarea. Las siguientes secciones describen el encabezado de bloque que consiste

en ````{r}`, seguido por un nombre opcional para el bloque, seguido entonces por **opciones separadas por comas**, y concluyendo con `}`. Inmediatamente después sigue tu código de R el bloque y el fin del bloque se indica con un ````` final.

Hay un nombre de bloque que tiene comportamiento especial: **setup**. Cuando te encuentras en modo notebook, el bloque llamado `setup` se ejecutará automáticamente una vez, antes de ejecutar cualquier otro código.

La salida de los bloques puede personalizarse con **options**, argumentos suministrados al encabezado del bloque.

5.1.6 Opciones en los bloques de código

La salida de los bloques puede personalizarse con **options**, argumentos suministrados en el encabezado del bloque. Knitr provee casi 60 opciones para que puedas usar para personalizar tus bloques de código, la lista completa puede verse en <http://yihui.name/knitr/options/>.

- **eval = FALSE** evita que código sea evaluado. (Y obviamente si el código no es ejecutado no se generaran resultados). Esto es útil para mostrar códigos de ejemplo, o para deshabilitar un gran bloque de código sin comentar cada línea.
- **include = FALSE** ejecuta el código, pero no muestra el código o los resultados en el documento final. Usa esto para que código de configuración que no quieres que abarrote tu reporte.
- **echo = FALSE** evita que se vea el código, pero no los resultados en el archivo final. Utiliza esto cuando quieres escribir reportes enfocados a personas que no quieren ver el código subyacente de R.
- **message = FALSE** o **warning = FALSE** evita que aparezcan mensajes o advertencias en el archivo final.
- **results = 'hide'** oculta el output impreso; **fig.show = 'hide'** oculta gráficos.
- **error = TRUE** causa que el render continúe incluso si el código devuelve un error. Esto es algo que raramente quieres incluir en la versión final de tu reporte.

Opción	Ejecuta	Muestra	Output	Gráficos	Mensajes	Advertencias
<code>eval = FALSE</code>	-		-	-	-	-
<code>include = FALSE</code>		-	-	-	-	-
<code>echo = FALSE</code>		-				
<code>results = "hide"</code>			-			
<code>fig.show = "hide"</code>				-		
<code>message = FALSE</code>					-	
<code>warning = FALSE</code>						-

formato chunk.bb

Contamos con algunas de estas opciones en el menú de **Configuración** en la parte superior-derecha del *Chunk* de código.

5.1.7 Tablas

Por defecto, las tablas se imprimen tal como salen como en la consola. Si queremos que los datos tengan un formato adicional puedes usar la función `knitr::kable()`. Aún más, recomendamos mirar los paquetes `kableExtra` y `formattable`.

Para una mayor personalización, se pueden considerar también los paquetes `xtable`, `stargazer`, `pander`, `tables`, y `ascii`. Cada uno provee un set de herramientas para generar tablas con formato a partir código de R.

5.1.8 Opciones globales

Algunas de las opciones default que tienen los bloques de código pueden no ajustarse a tus necesidades. Podemos setear cambios incluyendo `knitr::opts_chunk$set()` en un bloque de código. Por ejemplo:

```
knitr::opts_chunk$set(echo = FALSE)
```

Ocultará el código por defecto, así que sólo mostrará los bloques que deliberadamente elegimos mostrar con `echo = TRUE`.

5.1.9 Código en la línea

Otra forma de incluir código R en un documento R Markdown es insertarlo directamente en el texto, encerrando entre ```r código```. Esto puede ser muy útil si queremos mencionar propiedades o atributos de los datos o resultados en el texto.

Cuando insertamos números en el texto, `format()` nos va a ser de mucha ayuda. Esto permite establecer el número de dígitos para que no imprimas con un grado

rídículo de precisión, y una `big.mark` para hacer que los números sean mas fáciles de leer. Por ejemplo, en una función de ayuda:

```
formato <- function(x){
  format(x, digits = 2, big.mark = ".", decimal.mark = ",")
}
formato(3452345)
```

```
## [1] "3.452.345"
```

```
formato(.12358124331)
```

```
## [1] "0,12"
```

5.1.10 Formatos

Hasta ahora vimos R Markdown para producir documentos HTML:

```
---
title: "Clase"
output: html_document
---
```

Para sobrescribir los parámetros predeterminados se necesita usar un campo de `output` extendido. Por ejemplo, si queremos generar un `html_document` con una tabla de contenido flotante, usamos:

```
---
title: "Clase"
output:
  html_document:
    toc: true
    toc_float: true
---
```

Para los `html_document` otra opción es hacer que los fragmentos de código estén escondidos por defecto, pero visibles con un *click*:

```
---
title: "Clase"
output:
  html_document:
    code_folding: hide
---
```

5.1.11 Otros formatos

Hay todo un número de variaciones básicas para generar diferentes tipos de documentos:

- `pdf_document` crea un PDF con LaTeX (un sistema de código abierto de composición de textos), que necesitarás instalar. RStudio te notificará si no lo tienes.
- `word_document` para documentos de Microsoft Word (.docx).
- `odt_document` para documentos de texto OpenDocument (.odt).

y más!

5.1.12 Notebooks

Un notebook, `html_notebook` (“cuaderno” en español), es una variación de un `html_document`. Las salidas de los dos documentos son muy similares, pero tienen propósitos distintos. Un `html_document` está enfocado en la comunicación con los encargados de la toma de decisiones, mientras que un notebook está enfocado en colaborar con otros científicos de datos. Estos propósitos diferentes llevan a usar la salida HTML de diferentes maneras. Ambas salidas HTML contendrán la salida renderizada, pero **el notebook también contendrá el código fuente completo**. Esto significa que podemos usar el archivo `.nb.html` generado por el notebook de dos maneras:

- Podemos verlo en un navegador web, y ver la salida generada. A diferencia del `html_document`, esta renderización siempre incluye una copia incrustada del código fuente que lo generó.
- Podemos editarlo en RStudio. Cuando abramos un archivo `.nb.html`, RStudio automáticamente recreará el archivo `.Rmd` que lo creó.

5.1.13 Publicar

Desde RStudio tenemos la posibilidad de publicar nuestros Markdown en RPubS de forma gratuita, desde el botón **Publish document**. Todo lo que subamos a nuestra cuenta de RPubS será público.

5.1.14 FlexDashboard

Los dashboards (“tableros de control” en español) son una forma útil de comunicar grandes cantidades de información de forma visual y rápida. Flexdashboard hace que sea particularmente fácil crear dashboards usando R Markdown y proporciona una convención de cómo los encabezados afectan el diseño:

- Cada encabezado de Nivel 1 (#) comienza una nueva página en el dashboard.
- Cada encabezado de Nivel 2 (##) comienza una nueva columna.
- Cada encabezado de Nivel 3 (###) comienza una nueva fila.

Flexdashboard también proporciona herramientas simples para crear barras laterales, tabuladores, cuadros de valores y medidores. Podemos obtener más

información (en inglés) acerca de Flexdashboard en <http://rmarkdown.rstudio.com/flexdashboard/>.

5.2 Práctica Guiada

5.2.1 Introducción

El objetivo de esta clase es comenzar a trabajar utilizando el formato RNotebook para realizar reportes compilados directamente en RStudio, de forma tal que nuestro trabajo pueda quedar documentado y ser fácilmente compartido con otras personas.

Para esto utilizaremos un dataframe del paquete **datos**. Debemos instalarlo en caso de contar con el mismo, y luego cargarlo con la función `library()`. En particular, utilizaremos los datos de **encuesta**, que consiste en una muestra de variables categóricas de la Encuesta Social General de EE.UU.

```
encuesta <- datos::encuesta
```

El dataframe cuenta con 21.483 observaciones y 9 variables.

5.2.2 Explorando los datos

La muestra refiere a información obtenida entre 2000 y 2014. Se presentan datos sobre estado civil, raza, ingresos, partido político de pertenencia, religión, y cantidad de horas dedicadas a mirar televisión, para personas de entre 18 y 89 años.

5.2.2.1 Religión

En primer lugar, nos interesa ver la distribución en términos de la **religión** de las personas, haciendo énfasis en aquellas más populares.

```
## # A tibble: 15 x 2
##   religion          cantidad
##   <fct>             <int>
## 1 Protestante      10846
## 2 Católica         5124
## 3 Ninguna          3523
## 4 Cristiana        689
## 5 Judía            388
## 6 Otra             224
## 7 Budismo          147
## 8 Inter o no confesional 109
## 9 Musulmana/Islam    104
## 10 Cristiana ortodoxa    95
## 11 Sin respuesta       93
## 12 Hinduismo          71
```

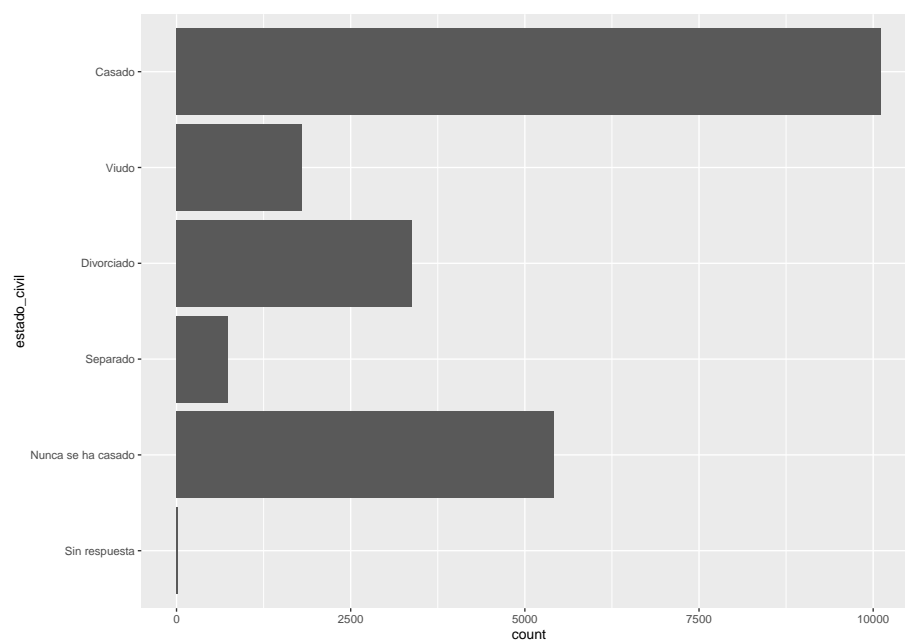


```
## 13 Otra religión oriental      32
## 14 Nativa americana           23
## 15 No sabe                     15
```

Puede verse que aquella que cuenta con más seguidores es la religión Protestante, con 10.846 fieles.

5.2.2.2 Estado Civil

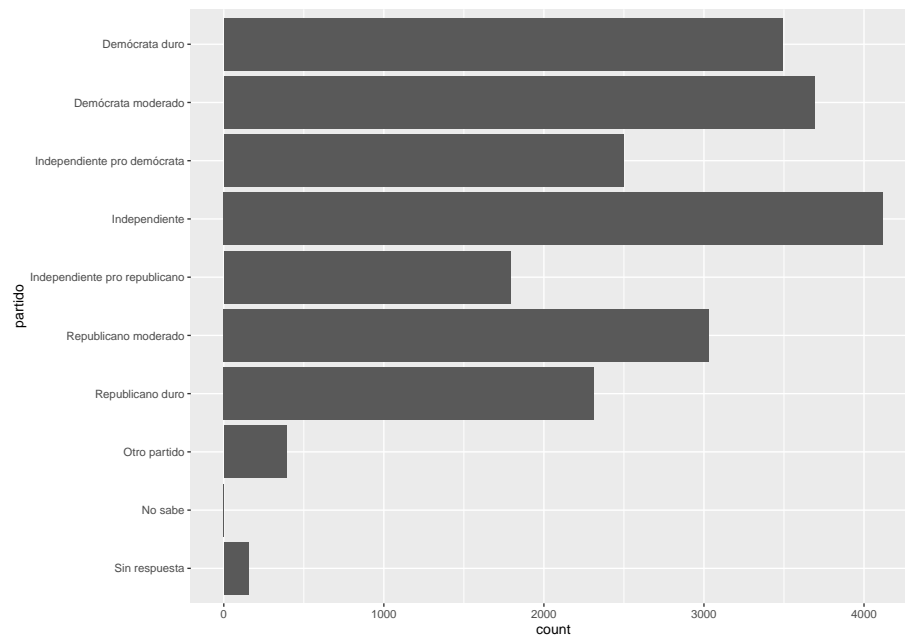
También podemos visualizar la distribución del **estado civil** de las personas.



Vemos que la mayoría de las personas (10.117 en total) responde “Casado” cuando se indaga sobre su estado civil.

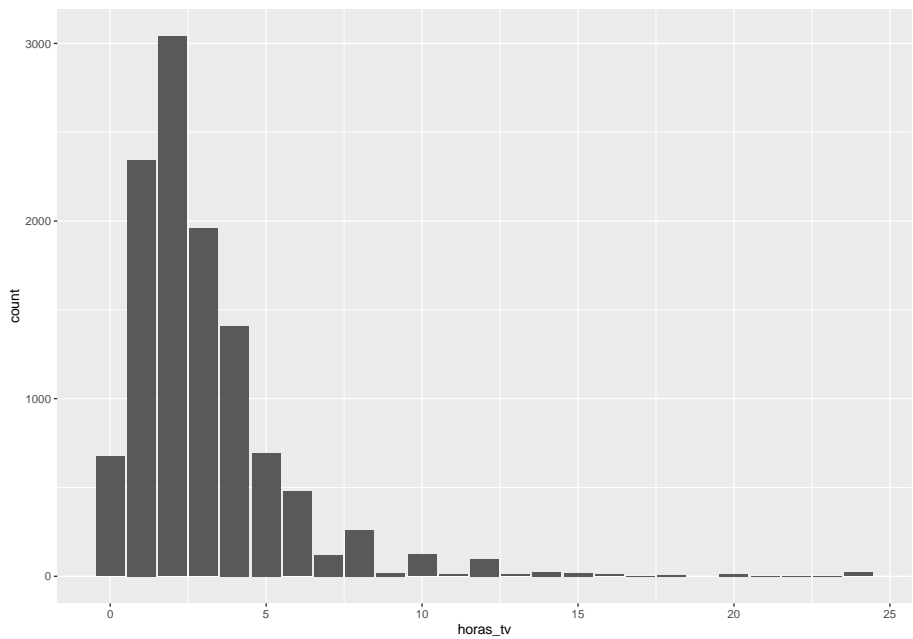
5.2.2.3 Partido político

La encuesta también nos permite conocer sobre las pertenencias partidarias de los individuos.



5.2.2.4 Horas de exposición a la televisión

A partir de los datos, sabemos que los individuos miran la televisión, en promedio, durante 3 horas por día. A continuación se presenta toda la distribución de la variable:



Chapter 6

Shiny apps

En este módulo veremos la utilización de Shinyapss para elaborar reportes reactivos. Entre otras cosas, veremos:

- Shiny como reportes dinámicos
- Su utilidad para el análisis exploratorio
- Lógica de servidor- interfaz de usuario
- Inputs- Outputs, funciones reactivas, widgets.

6.1 Explicación

6.1.1 ¿Qué es un shiny app?

Shiny es un paquete de R que facilita la creación de aplicaciones web interactivas directamente desde R. Permite a quienes no son versados en diseño web construir rápidamente una página reactiva para explorar la información.

6.1.2 Galería de ejemplos

Veamos algunos ejemplos de la página:

<https://shiny.rstudio.com/gallery/>

6.1.3 Componentes fundamentales de un Shiny app

Un Shiny App tiene dos componentes

- Interfaz de Usuario (*UI*): Contiene los *dispositivos* para recibir el **input** del usuario y mostrar los **outputs**
- Server: Recibe los **inputs** del UI y con ellos genera los **outputs**

6.1.3.1 Ejemplo 1. Old Faithful Geyser

Abrir el archivo ejemplo_1/app.R

Comencemos con el ejemplo más básico. Cuando creamos un nuevo shiny, nos genera este ejemplo como *template*.

- Primero cargamos la librería.

```
library(shiny)
```

- Luego definimos la interfaz de usuario.

Elementos del ui:

- **fluidPage**: La función con que definimos el *layout* general
 - **titlePanel**: Para definir el título
 - **sidebarLayout**: Definimos que el diseño de la app va a ser con una barra lateral y un panel central
 - * **sidebarPanel**: Dentro del *sidebarPanel* definimos los elementos que van en la barra lateral
 - **sliderInput**: Definimos que el tipo de input se es *slider*, y sus parámetros
 - * **mainPanel**: Dentro del *mainPanel* definimos los elementos que van en el panel central
 - **plotOutput**: con esta función indicamos que el output es un gráfico

```
ui <- fluidPage(
  titlePanel("Old Faithful Geyser Data"),
  sidebarLayout(
    sidebarPanel(
      sliderInput(inputId = "bins",
        label = "Number of bins:",
        min = 1,
        max = 50,
        value = 30)
    ),
    mainPanel(
      plotOutput("distPlot")
    )
  )
)
```

```
)
)
```

Elementos del server

- **input:** Es una lista de elementos que recibimos del *ui*. en este caso sólo contiene `bins` (el `inputId`)
- **output:** Es una lista que generamos dentro del server. En este caso definimos el elemento `distPlot`
- **renderPlot:** Es una *función reactiva*, que observa cada vez que *cambia el input* y vuelve a generar el output. Noten que lo que hace es envolver una porción del código entre llaves.

```
server <- function(input, output) {

  output$distPlot <- renderPlot({
    x <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })
}

shinyApp(ui = ui, server = server)
```

6.2 Práctica Guiada

6.2.1 Ejemplo 2. Gapminder

Construyamos nuestro propio ejemplo con los datos de Gapminder. Para eso, vamos a ver que la manera más cómoda de escribir una shiny app no es en el orden en que aparece el código final.

Al código hay que comerlo de a pedacitos.

1. Pensamos qué queremos mostrar
2. Escribimos código *estático* para un caso particular.
3. Pensamos qué partes queremos generalizar.
4. Armamos una función que tome como parámetros aquello que generalizamos
5. Armamos un shiny estático que nos muestre el resultado de la función con parámetros fijos
6. Agregamos los inputs en el ui
7. reemplazamos los parámetros fijos por los de input en el server
8. Agregamos texto y otros elementos ‘cosméticos’

A cada paso vamos armando un código que no falle. De esta forma es más fácil detectar los errores.


```
library(tidyverse)
library(gapminder)

gapminder <- gapminder
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
```

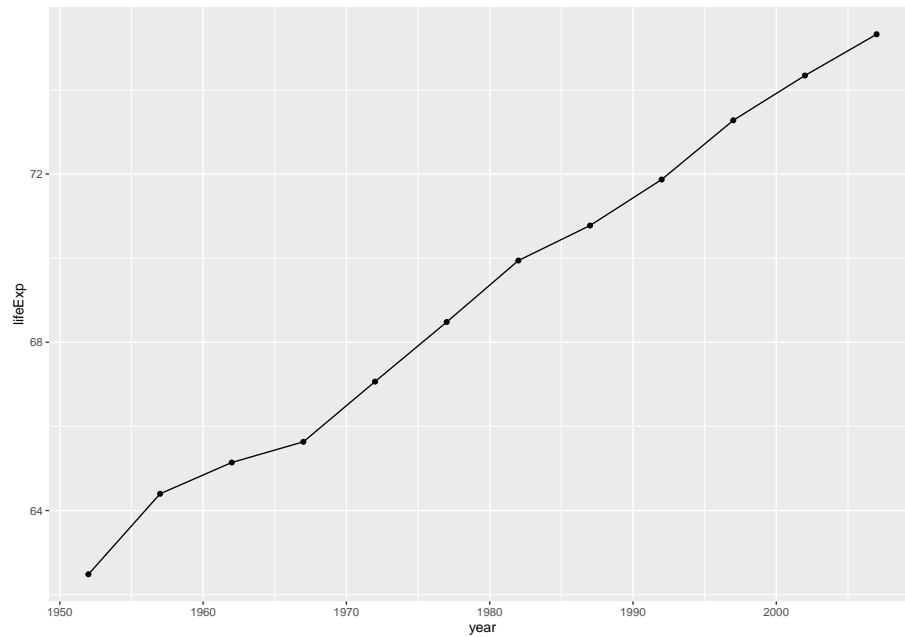
6.2.1.1 1. qué queremos mostrar

- tenemos tres variables que podrían ser agrupadoras: País, continente y año
- y tres variables que puede ser interesante representar: Esperanza de vida, población y PBI per cápita

Podríamos mostrar por ejemplo la serie de tiempo de algún país para alguna variable

6.2.1.2 2. código estático para un caso particular

```
gapminder %>%
  filter(country == 'Argentina') %>%
  ggplot(aes(year, lifeExp))+
  geom_line()+
  geom_point()
```



6.2.1.3 3. partes que queremos generalizar.

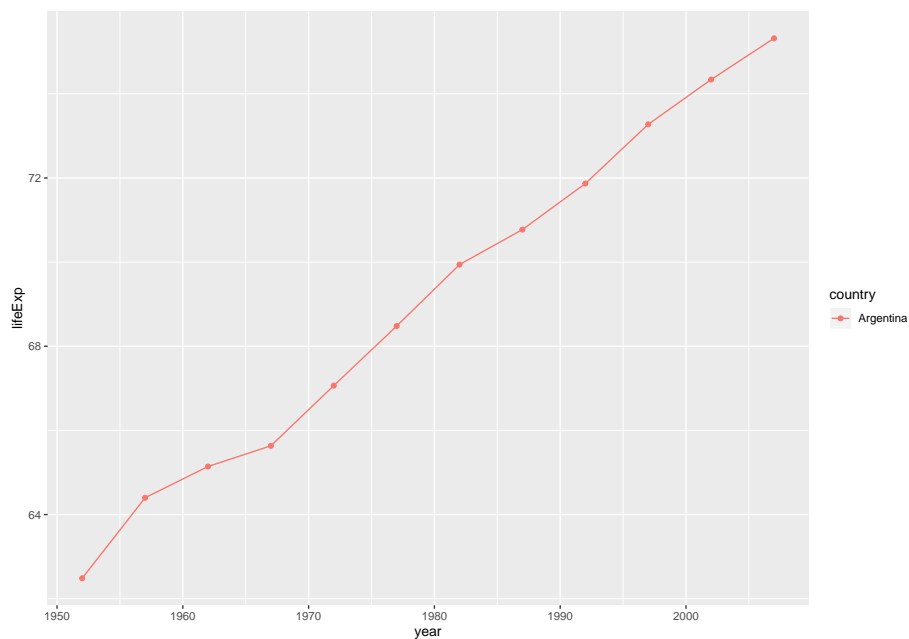
- El gráfico podría ser para cualquier país (o para un conjunto de países!)
- podríamos elegir qué variable ver

6.2.1.4 4. función que tome como parámetros aquello que general-

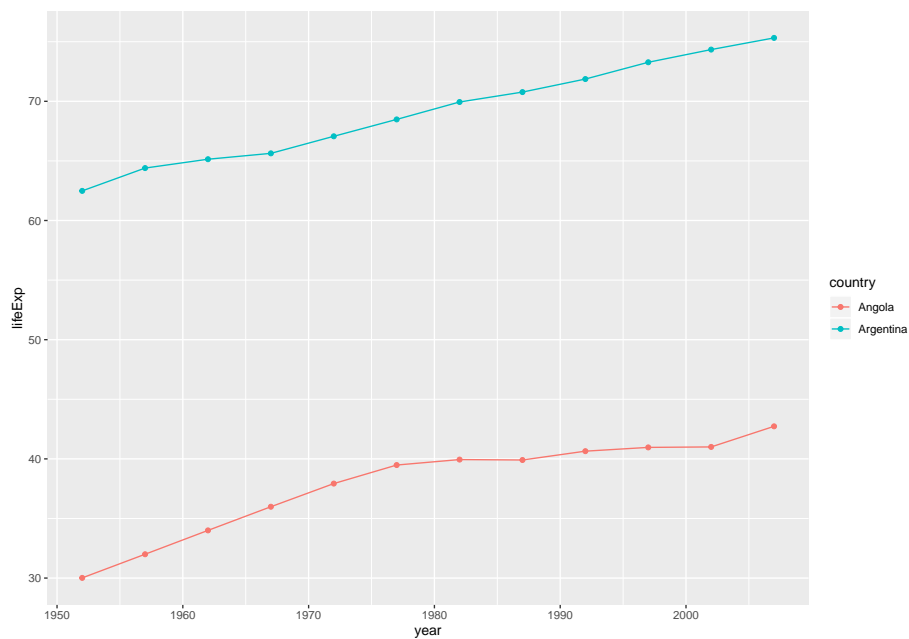
```
graficar <- function(pais, variable){

  gapminder %>%
  filter(country %in% pais) %>% ## reemplaze el == por %in% para que me reciba más de un país
  ggplot(aes_string("year", variable, color= "country"))+ ## Le cambio aes por aes_string
  geom_line()+
  geom_point()
}

graficar(pais = "Argentina", variable = "lifeExp")
```



```
graficar(pais = c("Argentina","Angola"), variable = "lifeExp")
```



6.2.1.5 5. shiny estático con parámetros fijos

ver ejemplo_2_a

6. Agregamos los inputs en el ui

Necesitamos agregar dos inputs: País y variable.

Para opciones podemos usar selectize

```
selectizeInput(inputId, label, choices, selected = NULL, multiple
= FALSE,                      options = NULL)
```

Podemos crear la lista de opciones de países automáticamente

```
unique(gapminder$country)[1:10]
```

```
## [1] Afghanistan Albania    Algeria    Angola    Argentina
## [6] Australia  Austria    Bahrain    Bangladesh Belgium
## 142 Levels: Afghanistan Albania Algeria Angola Argentina ... Zimbabwe
```

ver ejemplo_2_b

6.2.1.6 7. reemplazamos los parámetros fijos por los de input en el server

ver ejemplo_2_c

6.2.1.7 8. Tuneamos a discreción

Una vez que tenemos un shiny funcionando como queríamos, podemos agregar tags y texto para agregar explicaciones y emprolijar los resultados.

```
# Headers
# shiny::tags$h1('Nivel 1')
# shiny::tags$h2('Nivel 2')
# shiny::tags$h3('Nivel 3')
# shiny::tags$h4('Nivel 4')
# shiny::tags$h5('Nivel 5')
# shiny::tags$h6('Nivel 6')

shiny::br() # espacio en blanco

shiny::hr() # linea horizontal

shiny::helpText('texto para ayudas')
```

texto para ayudas

6.2.1.8 Múltiples pestañas

También puede ocurrir que queramos mostrar varios resultados en un mismo shiny. En nuestro ejemplo, podríamos querer mostrar una tabla con los datos.

- Para eso podemos usar `tabsetPanel` en el ui
- Imaginemos que queremos tener dos tabs: Una con el gráfico, y otra con una tabla de resultados:

Entonces, en el shiny debemos agregar:

```
mainPanel(  
  tabsetPanel(type = "tabs",  
    tabPanel("Gráfico", plotOutput("grafico")),  
    tabPanel("Tabla", tableOutput("tabla"))  
  )  
)
```

Mientras que en el server debemos generar un nuevo resultado, llamado *tabla* con los datos

```
output$tabla <- renderTable({  
  gapminder %>%  
    filter(country %in% input$inputPais)  
})
```

ver ejemplo_2_d

Chapter 7

Probabilidad y Estadística

Esta clase es un repaso de los rudimentos de probabilidad y estadística. El objetivo es obtener las herramientas básicas para la interpretación de resultados estadísticos.

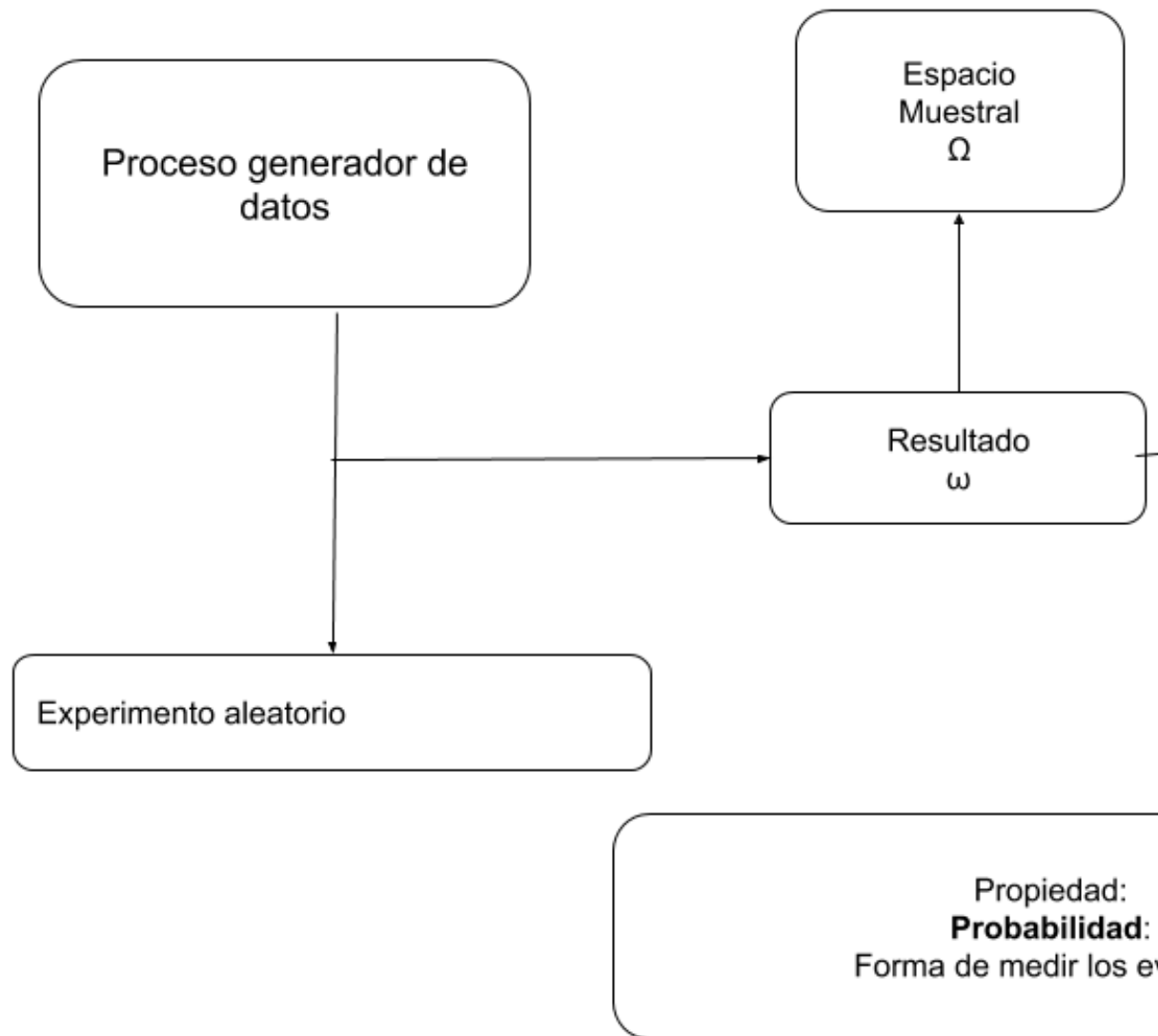
- Introducción a probabilidad
- Introducción a distribuciones
- El problema de la inversión
- Estadística
- Población y muestra
- Estimadores puntuales, tests de hipótesis
- Boxplots, histogramas y kernels

7.1 Explicación

7.1.1 Probabilidad

Previo a estudiar las herramientas de la estadística descriptiva, es necesario hacer un breve resumen de algunos conceptos fundamentales de probabilidad

7.1.1.1 Marco conceptual



- El análisis de las probabilidades parte de un **proceso generador de datos** entendido como cualquier fenómeno que produce algún tipo de información de forma sistemática.
- Cada iteración de este proceso produce información, que podemos interpretar como un **resultado**.
- Existe un conjunto de posibles resultados, que definimos como **espacio muestral**.
- Un **evento** es el conjunto de resultados ocurridos.

- En este marco, la **probabilidad** es un atributo de los eventos. Es la forma de medir los eventos tal que, siguiendo la definición moderna de probabilidad:

- A) $P(A) \geq 0 \forall A \subseteq \Omega$
- B) $P(\Omega) = 1$
- C) $P(A \cup B) = P(A) + P(B)$ si $A \cap B = \emptyset$

ejemplo, tiramos un dado y sale tres

- Espacio muestral: 1,2,3,4,5,6
- Resultado: 3
- Evento: impar (el conjunto 1,3,5)

7.1.1.2 Distribución de probabilidad

- La distribución de probabilidad hace referencia a los posibles valores teóricos de cada uno de los resultados pertenecientes al espacio muestral.
- Existen dos tipos de distribuciones, dependiendo si el espacio muestral es o no numerable.

7.1.1.2.1 Distribuciones discretas

Sigamos con el ejemplo de dado.

Podríamos definir la distribución de probabilidad, si el dado no está cargado, como:

```
## # A tibble: 6 x 2
##   valor probabilidad
##   <int> <chr>
## 1     1 1/6
## 2     2 1/6
## 3     3 1/6
## 4     4 1/6
## 5     5 1/6
## 6     6 1/6
```

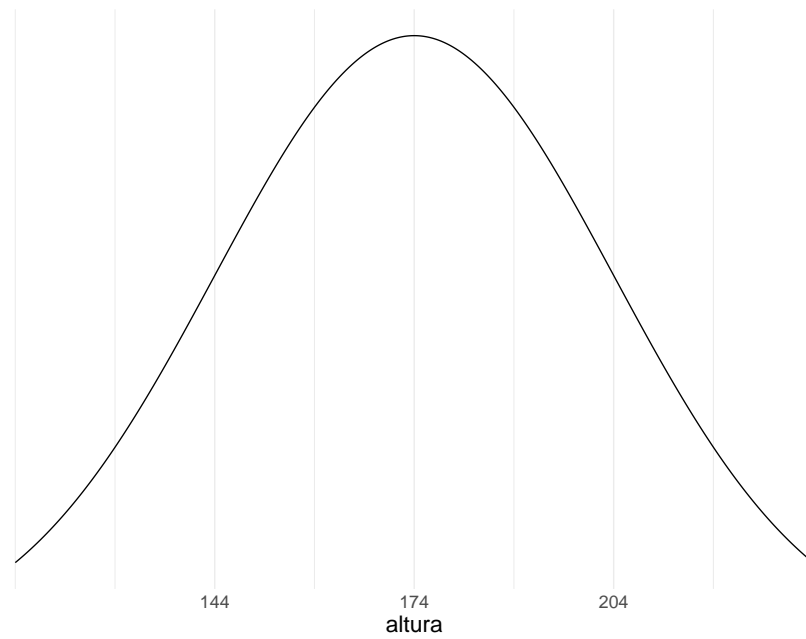
Como el conjunto de resultados posibles es acotado, podemos definirlo en una tabla, esta es una distribución *discreta*.

7.1.1.2.2 Distribuciones continuas

¿Qué pasa cuando el conjunto de resultados posibles es tan grande que no se puede enumerar la probabilidad de cada caso?

Si, por definición o por practicidad, no se puede enumerar cada caso, lo que tenemos es una **distribución continua**

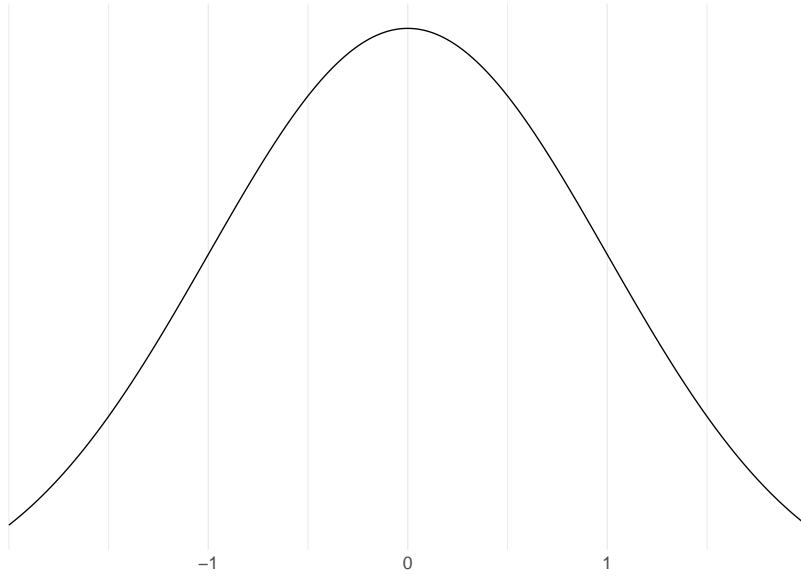
Por ejemplo, la altura de la población



- En este caso, no podemos definir en una tabla la probabilidad de cada uno de los posibles valores. *de hecho, la probabilidad puntual es 0.*
- Sin embargo, sí podemos definir una *función de probabilidad*, la *densidad*.
- Según qué función utilicemos, cambiará la forma de la curva.

Por ejemplo:

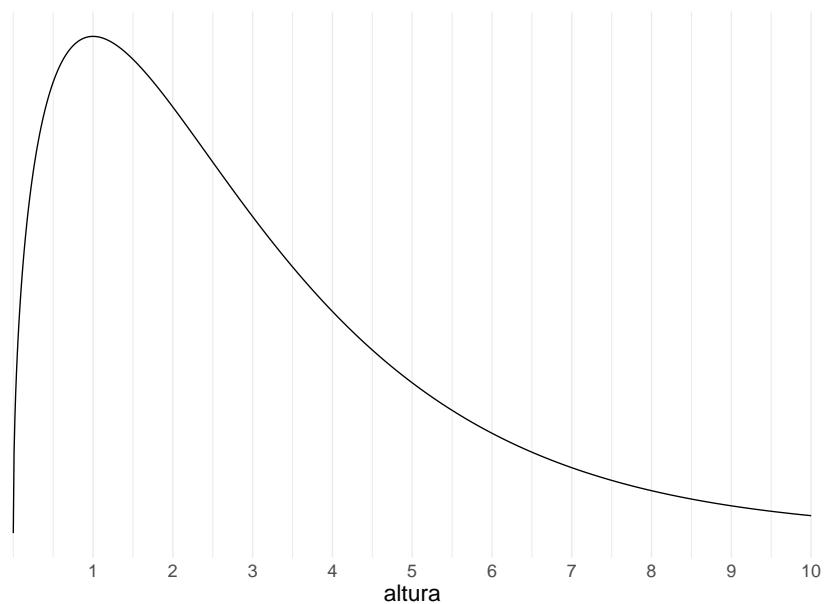
Distribución Normal



Distribución t



Distribución Chi cuadrado

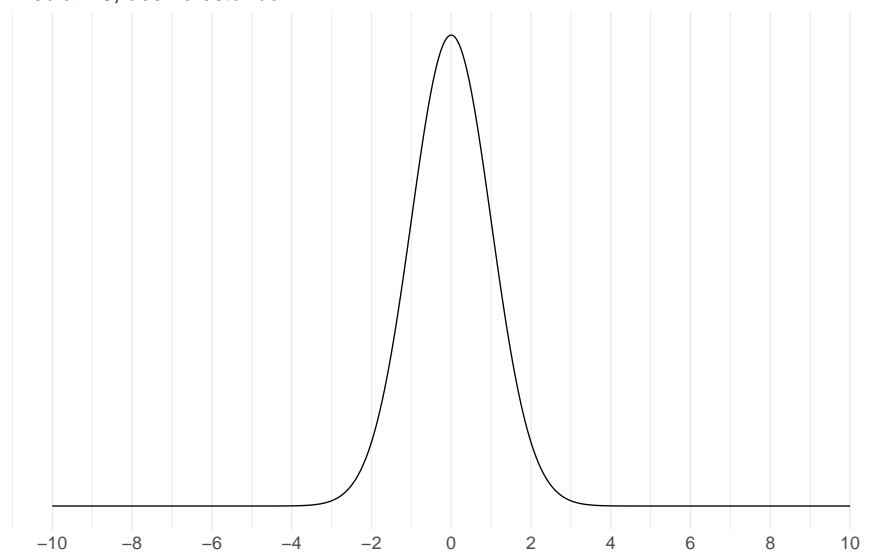


Una distribución de probabilidad se **caracteriza** por sus *parámetros*.

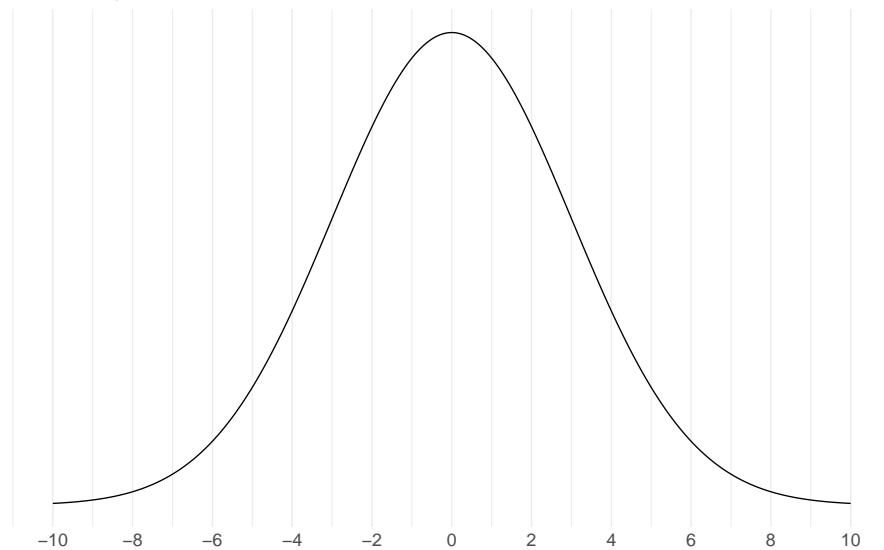
- Por ejemplo, la distribución normal se caracteriza por su *esperanza* y su *varianza* (o desvío estándar)

Distribución Normal

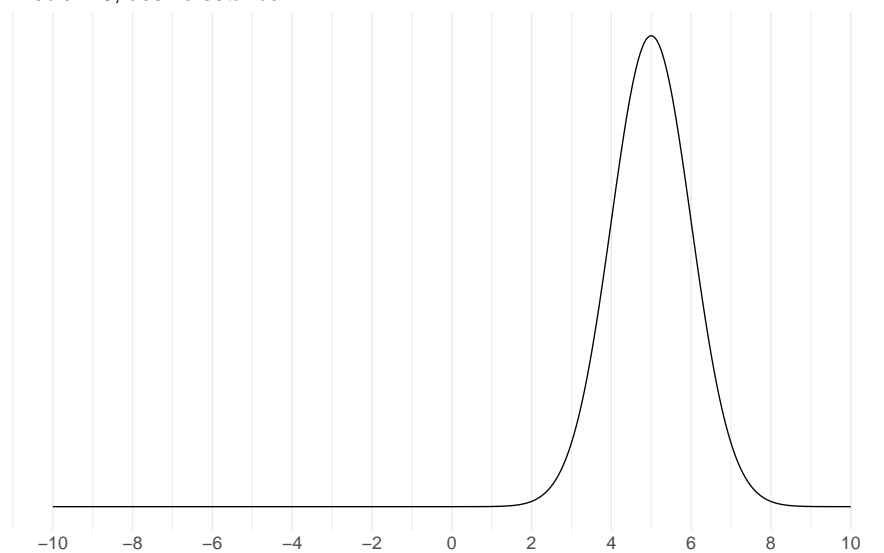
media = 0, desvío estándar = 1



Distribución Normal
media = 0, desvío estándar = 3



Distribución Normal
media = 5, desvío estándar = 1



7.1.2 Estadística

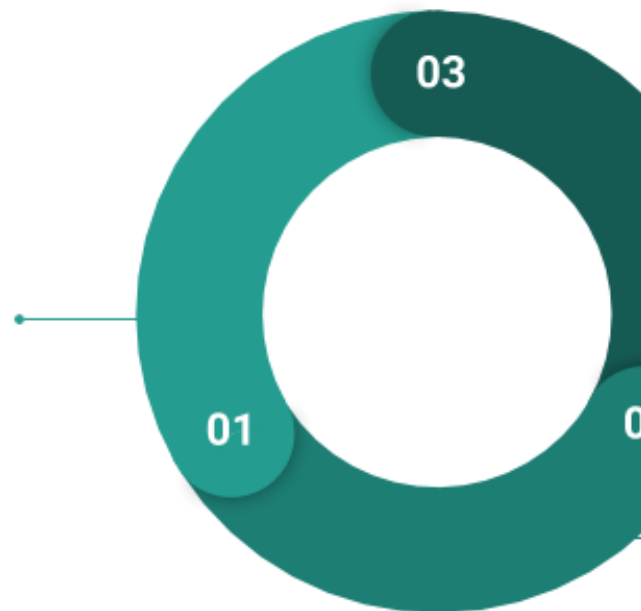
7.1.2.1 El problema de la inversión

El problema de la probabilidad se podría pensar de la siguiente forma:

1. Vamos a partir de un **proceso generador de datos**
2. Para calcular su **distribución de probabilidad**, los **parámetros** que caracterizan a ésta, y a partir de allí,
3. Calcular la probabilidad de que, al tomar una **muestra**, tenga ciertos eventos.

El problema de la inversión I: La

**Proceso Generador
de Datos**

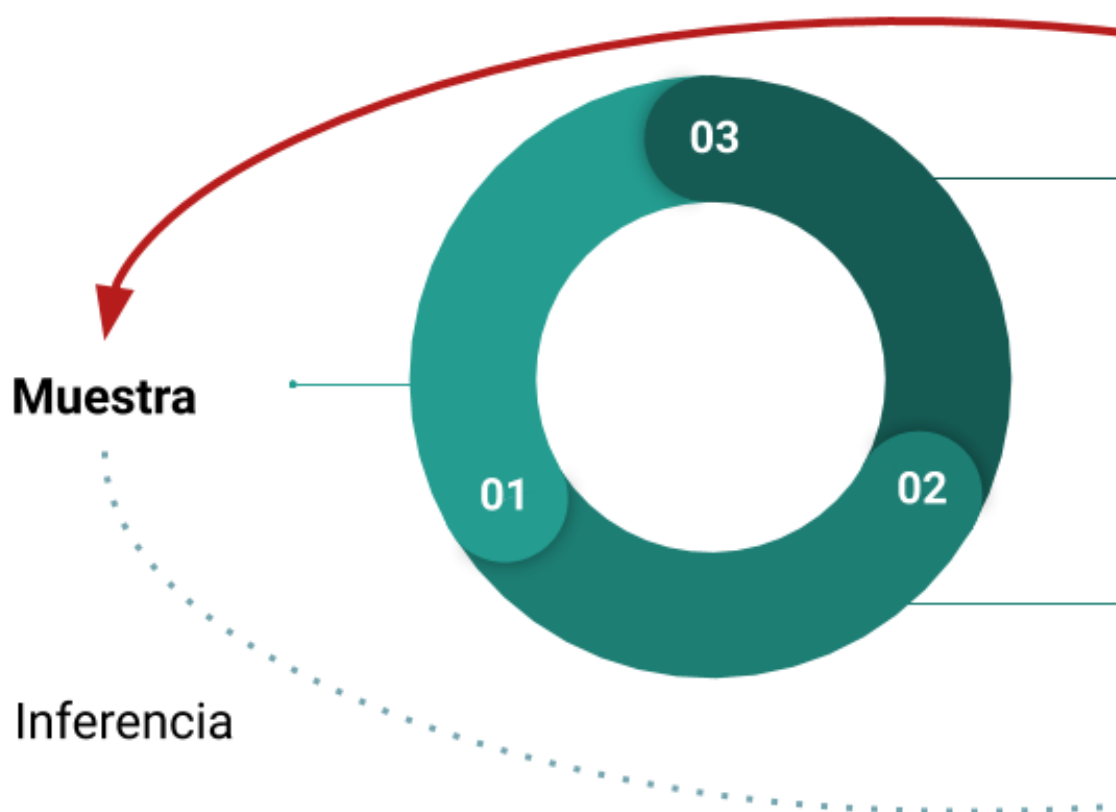


El problema de la estadística es exactamente el contrario:

1. Partimos de una **muestra** para
2. Inferir cuál es la **distribución de probabilidad**, y los **parámetros** que la caracterizan

3. Para finalmente poder sacar conclusiones sobre el **proceso generador de datos**

El problema de la inversión II: La inferencia



7.1.2.1.1 Población y muestra

En este punto podemos hacer la distinción entre **población** y **muestra**

- **Población:** El universo en estudio. Puede ser:
 - finita: Los votantes en una elección.
 - infinita: El lanzamiento de una moneda.
- **Muestra:** subconjunto de n observaciones de una población.

Solemos utilizar las mayúsculas (N) para la población y las minúsculas (n) para las muestras

7.1.2.1.2 Parámetros y Estimadores

- Como dijimos, los **parámetros** describen a la función de probabilidad. Por lo tanto hacen referencia a los atributos de la **población**. Podemos suponer que son *constantes*.
- Un **estimador** es un estadístico (esto es, una función de la muestra) usado para estimar un parámetro desconocido de la población.

7.1.2.1.3 Ejemplo. La media

Esperanza o Media Poblacional:

$$\mu = E(x) = \sum_{i=1}^N x_i p(x_i)$$

Media muestral:

$$\bar{X} = \sum_{i=1}^n \frac{X_i}{n}$$

Como no puedo conocer μ , lo estimo mediante \bar{X}

7.1.2.2 Estimación puntual, Intervalos de confianza y Tests de hipótesis

- El estimador \bar{X} nos devuelve un número. Esto es una inferencia de cuál creemos que es la media. Pero no es seguro que esa sea realmente la media. Esto es lo que denominamos estimación puntual.
- También podemos estimar un intervalo, dentro del cual consideramos que se encuentra la media poblacional. La ventaja de esta metodología es que podemos definir la probabilidad de que el parámetro poblacional realmente esté dentro de este intervalo. Esto se conoce como **intervalos de confianza**.
- Por su parte, también podemos calcular la probabilidad de que el parámetro poblacional sea mayor, menor o igual a un cierto valor. Esto es lo que se conoce como **test de hipótesis**.
- En el fondo, los intervalos de confianza y los tests de hipótesis se construyen de igual manera. Son funciones que se construyen a partir de los datos, que se comparan con distribuciones conocidas, *teóricas*.

7.1.2.2.1 Definición de los tests

- Los tests se construyen con dos hipótesis: La hipótesis nula H_0 , y la hipótesis alternativa, H_1 . Lo que buscamos es ver si *hay evidencia suficiente para rechazar la hipótesis nula*.

Por ejemplo, si queremos comprobar si la media poblacional, μ de una distribución es mayor a X_i , haremos un test con las siguientes hipótesis:

- $H_0 : \mu = X_i$
- $H_1 : \mu > X_i$

Si la evidencia es lo suficientemente fuerte, podremos rechazar la hipótesis H_0 , pero no afirmar la hipótesis H_1

7.1.2.2.2 Significatividad en los tests

- Muchas veces decimos que algo es “**estadísticamente significativo**”. Detrás de esto se encuentra un test de hipótesis que indica que hay una suficiente *significatividad estadística*.
- La *significatividad estadística*, representada con α , es la probabilidad de rechazar H_0 cuando en realidad es cierta. Por eso, cuanto más bajo el valor de α , más seguros estamos de no equivocarnos. Por lo general testeamos con valores de alpha de 1%, 5% y 10%, dependiendo del área de estudio.
- El **p-valor** es la *mínima significatividad* para la que rechazo el test. Es decir, cuanto más bajo es el p-valor, más seguros estamos de rechazar H_0 .
- El resultado de un test está determinado por:
 1. **La fuerza de la evidencia empírica:** Si nuestra duda es si la media poblacional es mayor a, digamos, 10, y la media muestral es 11, no es lo mismo que si es 100, 1000 o 10000.
 2. **El tamaño de la muestra:** En las fórmulas que definen los test siempre juega el tamaño de la muestra: cuanto más grande es, más seguros estamos de que el resultado no es producto del mero azar.
 3. **La veracidad de los supuestos:** Otra cosa importante es que los test asumen ciertas cosas:
 - Normalidad en los datos.
 - Que conocemos algún otro parámetro de la distribución, como la varianza.
 - Que los datos son independientes entre sí,
 - Etc.

Cada Test tiene sus propios supuestos. Por eso a veces, luego de hacer un test, hay que hacer otros tests para validar que los supuestos se cumplen.
- Lo primero, la fuerza de la evidencia, es lo que más nos importa, y no hay mucho por hacer.

- El tamaño de la muestra es un problema, porque si la muestra es muy chica, entonces podemos no llegar a conclusiones significativas aunque sí ocurra aquello que queríamos probar.
- Sin embargo, el verdadero problema en *La era del big data* es que tenemos muestras demasiado grandes, por lo que cualquier test, por más mínima que sea la diferencia, puede dar significativo.

Por ejemplo, podemos decir que la altura promedio en Argentina es 1,74. Pero si hacemos un test, utilizando como muestra 40 millones de personas, vamos a rechazar que ese es el valor, porque en realidad es 1,7401001. En términos de lo que nos puede interesar, 1,74 sería válido, pero estadísticamente rechazaríamos.

- Finalmente, según la información que tengamos de la población y cuál es el problema que queremos resolver, vamos a tener que utilizar distintos tipos de tests. La cantidad de tests posibles es ENORME, y escapa al contenido de este curso, así como sus fórmulas. A modo de ejemplo, les dejamos el siguiente machete:

7.1.3 Algunos estimadores importantes

7.1.3.1 Medidas de centralidad

- **Media**

$$\bar{X} = \sum_{i=1}^n \frac{X_i}{n}$$

- **Mediana:**

Es el valor que parte la distribución a la mitad

- **Moda**

La moda es el valor más frecuente de la distribución

Flow Chart for Selecting Commonly Used Statistical Tests

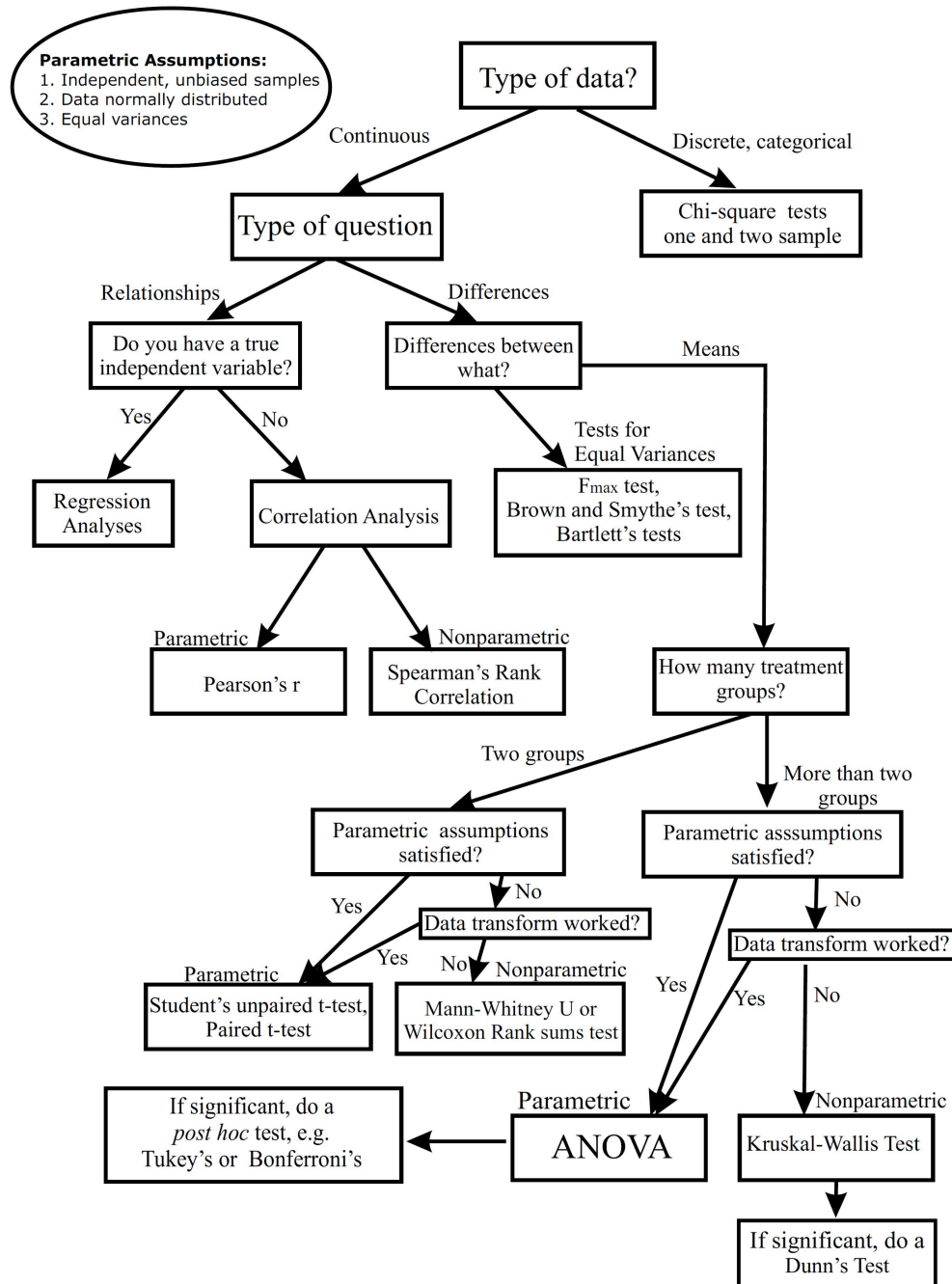
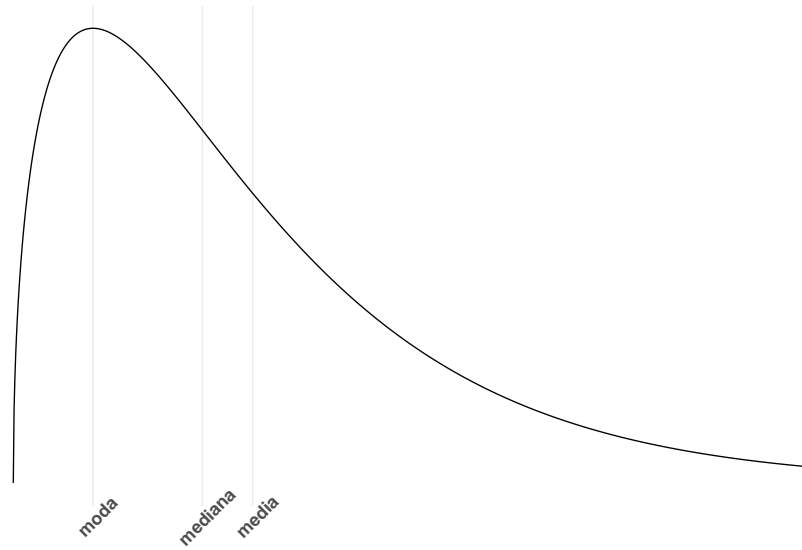


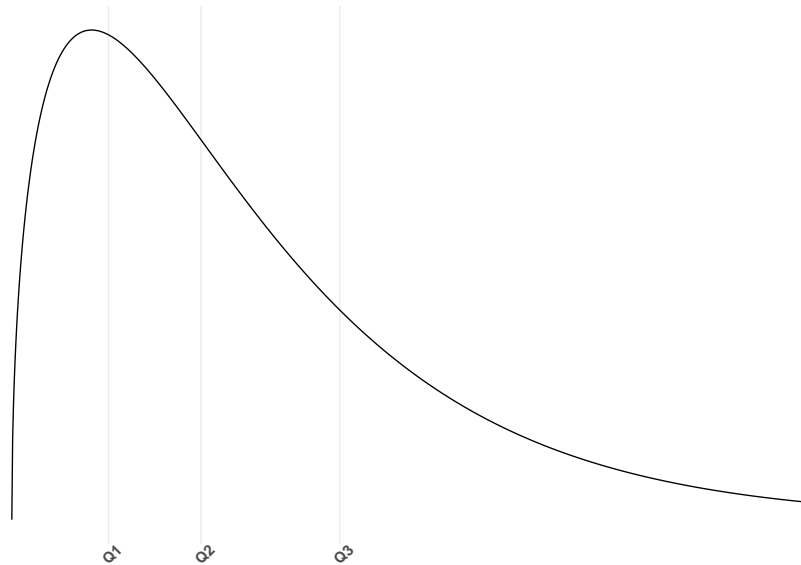
Figure 7.1: fuente: <http://abacus.bates.edu/~ganderso/biology/resources/statistics.html>



7.1.3.2 Cuantiles

Así como dijimos que la mediana es el valor que deja al 50% de los datos de un lado y al 50% del otro, podemos generalizar este concepto a cualquier $X\%$. Esto son los cuantiles. El cuantil x , es el valor tal que queda un $x\%$ de la distribución a izquierda, y $1-x$ a derecha.

Algunos de los más utilizados son el del 25%, también conocido como Q_1 (el *cuartil* 1), el Q_2 (la mediana) y el Q_3 (el *cuartil* 3), que deja el 75% de los datos a su derecha. Veamos cómo se ven en la distribución de arriba.



7.1.3.3 Desvío estándar

- El *desvío estándar* es una medida de dispersión de los datos, que indica cuánto se suelen alejar de la media.

7.1.4 Gráficos estadísticos

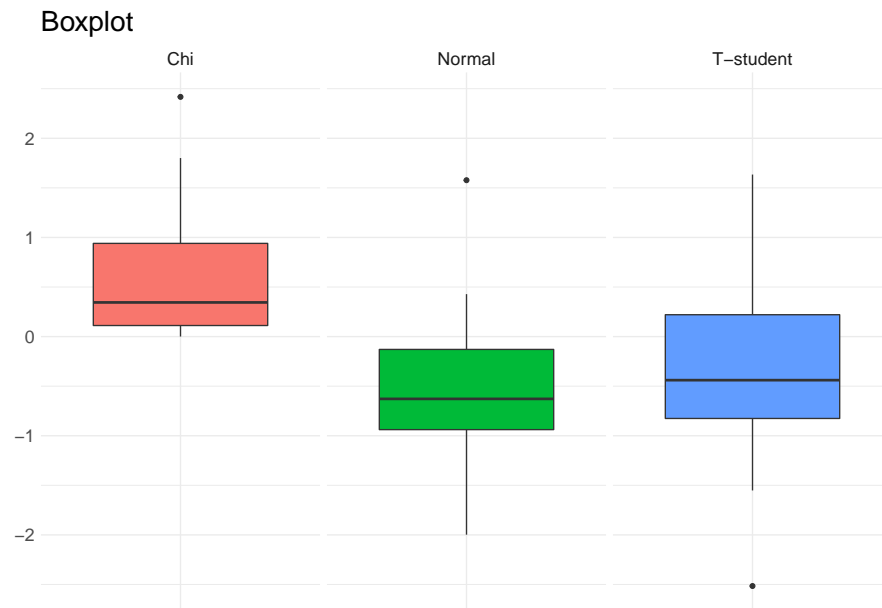
Cerramos la explicación con algunos gráficos que resultan útiles para entender las propiedades estadísticas de los datos.

7.1.4.1 Boxplot

El Boxplot es muy útil para describir una distribución y para detectar outliers. Reúne los principales valores que caracterizan a una distribución:

- Q_1
- Q_2 (la mediana)
- Q_3
- el *rango intercuartílico* $Q_3 - Q_1$, que define el centro de la distribución
- Outliers, definidos como aquellos puntos que se encuentran a más de 1,5 veces el rango intercuartílico del centro de la distribución.

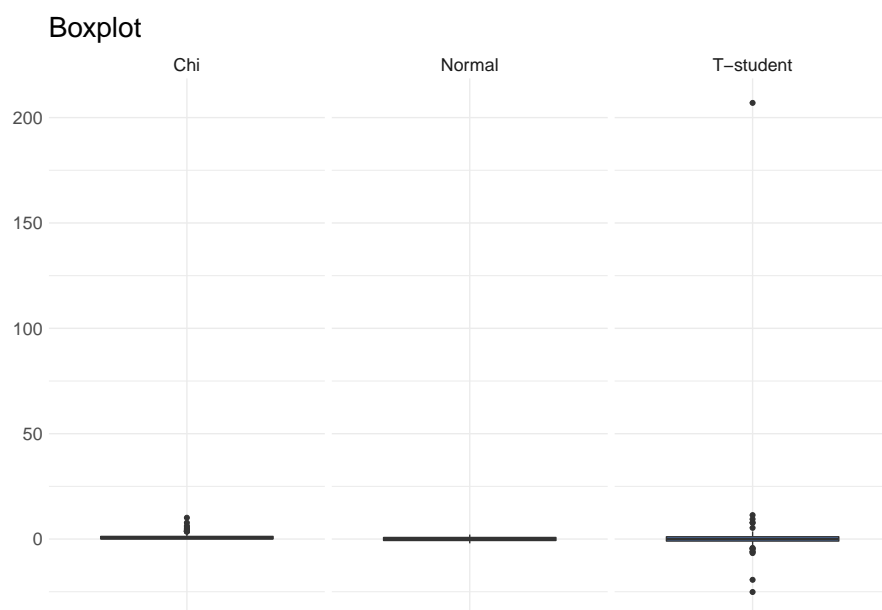
Veamos qué pinta tienen los boxplot de números generados aleatoriamente a partir de tres distribuciones que ya vimos. En este caso, sólo tomaremos 15 valores de cada distribución



Algunas cosas que resaltan:

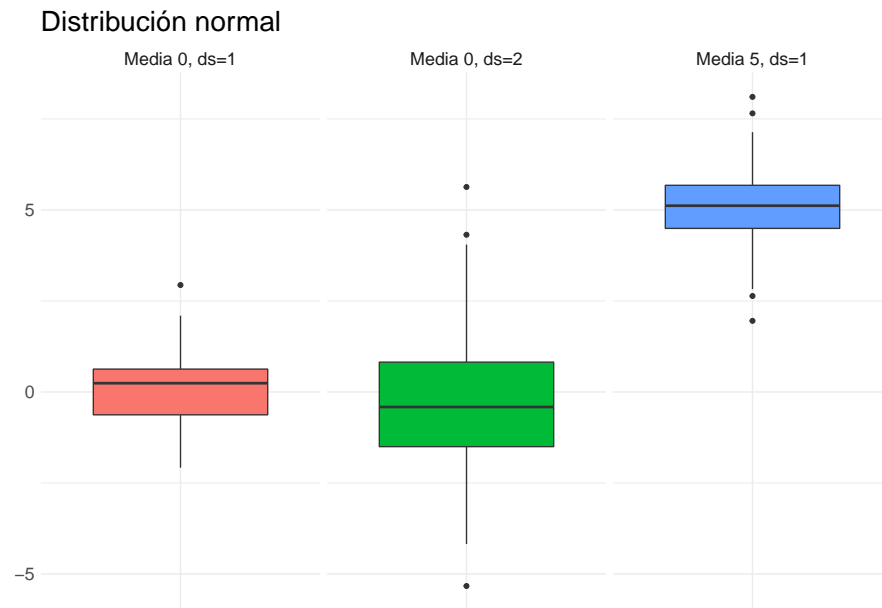
- la distribución χ^2 no toma valores en los negativos.
- La normal esta más concentrada en el centro de la distribución.

Podemos generar 100 números aleatorios en lugar de 15:



Cuando generamos 100 valores en lugar de 15, tenemos más chances de agarrar un punto alejado en la distribución. De esta forma podemos apreciar las diferencias entre la distribución normal y la T-student.

También podemos volver a repasar qué efecto generan los distintos parámetros. Por ejemplo:



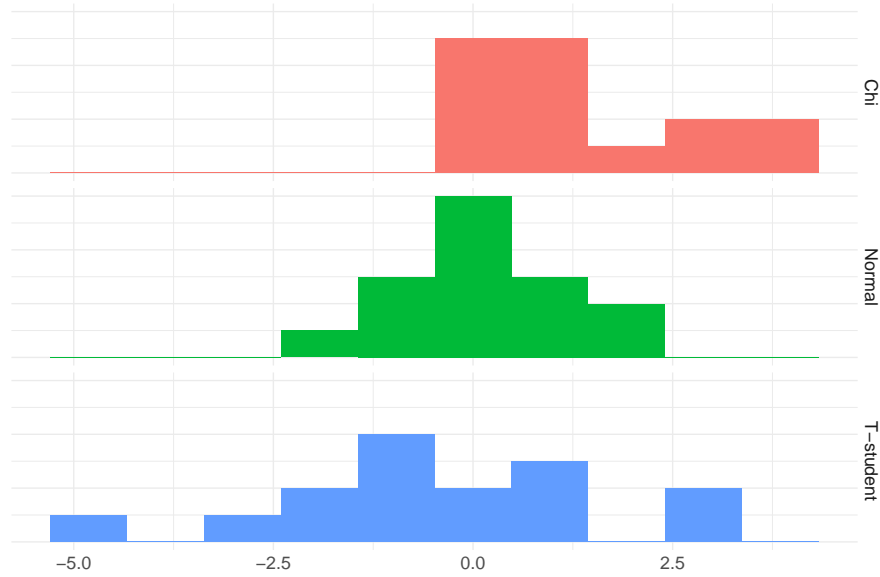
7.1.4.2 Histograma

Otra forma de analizar una distribución es mediante los histogramas:

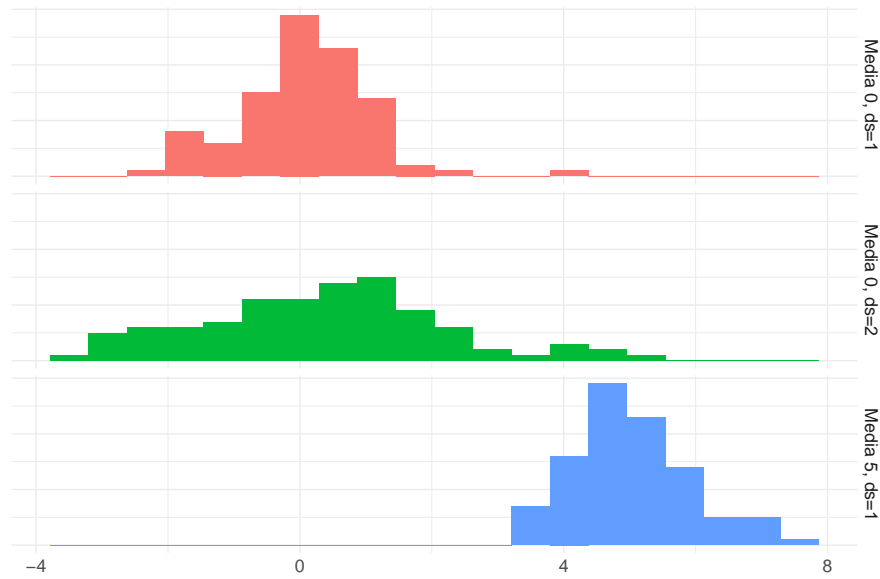
- En un histograma agrupamos las observaciones en rangos fijos de la variable y contamos la cantidad de ocurrencias.
- Cuanto más alta es una barra, es porque más observaciones se encuentran en dicho rango.

Veamos el mismo ejemplo que arriba, pero con histogramas:

Histograma



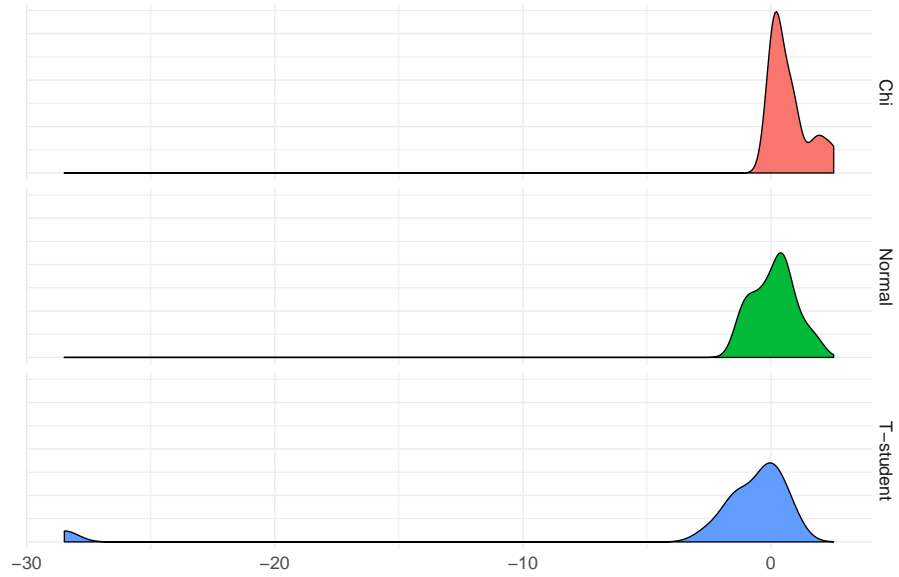
Distribución normal



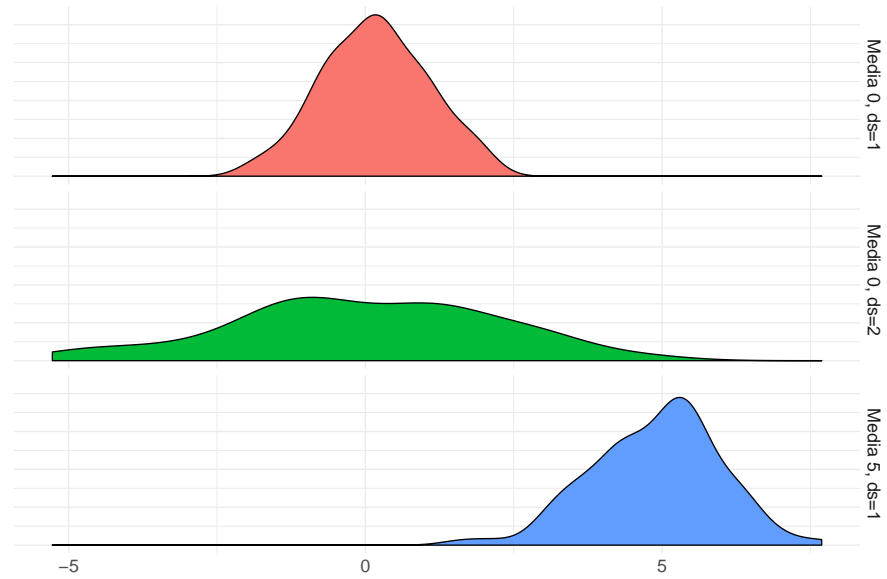
7.1.4.3 Kernel

Los Kernels son simplemente un suavizado sobre los histogramas.

Histograma

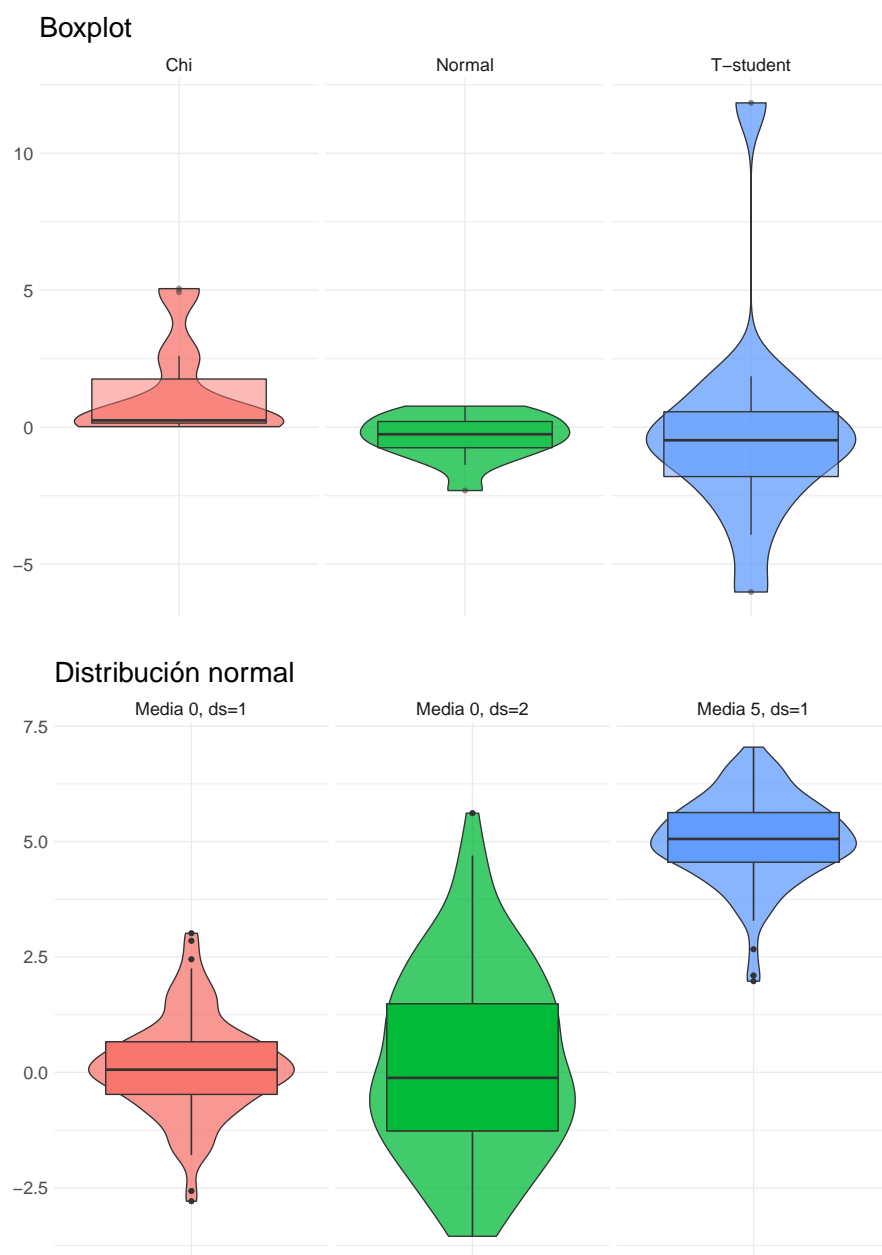


Distribución normal



7.1.4.4 Violin plots

Combinando la idea de Kernels y Boxplots, se crearon los violin plots, que simplemente muestran a los kernels duplicados.



7.1.5 Bibliografía de consulta

Quién quiera profundizar en estos temas, puede ver los siguientes materiales:

- <https://seeing-theory.brown.edu/>

- https://lagunita.stanford.edu/courses/course-v1:OLI+ProbStat+Open_Jan2017/about
- Jay L. Devore, “Probabilidad y Estadística para Ingeniería y Ciencias”, International Thomson Editores. <https://inferencialtm.files.wordpress.com/2018/04/probabilidad-y-estadistica-para-ingenieria-y-ciencias-devore-7th.pdf>

7.2 Práctica Guiada

```
library(tidyverse)
```

7.2.1 Generación de datos aleatorios

Para generar datos aleatorios, usamos las funciones:

- `rnorm` para generar datos que surgen de una distribución normal
- `rt` para generar datos que surgen de una distribución T-student
- `rchisq` para generar datos que surgen de una distribución Chi cuadrado

Pero antes, tenemos que fijar la *semilla* para que los datos sean reproducibles

```
set.seed(1234)
rnorm(n = 15, mean = 0, sd = 1 )
```

```
## [1] -1.20706575  0.27742924  1.08444118 -2.34569770  0.42912469
## [6]  0.50605589 -0.57473996 -0.54663186 -0.56445200 -0.89003783
## [11] -0.47719270 -0.99838644 -0.77625389  0.06445882  0.95949406
```

```
rt(n = 15, df=1 )
```

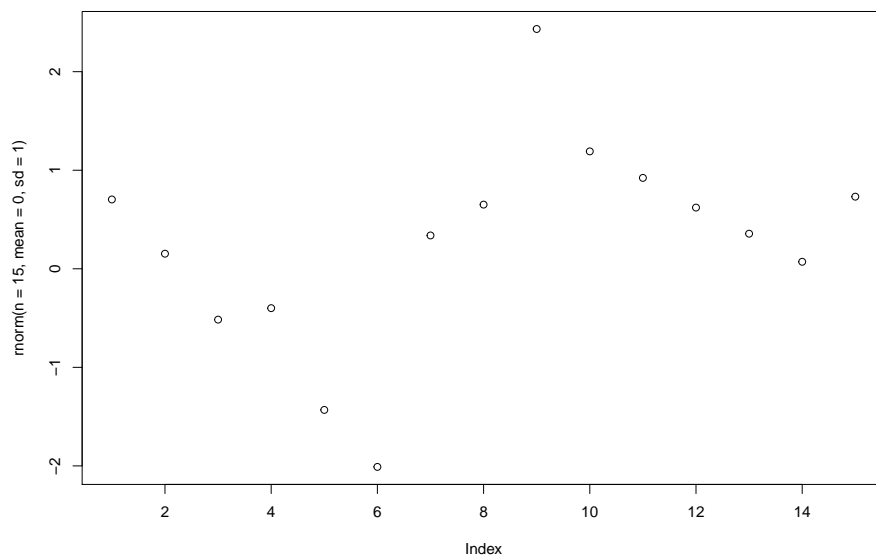
```
## [1] -0.363717710 -1.603466805 -0.388596796 -0.588007490  0.007839245
## [6] 14.690527710 -1.863488555  0.022667470 -2.084247299 -0.249237745
## [11] -1.311594174 -3.569055208 -2.490838240 -3.848779244 -4.271087169
```

```
rchisq(n = 15,df=1)
```

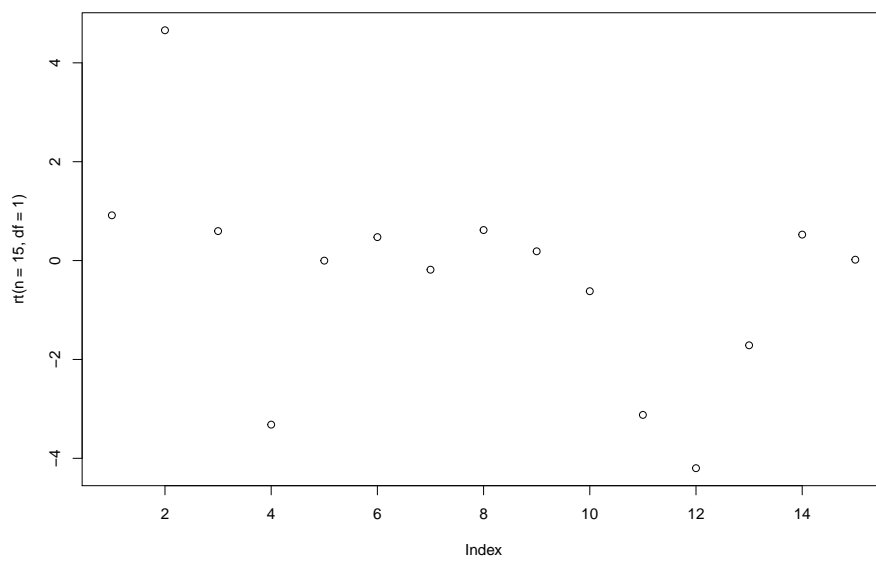
```
## [1] 0.5317744 1.4263809 4.2797098 0.2184660 0.6923773 0.0455256 3.1902100
## [8] 0.2949942 0.5403827 0.1543732 0.8639196 0.1417290 1.1386091 0.2966193
## [15] 0.5110879
```

Para poder ver rápidamente de qué se tratan los valores, podemos usar el comando `plot`

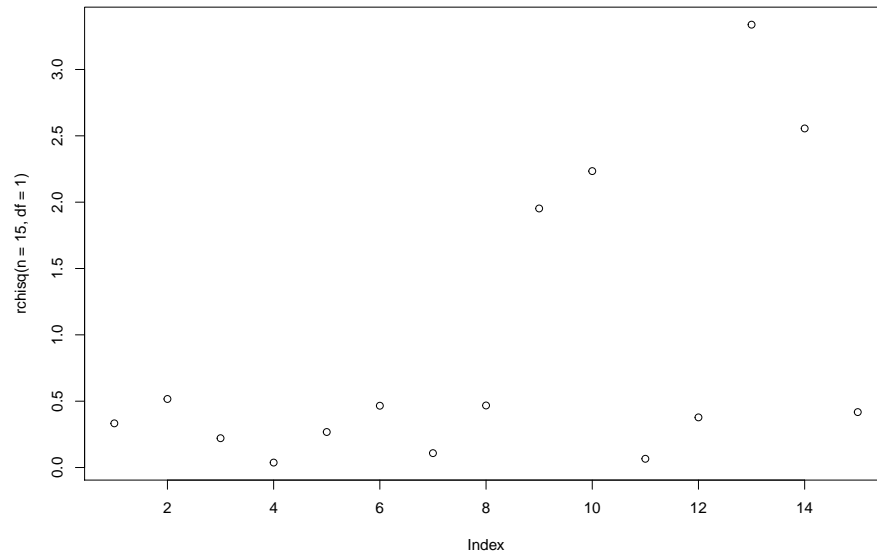
```
plot(rnorm(n = 15,mean = 0, sd = 1 ))
```



```
plot(rt(n = 15,df=1 ))
```



```
plot(rchisq(n = 15, df=1))
```



Noten que el eje X es el índice de los valores, es decir que no agrega información.

7.2.2 Tests

Utilicemos ahora datos reales.

Los datos salen de <https://data.buenosaires.gob.ar/dataset/femicidios>

Vamos a ver ahora las estadísticas de Buenos Aires sobre la cantidad de femicidios por grupo etario. Es interesante preguntarse si hay más femicidios para cierto rango etario.

```
femicidios <- read_csv(file = 'fuentes/vict_fem_annio__g_edad_limpio.csv')
femicidios
```

```
## # A tibble: 19 x 3
##   anio cantidad_femicidios grupo_edad
##   <dbl> <chr>               <chr>
## 1  2015 1                0 - 15
## 2  2015 2                16 - 20
## 3  2015 5                21 - 40
## 4  2015 3                41 - 60
## 5  2015 -                61 y más
## 6  2015 1                Ignorado
## 7  2016 2                0 - 15
```

```
## 8 2016 3 16 - 20
## 9 2016 4 21 - 40
## 10 2016 1 41 - 60
## 11 2016 2 61 y más
## 12 2016 2 Ignorado
## 13 2017 ... 0 - 15
## 14 2017 ... 16 - 20
## 15 2017 ... 21 - 40
## 16 2017 ... 41 - 60
## 17 2017 ... 61 y más
## 18 2017 ... Ignorado
## 19 2017 9 TOTAL
```

Fijense que las estadísticas no están desagregadas por rango etario para 2017, que en caso de que haya 0 femicidios pusieron '-' en lugar de 0. Además, como tenemos pocos datos, es mejor hacer un test que compare solamente dos grupos.

Vamos a reorganizar la información para corregir todas estas cosas

```
femicidios <- femicidios %>%
  filter(anio!=2017, grupo_edad !='Ignorado') %>% #Sacamos al 2017 y los casos donde se ignora l
  mutate(cantidad_femicidios = case_when(cantidad_femicidios=='-' ~ 0, # reemplazamos el - por 0
                                          TRUE ~as.numeric(cantidad_femicidios)), # y convertimos
  grupo_edad = case_when(grupo_edad %in% c('0 - 15','16 - 20','21 - 40') ~ '0-40', # agrup
                          grupo_edad %in% c('41 - 60','61 y más') ~ '41 y más')) %>%
  group_by(grupo_edad) %>%
  summarise(cantidad_femicidios= sum(cantidad_femicidios)) # sumamos los años y grupos para tener
femicidios
```

```
## # A tibble: 2 x 2
##   grupo_edad cantidad_femicidios
##   <chr>          <dbl>
## 1 0-40          17
## 2 41 y más      6
```

Con esta tabla de contingencia podemos hacer un test de hipótesis.

¿Cuál usamos? Nos fijamos en el machete, o googleamos, y vemos que como queremos comparar la cantidad de casos por grupos categóricos, tenemos que usar el test Chi.

- H_0 No hay asociación entre las variables
- H_1 Hay asociación entre las variables

La idea es que tenemos dos variables: El rango etario y la cantidad de femicidios

```
chisq.test(femicidios$cantidad_femicidios)
```

```
##
## Chi-squared test for given probabilities
```

```
##
## data:  femicidios$cantidad_femicidios
## X-squared = 5.2609, df = 1, p-value = 0.02181
```

Noten que el resultado lo dan en términos del p-valor. Como el valor es bajo, menor a 0.05, entonces podemos rechazar que no existe relación. O en otros términos, pareciera que la diferencia es significativa estadísticamente.

7.2.3 Descripción estadística de los datos

Volveremos a ver los datos de sueldos de funcionarios

```
suealdos <- read_csv('fuentes/sueldo_funcionarios_2019.csv')
```

Con el comando `summary` podemos ver algunos de los principales estadísticos de resumen

```
summary(suealdos$asignacion_por_cargo_i)
```

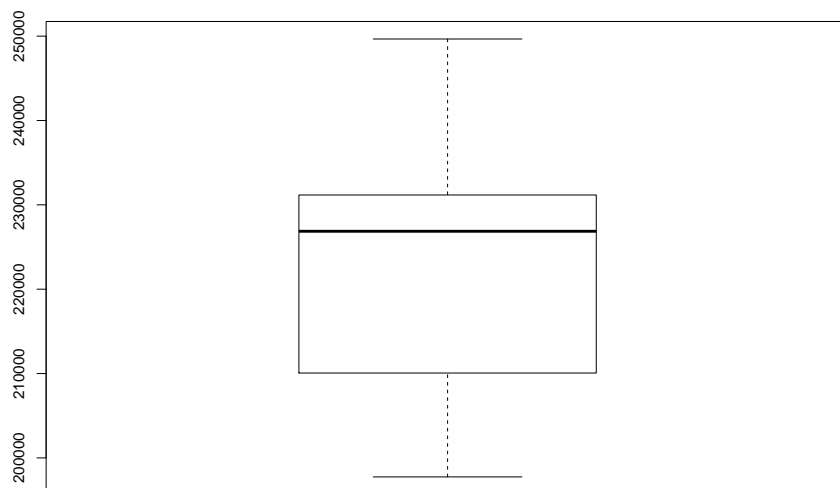
```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 197746  210061  226866  225401  231168  249662
```

7.2.4 Gráficos estadísticos

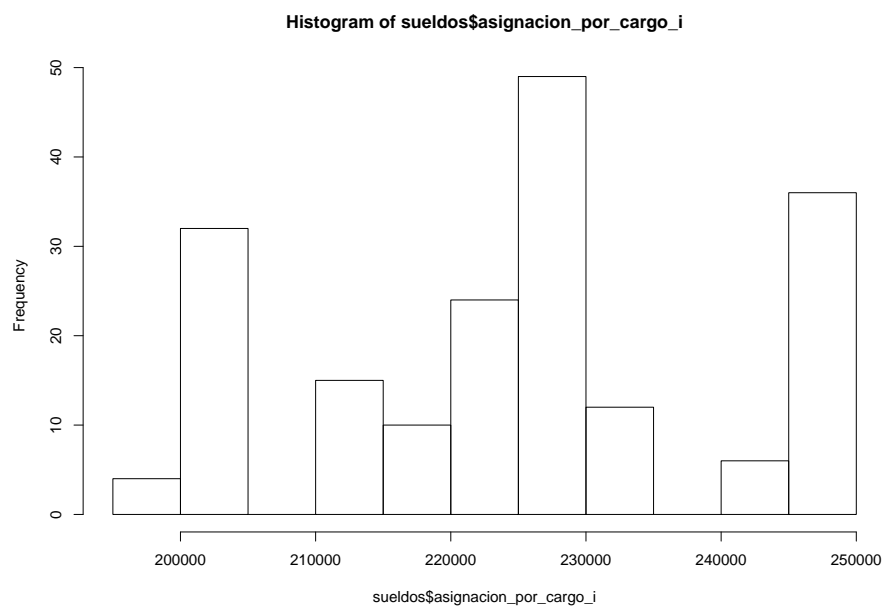
No nos vamos a detener demasiado a ver cómo hacer los gráficos de resumen, porque la próxima clase veremos como realizar gráficos de mejor calidad, como los presentados en las notas de clase.

A modo de ejemplo, dejamos los comandos de R base para realizar gráficos.

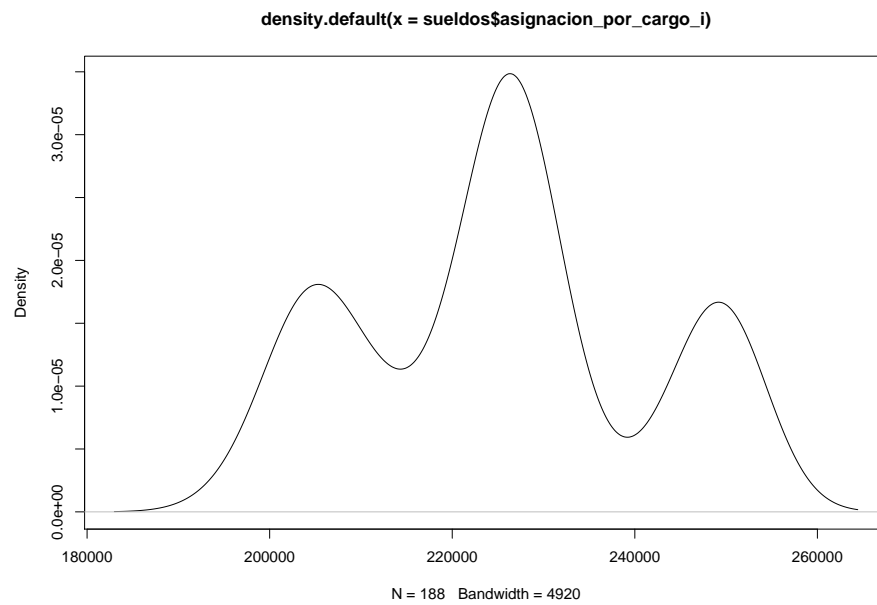
```
boxplot(suealdos$asignacion_por_cargo_i)
```

```
hist(sueldos$asignacion_por_cargo_i)
```



```
plot(density(sueldos$asignacion_por_cargo_i))
```



Chapter 8

Modelo Lineal

- Análisis de correlación.
- Presentación conceptual del modelo lineal
- El modelo lineal desde una perspectiva computacional
- Supuestos del modelo lineal
- Modelo lineal en R
- Modelo lineal en el tidyverse

8.1 Explicación

En este módulo vamos a ver cómo analizar la relación entre dos variables. Primero, veremos los conceptos de covarianza y correlación, y luego avanzaremos hasta el modelo lineal.

```
knitr::opts_chunk$set(warning = FALSE, message = FALSE)
library(tidyverse)
library(modelr)
library(GGally)
library(plot3D)
```

8.1.1 Covarianza y Correlación.

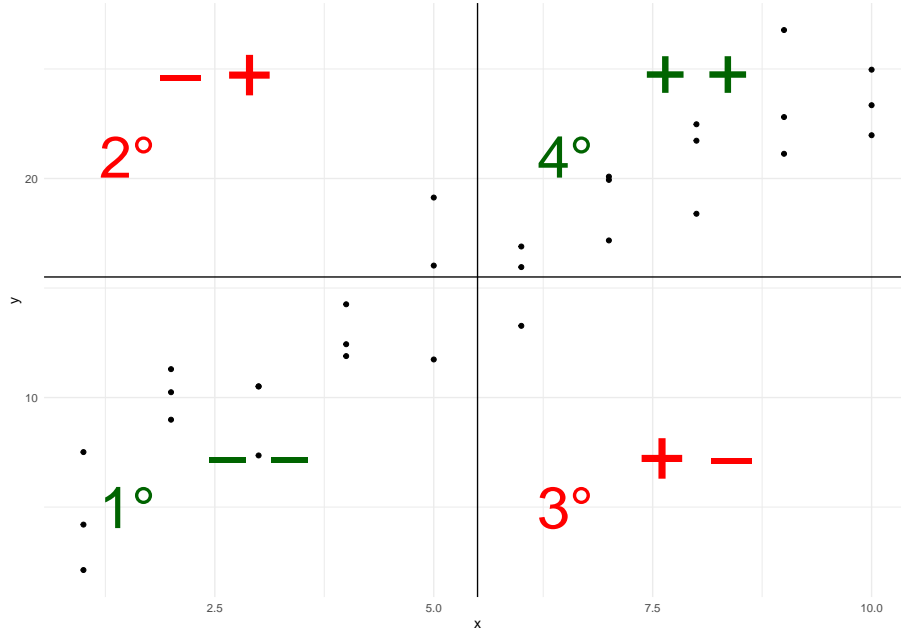
La covarianza mide cómo varían de forma conjunta dos variables, en promedio. Se define como:

$$\text{cov}(x, y) = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$$

Esto es: La covarianza entre dos variables, x e y es el promedio (noten que hay

una sumatoria y un dividido n) de las diferencias de los puntos a sus medias en x e y .

tratemos de entender el trabalenguas con la ayuda del siguiente gráfico:



Aquí marcamos \bar{x} y \bar{y} y dividimos el gráfico en cuatro cuadrantes.

1. En el primer cuadrante los puntos son más chicos a sus medias en x y en y , $(x - \hat{x})$ es negativo y $(y - \hat{y})$ también. Por lo tanto, su producto es positivo.
 2. En el segundo cuadrante la diferencia es negativa en x , pero positiva en y . Por lo tanto el producto es negativo.
 3. En el tercer cuadrante la diferencia es negativa en y , pero positiva en x . Por lo tanto el producto es negativo.
 4. Finalmente, en el cuarto cuadrante las diferencias son positivas tanto en x como en y , y por lo tanto también el producto.
- Si la covarianza es **positiva** y grande, entonces valores chicos en una de las variables suceden en conjunto con valores chicos en la otra, y viceversa.
 - Al contrario, si la covarianza es **negativa** y grande, entonces valores altos de una variable suceden en conjunto con valores pequeños de la otra y viceversa.

La correlación se define como sigue:

$$\rho_{x,y} = \frac{cov(x,y)}{\sigma_x \sigma_y}$$

Es decir, normalizamos la covarianza por el desvío en x y en y . de esta forma, la correlación se define entre -1 y 1.

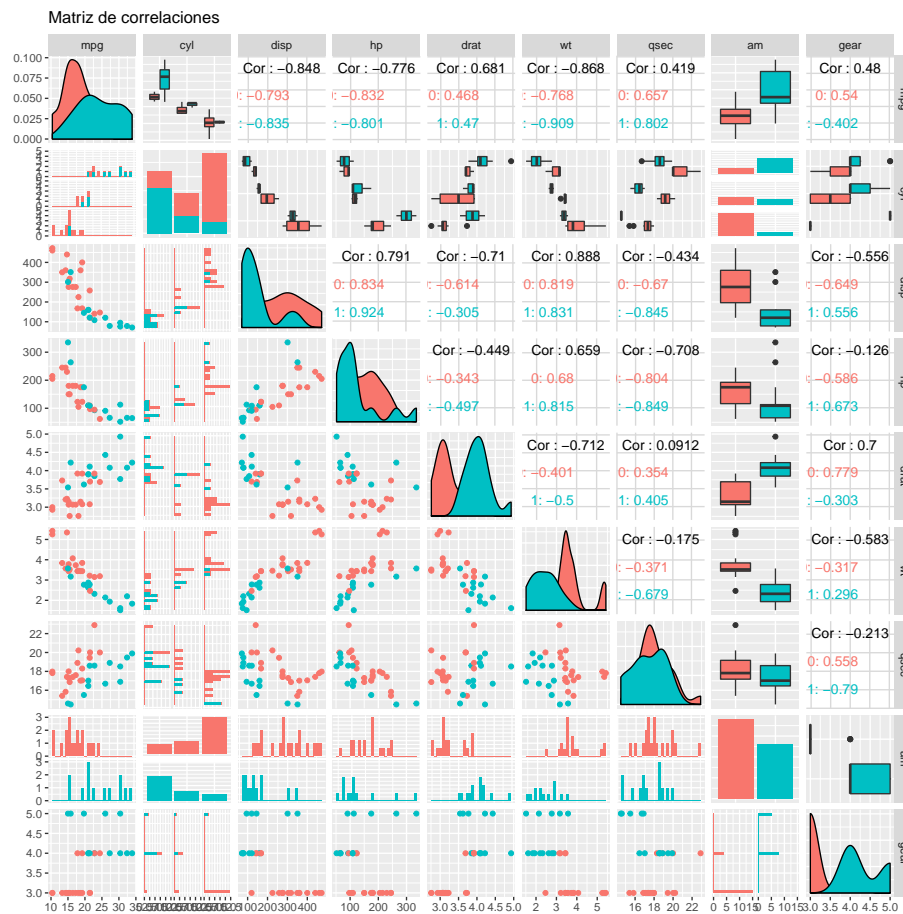
8.1.1.1 ggpairs

Para ver una implementación práctica de estos conceptos, vamos a utilizar la librería *GGally* para graficar la correlación por pares de variables.

- Con `ggpairs()`, podemos graficar todas las variables, y buscar las correlaciones. Coloreamos por:

-*am*: Tipo de transmisión: automático (*am*=0) o manual (*am*=1)

```
mtcars %>%  
  select(-carb, -vs) %>%  
  mutate(cyl = factor(cyl),  
         am = factor(am)) %>%  
  ggpairs(.,  
         title = "Matriz de correlaciones",  
         mapping = aes(colour= am))
```



Veamos la correlación entre:

- *mpg*: Miles/(US) gallon. Eficiencia de combustible
- *hp*: Gross horsepower: Potencia del motor

```
cor(mtcars$mpg, mtcars$hp)
```

```
## [1] -0.7761684
```

nos da negativa y alta.

- Si quisiéramos testear la significatividad de este estimador, podemos realizar un test:

$$H_0 : = 0$$

$$H_1 : \neq 0$$

```
cor.test(mtcars$mpg, mtcars$hp)
```

```
##
```

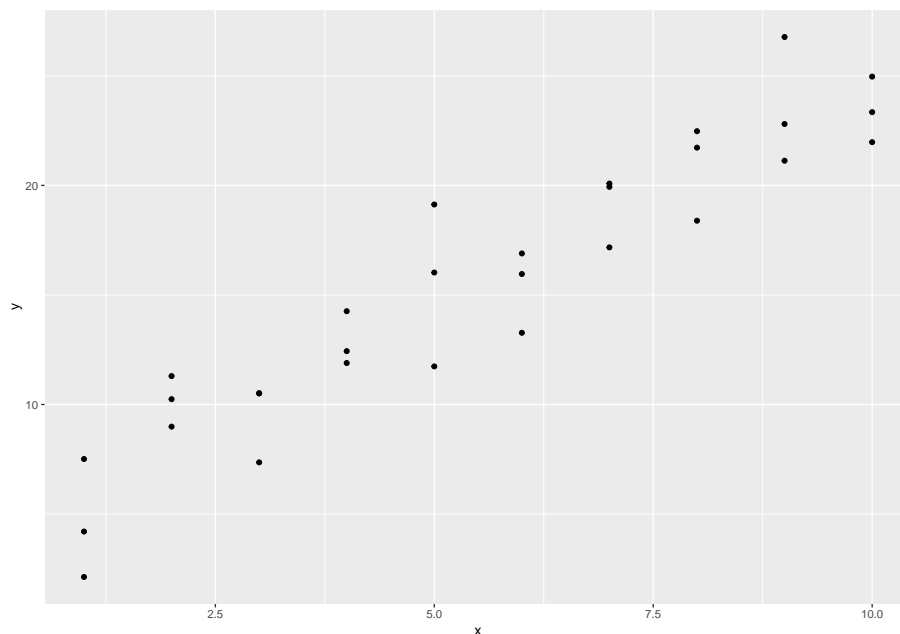
```
## Pearson's product-moment correlation
##
## data: mtcars$mpg and mtcars$hp
## t = -6.7424, df = 30, p-value = 1.788e-07
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
## -0.8852686 -0.5860994
## sample estimates:
##      cor
## -0.7761684
```

Con este p-value rechazamos H_0

8.1.2 Modelo Lineal

sigamos utilizando los datos de *sim1*

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



Se puede ver un patrón fuerte en los datos. Pareciera que el modelo lineal $y = a_0 + a_1 * x$ podría servir.

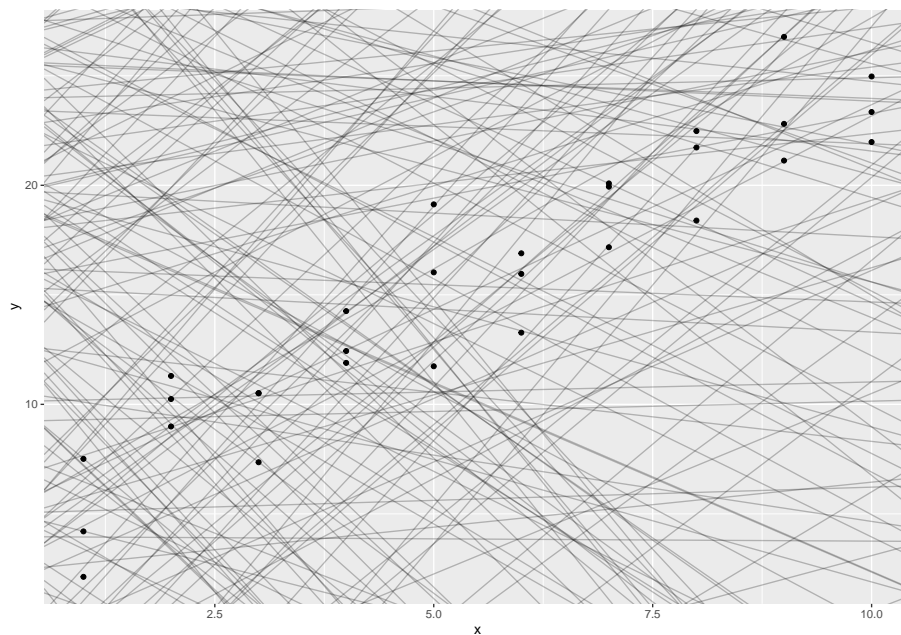
8.1.2.1 Modelos al azar

Para empezar, generemos aleatoriamente varios modelos lineales para ver qué pinta tienen. Para eso, podemos usar `geom_abline()` que toma una pendiente

e intercepto como parámetros.

```
models <- tibble(
  a1 = runif(250, -20, 40),
  a2 = runif(250, -5, 5)
)

ggplot(sim1, aes(x, y)) +
  geom_abline(aes(intercept = a1, slope = a2), data = models, alpha = 1/4) +
  geom_point()
```



A simple vista podemos apreciar que algunos modelos son mejores que otros. Pero necesitamos una forma de cuantificar cuales son los *mejores* modelos.

8.1.2.2 distancias

Una forma de definir *mejor* es pensar en aquel modelo que minimiza la distancia vertical con cada punto:

Para eso, eligamos un modelo cualquiera:

$$y = 7 + 1.5 * x$$

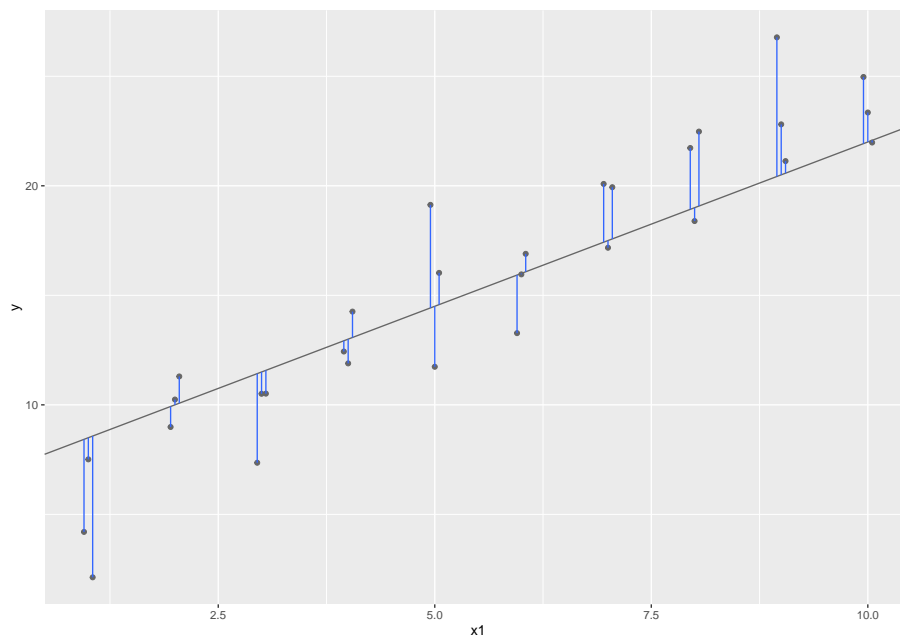
(para que se vean mejor las distancias, corremos un poquito cada punto sobre el eje x)


```

dist1 <- sim1 %>%
  mutate(
    dodge = rep(c(-1, 0, 1) / 20, 10),
    x1 = x + dodge,
    pred = 7 + x1 * 1.5
  )

ggplot(dist1, aes(x1, y)) +
  geom_abline(intercept = 7, slope = 1.5, colour = "grey40") +
  geom_point(colour = "grey40") +
  geom_linerange(aes(ymin = y, ymax = pred), colour = "#3366FF")

```



La distancia de cada punto a la recta es la diferencia entre lo que predice nuestro modelo y el valor real

Para computar la distancia, primero necesitamos una función que represente a nuestro modelo:

Para eso, vamos a crear una función que reciba un vector con los parámetros del modelo, y el set de datos, y genere la predicción:

```

model1 <- function(a, data) {
  a[1] + data$x * a[2]
}

model1(c(7, 1.5), sim1)

```

```
## [1] 8.5 8.5 8.5 10.0 10.0 10.0 11.5 11.5 11.5 13.0 13.0 13.0 14.5 14.5
## [15] 14.5 16.0 16.0 16.0 17.5 17.5 17.5 19.0 19.0 19.0 20.5 20.5 20.5 22.0
## [29] 22.0 22.0
```

Ahora, necesitamos una forma de calcular los residuos y agruparlos. Esto lo vamos a hacer con el error cuadrático medio

$$ECM = \sqrt{\frac{\sum_i^n (\hat{y}_i - y_i)^2}{n}}$$

```
measure_distance <- function(mod, data) {
  diff <- data$y - model1(mod, data)
  sqrt(mean(diff ^ 2))
}

measure_distance(c(7, 1.5), sim1)
```

```
## [1] 2.665212
```

8.1.2.3 Evaluando los modelos aleatorios

Ahora podemos calcular el **ECM** para todos los modelos del dataframe *models*. Para eso utilizamos el paquete **purrr**, para ejecutar varias veces la misma función sobre varios elementos.

Tenemos que pasar los valores de *a1* y *a2* (dos parámetros → *map2*), pero como nuestra función toma sólo uno (el vector *a*), nos armamos una función de ayuda para *wrappear* *a1* y *a2*

```
sim1_dist <- function(a1, a2) {
  measure_distance(c(a1, a2), sim1)
}

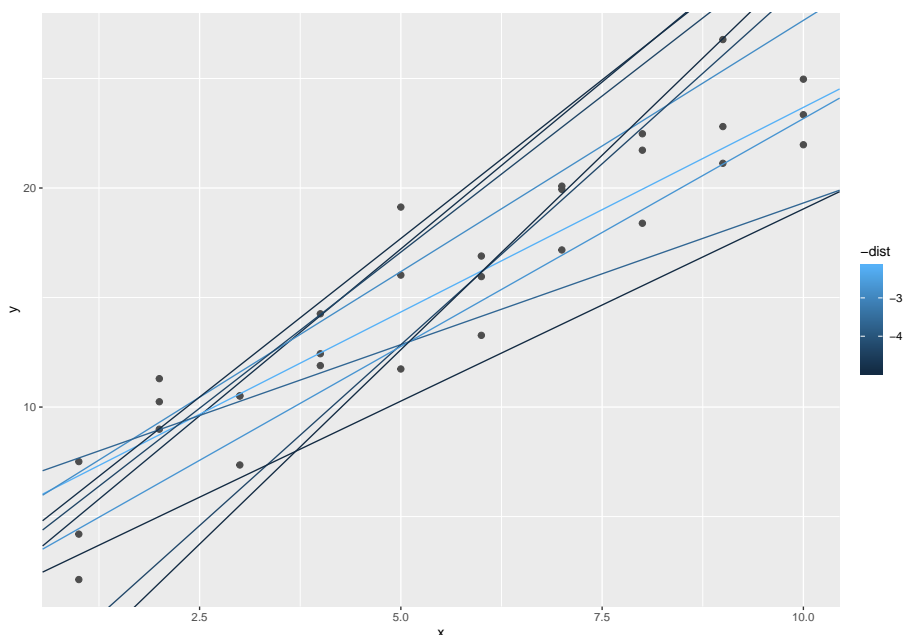
models <- models %>%
  mutate(dist = purrr::map2_dbl(a1, a2, sim1_dist))
models
```

```
## # A tibble: 250 x 3
##       a1      a2 dist
##   <dbl> <dbl> <dbl>
## 1 -18.3  4.74  11.2
## 2  35.8  4.51  45.7
## 3   4.63  1.16   5.59
## 4  37.4  4.65  48.0
## 5  -3.67  3.78   5.64
## 6  11.0 -3.80  30.5
## 7  38.7  0.963 28.7
## 8   2.18 -1.43 23.5
```

```
## 9 -1.37 1.75 7.61
## 10 -17.9 1.38 26.0
## # ... with 240 more rows
```

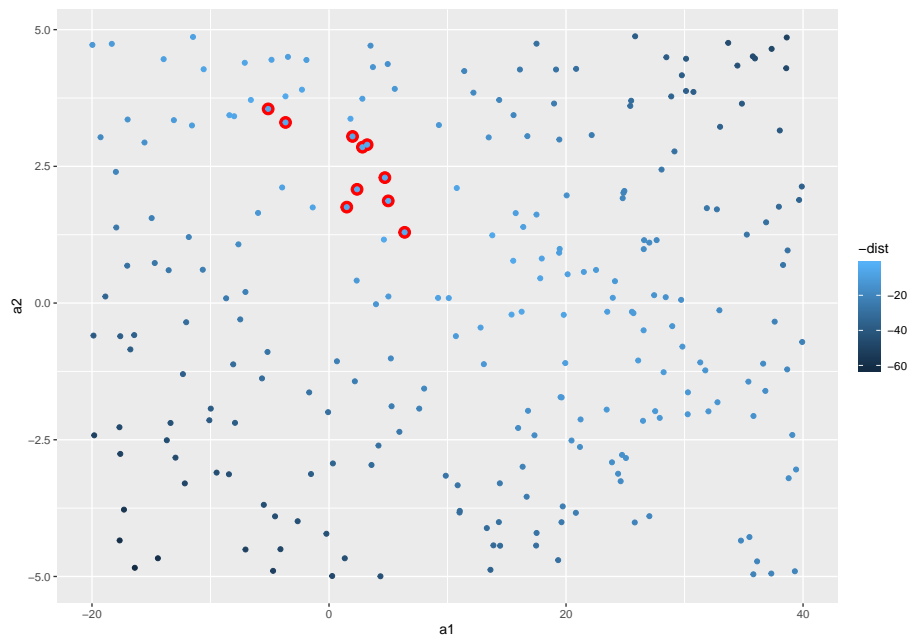
A continuación, superpongamos los 10 mejores modelos a los datos. Coloreamos los modelos por `-dist`: esta es una manera fácil de asegurarse de que los mejores modelos (es decir, los que tienen la menor distancia) obtengan los colores más brillantes.

```
ggplot(sim1, aes(x, y)) +
  geom_point(size = 2, colour = "grey30") +
  geom_abline(
    aes(intercept = a1, slope = a2, colour = -dist),
    data = filter(models, rank(dist) <= 10)
  )
```



También podemos pensar en estos modelos como observaciones y visualizar con un gráfico de dispersión de `a1` vs `a2`, nuevamente coloreado por `-dist`. Ya no podemos ver directamente cómo se compara el modelo con los datos, pero podemos ver muchos modelos a la vez. Nuevamente, destacamos los 10 mejores modelos, esta vez dibujando círculos rojos debajo de ellos.

```
ggplot(models, aes(a1, a2)) +
  geom_point(data = filter(models, rank(dist) <= 10), size = 4, colour = "red") +
  geom_point(aes(colour = -dist))
```

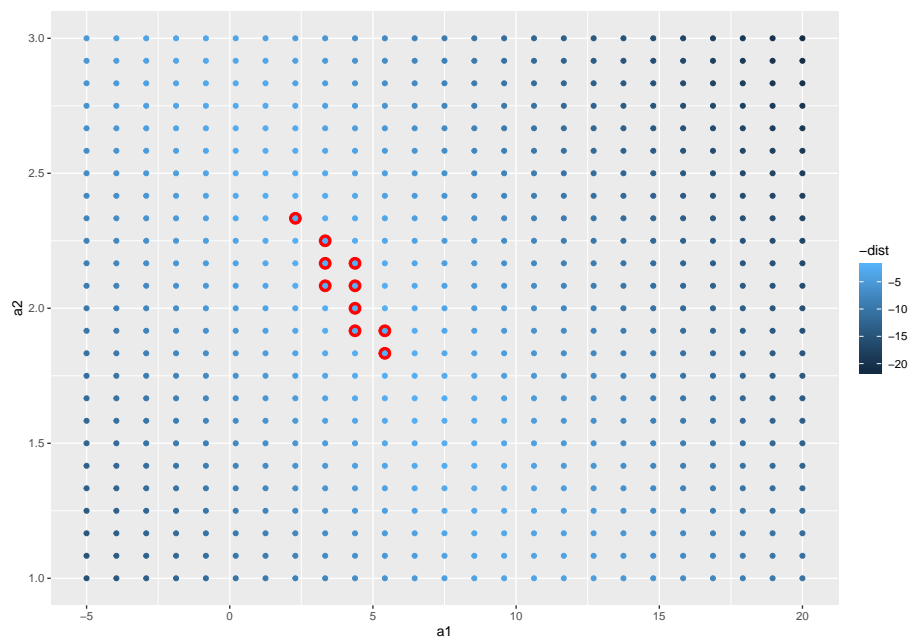


8.1.2.4 Grid search

En lugar de probar muchos modelos aleatorios, podríamos ser más sistemáticos y generar una cuadrícula de puntos uniformemente espaciada (esto se denomina grid search). Elegimos los parámetros de la grilla aproximadamente mirando dónde estaban los mejores modelos en el gráfico anterior.

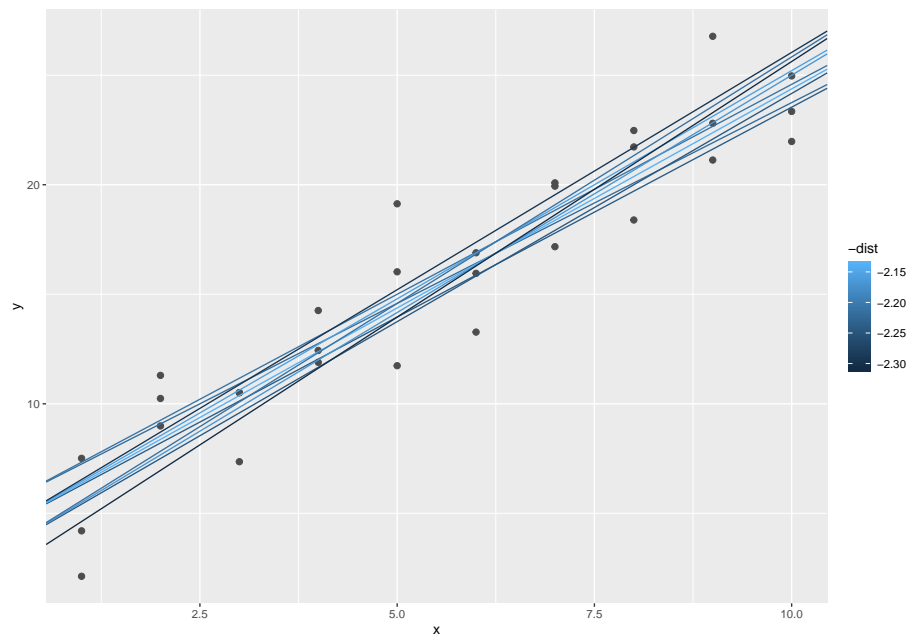
```
grid <- expand.grid(
  a1 = seq(-5, 20, length = 25),
  a2 = seq(1, 3, length = 25)
) %>%
  mutate(dist = purrr::map2_dbl(a1, a2, sim1_dist))

grid %>%
  ggplot(aes(a1, a2)) +
  geom_point(data = filter(grid, rank(dist) <= 10), size = 4, colour = "red") +
  geom_point(aes(colour = -dist))
```



Cuando superponemos los 10 mejores modelos en los datos originales, todos se ven bastante bien:

```
ggplot(sim1, aes(x, y)) +
  geom_point(size = 2, colour = "grey30") +
  geom_abline(
    aes(intercept = a1, slope = a2, colour = -dist),
    data = filter(grid, rank(dist) <= 10)
  )
```



8.1.2.5 óptimo por métodos numéricos

Podríamos imaginar este proceso iterativamente haciendo la cuadrícula más fina y más fina hasta que nos centramos en el mejor modelo. Pero hay una forma mejor de abordar ese problema: una herramienta de minimización numérica llamada búsqueda de **Newton-Raphson**.

La intuición de Newton-Raphson es bastante simple: Se elige un punto de partida y se busca la pendiente más inclinada. Luego, desciende por esa pendiente un poco, y se repite una y otra vez, hasta que no se puede seguir bajando.

En R, podemos hacer eso con `optim()`:

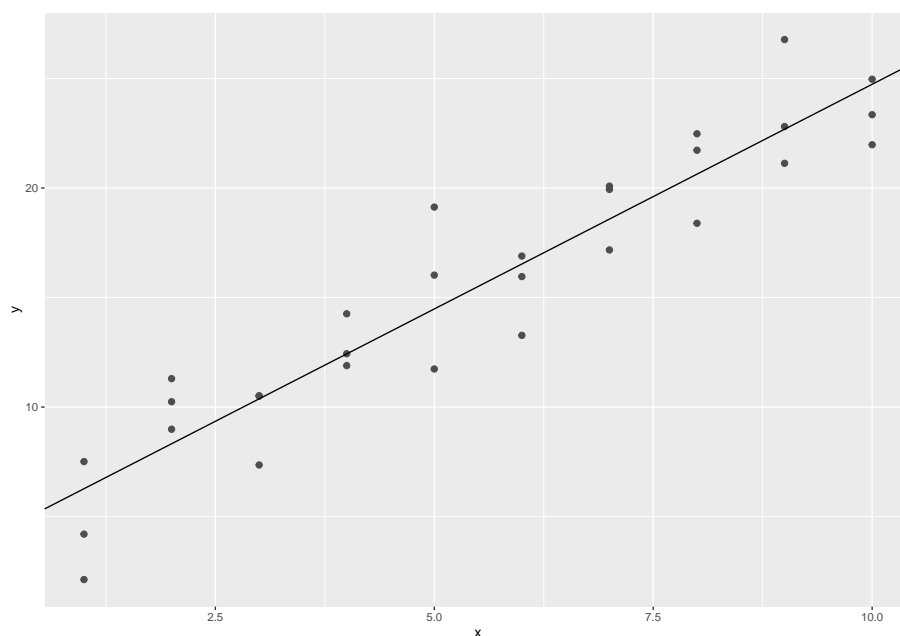
- necesitamos pasarle un vector de puntos iniciales. Elegimos 4 y 2, porque los mejores modelos andan cerca de esos valores
- le pasamos nuestra función de distancia, y los parámetros que nuestra función necesita (`data`)

```
best <- optim(c(4,2), measure_distance, data = sim1)
best
```

```
## $par
## [1] 4.221029 2.051528
##
## $value
## [1] 2.128181
##
```

```
## $counts
## function gradient
##      49      NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
ggplot(sim1, aes(x, y)) +
  geom_point(size = 2, colour = "grey30") +
  geom_abline(intercept = best$par[1], slope = best$par[2])
```



8.1.2.6 Óptimo para el modelo lineal

Este procedimiento es válido para muchas familias de modelos. Pero para el caso del modelo lineal, conocemos otras formas de resolverlo

Si nuestro modelo es

$$y = a_1 + a_2x + \epsilon$$

La solución del óptimo que surge de minimizar el Error Cuadrático Medio es:

$$\hat{a}_1 = \bar{y} - \hat{a}_2 \bar{x}$$

$$\hat{a}_2 = \frac{\sum_i^n (y_i - \bar{y})(x_i - \bar{x})}{\sum_i^n (x_i - \bar{x})^2}$$

R tiene una función específica para el modelo lineal `lm()`. Como esta función sirve tanto para regresiones lineales simples como múltiples, debemos especificar el modelo en las *formulas*: `y ~ x`

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

8.1.2.7 Interpretando la salida de la regresión

```
summary(sim1_mod)

##
## Call:
## lm(formula = y ~ x, data = sim1)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -4.1469 -1.5197  0.1331  1.4670  4.6516
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   4.2208     0.8688   4.858 4.09e-05 ***
## x             2.0515     0.1400  14.651 1.17e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.203 on 28 degrees of freedom
## Multiple R-squared:  0.8846, Adjusted R-squared:  0.8805
## F-statistic: 214.7 on 1 and 28 DF,  p-value: 1.173e-14
```

Analicemos los elementos de la salida:

- **Residuals:** La distribución de los residuos. Hablaremos más adelante.
- **Coefficients:** Los coeficientes del modelo. El intercepto y la variable explicativa
 - *Estimate:* Es el valor estimado para cada parámetro
 - *Pr(>|t|):* Es el *p-valor* asociado al test que mide que el parámetro sea mayor que 0. Si el p-valor es cercano a 0, entonces el parámetro es significativamente mayor a 0.
- **Multiple R-squared:** El R^2 indica que proporción del movimiento en y es explicado por x .

- **F-statistic:** Es el resultado de un test *de significatividad global* del modelo. Con un p-valor bajo, rechazamos la hipótesis nula, que indica que el modelo no explicaría bien al fenómeno.

interpretación de los parámetros: El valor estimado del parámetro se puede leer como “cuanto varía y cuando x varía en una unidad”. Es decir, es la pendiente de la recta

8.1.2.8 Análisis de los residuos

Los residuos del modelo indican cuanto le erra el modelo en cada una de las observaciones. Es la distancia que intentamos minimizar de forma agregada.

Podemos agregar los residuos al dataframe con `add_residuals()` de la librería `modelr`.

```
sim1 <- sim1 %>%
  add_residuals(sim1_mod)

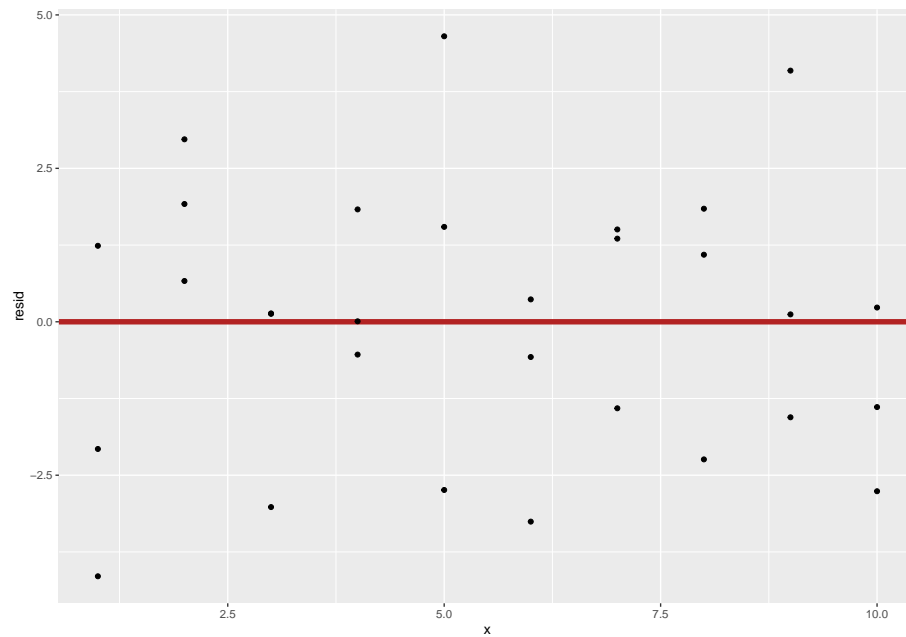
sim1 %>%
  sample_n(10)
```

```
## # A tibble: 10 x 3
##       x     y resid
##   <int> <dbl> <dbl>
## 1     3  7.36 -3.02
## 2     6 16.0 -0.574
## 3     6 16.9  0.365
## 4     1  2.13 -4.15
## 5     6 13.3 -3.26
## 6    10 23.3 -1.39
## 7     2 10.2  1.92
## 8     4 11.9 -0.534
## 9     7 19.9  1.35
## 10    4 14.3  1.83
```

- Si cuando miramos los residuos notamos que **tienen una estructura**, eso significa que nuestro modelo no está bien especificado. En otros términos, nos olvidamos de un elemento importante para explicar el fenómeno.
- Lo que debemos buscar es que los residuos estén homogéneamente distribuidos en torno al 0.

Hay muchas maneras de analizar los residuos. Una es con las estadísticas de resumen que muestra el `summary`. Otra forma es graficándolos.

```
ggplot(sim1, aes(x, resid)) +
  geom_ref_line(h = 0, size = 2, colour = "firebrick") +
  geom_point()
```

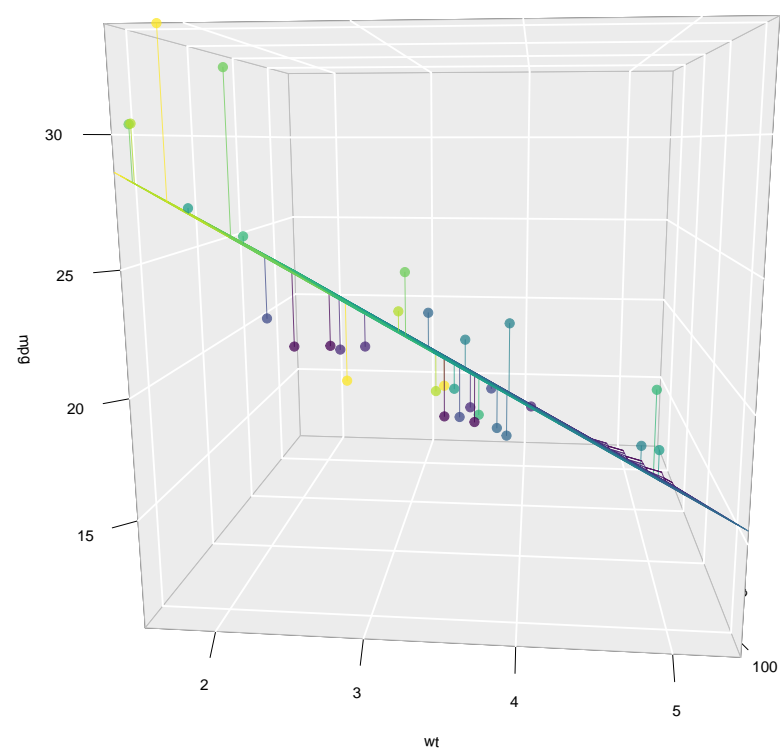


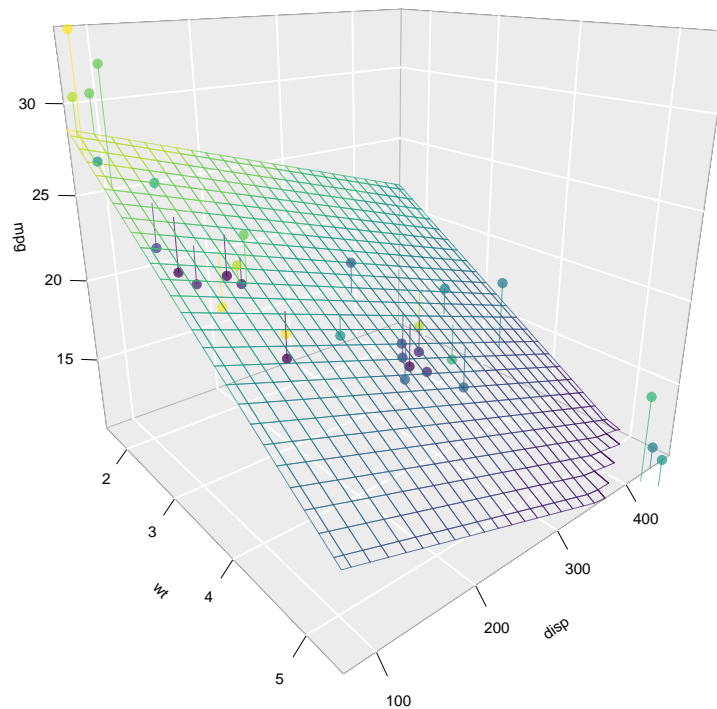
8.1.3 Regresión lineal múltiple

Si bien escapa a los alcances de esta clase ver en detalle el modelo lineal múltiple, podemos ver alguna intuición.

- Notemos que el modelo ya no es una línea en un plano, sino que ahora el modelo es un plano, en un espacio de 3 dimensiones:

Para cada par de puntos en x_1 y x_2 vamos a definir un valor para y





- El criterio para elegir el mejor modelo va a seguir siendo *minimizar las distancias verticales*. Esto quiere decir, respecto de la variable que queremos predecir.
- **interpretación de los parámetros:** El valor estimado del parámetro se puede leer como “cuanto varía y cuando x varía en una unidad, **cuando todo lo demás permanece constante**”. Noten que ahora para interpretar los resultados tenemos que hacer la abstracción de dejar todas las demás variables constantes
- **Adjusted R-squared:** Es similar a R^2 , pero ajusta por la cantidad de variables del modelo (nosotros estamos utilizando un modelo de una sola variable), sirve para comparar entre modelos de distinta cantidad de variables.

8.1.4 Para profundizar

Estas notas de clase estan fuertemente inspiradas en los siguientes libros/notas:

- R para Ciencia de Datos
- Apuntes regresión lineal

Un punto pendiente de estas clases que es muy importante son los **supuestos** que tiene detrás el modelo lineal.

8.2 Práctica Guiada

```
library(tidyverse)
```

8.2.1 Datos de Properati

Para este ejercicio utilizaremos los datos provistos por Properati: <https://www.properati.com.ar/data/>

Primero acondicionamos la base original, para quedarnos con una base más fácil de trabajar, y que contiene unicamente los datos interesantes. (no es necesario correrlo)

```
ar_properties <- read_csv("~/Downloads/ar_properties.csv")
ar_properties %>%
  filter(operation_type=='Venta',
         property_type %in% c('Casa', 'PH', 'Departamento'),
         currency=='USD',
         l1=='Argentina',
         l2=='Capital Federal',
         !is.na(rooms),
         !is.na(surface_total),
         !is.na(surface_covered),
         !is.na(bathrooms),
         !is.na(l3)) %>%
  select(-c(lat,lon, title,description, ad_type,start_date, end_date,operation_type,currency, l1,
            l2,l3))
saveRDS('fuentes/datos_properati.RDS')
```

```
df <- read_rds('fuentes/datos_properati.RDS')
```

```
glimpse(df)
```

```
## Observations: 52,246
```

```
## Variables: 9
```

```
## $ id <chr> "OgLe3YSDR0da+JUQZgmTtA==", "Z3j1BtQN1kzuJr20c..."
```

```
## $ created_on <date> 2019-05-09, 2019-05-09, 2019-05-09, 2019-05-0...
```

```
## $ l3 <chr> "Nuñez", "Nuñez", "Almagro", "Belgrano", "Flor..."
```

```
## $ rooms          <dbl> 3, 3, 3, 5, 5, 3, 3, 2, 5, 5, 4, 2, 3, 5, 3, 3...
## $ bathrooms      <dbl> 1, 1, 1, 2, 2, 1, 1, 1, 2, 1, 1, 1, 2, 4, 2, 3...
## $ surface_total   <dbl> 77, 97, 69, 230, 168, 65, 95, 50, 181, 180, 89...
## $ surface_covered <dbl> 68, 65, 69, 200, 168, 65, 92, 38, 110, 120, 11...
## $ price           <dbl> 180000, 265000, 230000, 380000, 255000, 119000...
## $ property_type   <chr> "PH", "PH", "PH", "PH", "PH", "PH", "PH", "PH"...
```

```
summary(df$price)
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.     Max.
##      6000 119000 170000 251944 272000 6000000
```

```
df[df$price<10000,]
```

```
## # A tibble: 4 x 9
##   id      created_on l3    rooms bathrooms surface_total surface_covered
##   <chr> <date>      <chr> <dbl>      <dbl>          <dbl>          <dbl>
## 1 uZe6~ 2019-03-28 Pale~     5          4            340            320
## 2 +JnI~ 2019-04-01 Parq~     1          1             31             31
## 3 MEQM~ 2019-03-15 Puer~     3          3            275            220
## 4 o6Qf~ 2019-04-30 Reco~     3          2            340            200
## # ... with 2 more variables: price <dbl>, property_type <chr>
```

```
df <- df %>%
  filter(price>10000)
```

Tenemos un par de outliers que no tienen mucho sentido. Es posible que el precio este mal cargado.

```
df[df$price>5000000,]
```

```
## # A tibble: 11 x 9
##   id      created_on l3    rooms bathrooms surface_total surface_covered
##   <chr> <date>      <chr> <dbl>      <dbl>          <dbl>          <dbl>
## 1 ZONE~ 2019-04-13 Reco~     6          3            600            600
## 2 gRZz~ 2019-01-25 Reco~     6          3            600            600
## 3 sP/J~ 2019-05-18 Reco~     8          5            677            568
## 4 VVkm~ 2019-04-05 Reco~    10          3            978            489
## 5 h6gp~ 2019-06-15 Reco~     6          3            600            600
## 6 HWNt~ 2019-06-19 San ~     3          1             60             56
## 7 e2Wf~ 2019-01-28 Pale~     4          4            404            404
## 8 OzkE~ 2019-01-28 Pale~     4          4            404            404
## 9 6DhC~ 2019-02-01 Caba~     1          1             41             37
## 10 Jz4a~ 2019-03-01 Caba~     1          1             41             37
## 11 1R9Q~ 2019-01-17 Caba~     1          1             41             37
## # ... with 2 more variables: price <dbl>, property_type <chr>
```

Los precios más alto tienen algunas cosas sorprendentes, pero sería arriesgado descartarlos por errores.

```
lm_fit <- lm(price ~ l3 + rooms + bathrooms + surface_total + property_type, data = df)
```

```
summary(lm_fit)
```

```
##
## Call:
## lm(formula = price ~ l3 + rooms + bathrooms + surface_total +
##     property_type, data = df)
##
## Residuals:
```

	Min	1Q	Median	3Q	Max
	-2152714	-72322	-4147	46114	5284489

```
##
## Coefficients:
```

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	-1.599e+05	1.388e+04	-11.520	< 2e-16 ***
l3Agronomía	2.024e+04	2.605e+04	0.777	0.437102
l3Almagro	-7.962e+03	1.295e+04	-0.615	0.538764
l3Balvanera	-2.616e+04	1.360e+04	-1.924	0.054387 .
l3Barracas	5.459e+02	1.591e+04	0.034	0.972629
l3Barrio Norte	6.416e+04	1.331e+04	4.821	1.43e-06 ***
l3Belgrano	1.212e+05	1.290e+04	9.396	< 2e-16 ***
l3Boca	-4.934e+04	2.020e+04	-2.443	0.014559 *
l3Boedo	-1.421e+04	1.553e+04	-0.915	0.360345
l3Caballito	6.359e+03	1.298e+04	0.490	0.624125
l3Catalinas	-2.566e+04	1.059e+05	-0.242	0.808533
l3Centro / Microcentro	-3.333e+04	1.969e+04	-1.693	0.090542 .
l3Chacarita	2.843e+04	1.609e+04	1.768	0.077139 .
l3Coghlan	5.894e+04	1.643e+04	3.587	0.000335 ***
l3Colegiales	3.945e+04	1.455e+04	2.710	0.006724 **
l3Congreso	-2.853e+04	1.610e+04	-1.773	0.076275 .
l3Constitución	-2.953e+04	1.767e+04	-1.671	0.094633 .
l3Flores	-2.403e+04	1.363e+04	-1.763	0.077967 .
l3Floresta	-1.220e+04	1.516e+04	-0.804	0.421184
l3Las Cañitas	1.193e+05	1.758e+04	6.785	1.17e-11 ***
l3Liniers	-2.029e+04	1.592e+04	-1.275	0.202348
l3Mataderos	-3.332e+04	1.612e+04	-2.067	0.038736 *
l3Montserrat	-9.560e+03	1.570e+04	-0.609	0.542461
l3Monte Castro	1.875e+04	1.793e+04	1.046	0.295781
l3Nuñez	9.191e+04	1.373e+04	6.695	2.18e-11 ***
l3Once	-2.203e+04	1.598e+04	-1.379	0.168006
l3Palermo	1.276e+05	1.272e+04	10.033	< 2e-16 ***
l3Parque Avellaneda	-1.666e+04	2.199e+04	-0.758	0.448651
l3Parque Centenario	-3.832e+04	1.523e+04	-2.515	0.011903 *
l3Parque Chacabuco	-1.329e+03	1.569e+04	-0.085	0.932517

```

## 13Parque Chas          2.209e+04  2.267e+04   0.975  0.329726
## 13Parque Patricios     -1.126e+04  1.768e+04  -0.637  0.524163
## 13Paternal             -2.778e+03  1.550e+04  -0.179  0.857733
## 13Pompeya              -6.158e+04  2.211e+04  -2.786  0.005340 **
## 13Puerto Madero        5.295e+05  1.457e+04  36.353  < 2e-16 ***
## 13Recoleta             1.294e+05  1.309e+04   9.883  < 2e-16 ***
## 13Retiro                7.507e+04  1.571e+04   4.779  1.76e-06 ***
## 13Saavedra              3.674e+04  1.485e+04   2.473  0.013387 *
## 13San Cristobal        -1.197e+04  1.479e+04  -0.809  0.418323
## 13San Nicolás          -4.616e+03  1.534e+04  -0.301  0.763503
## 13San Telmo             1.763e+04  1.460e+04   1.208  0.227176
## 13Tribunales           -4.234e+04  2.555e+04  -1.657  0.097553 .
## 13Velez Sarsfield       1.664e+03  2.487e+04   0.067  0.946644
## 13Versalles             4.516e+03  1.988e+04   0.227  0.820295
## 13Villa Crespo         1.072e+04  1.303e+04   0.823  0.410681
## 13Villa del Parque      2.440e+04  1.470e+04   1.660  0.096951 .
## 13Villa Devoto          3.089e+04  1.440e+04   2.146  0.031896 *
## 13Villa General Mitre  -2.567e+04  2.024e+04  -1.268  0.204656
## 13Villa Lugano         -1.002e+05  1.749e+04  -5.729  1.02e-08 ***
## 13Villa Luro            7.208e+03  1.617e+04   0.446  0.655849
## 13Villa Ortuzar         2.826e+04  2.042e+04   1.383  0.166525
## 13Villa Pueyrredón      2.686e+04  1.590e+04   1.689  0.091191 .
## 13Villa Real            1.343e+04  2.592e+04   0.518  0.604258
## 13Villa Riachuelo       -5.135e+04  4.988e+04  -1.029  0.303274
## 13Villa Santa Rita      6.264e+03  1.909e+04   0.328  0.742874
## 13Villa Soldati         -9.211e+04  3.636e+04  -2.534  0.011295 *
## 13Villa Urquiza         4.076e+04  1.333e+04   3.058  0.002230 **
## rooms                  5.199e+04  8.989e+02  57.839  < 2e-16 ***
## bathrooms              1.419e+05  1.461e+03  97.128  < 2e-16 ***
## surface_total          5.808e+00  1.141e+00   5.092  3.56e-07 ***
## property_typeDepartamento 4.855e+03  5.267e+03   0.922  0.356609
## property_typePH        -4.780e+04  5.691e+03  -8.399  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 210300 on 52180 degrees of freedom
## Multiple R-squared:  0.4872, Adjusted R-squared:  0.4866
## F-statistic: 812.7 on 61 and 52180 DF, p-value: < 2.2e-16

```

¿ Qué pasó con las variables no numéricas? ¿Son significativos los estimadores? ¿cuales? ¿Cómo se leen los valores de los estimadores?

Dado que muchos de los barrios no explican significativamente los cambios en los precios, no esta bueno conservarlos todos. A su vez, no sabemos respecto a qué barrio se compara.

Una solución puede ser agrupar los barrios en tres categorías respecto a su efecto

en el precio:

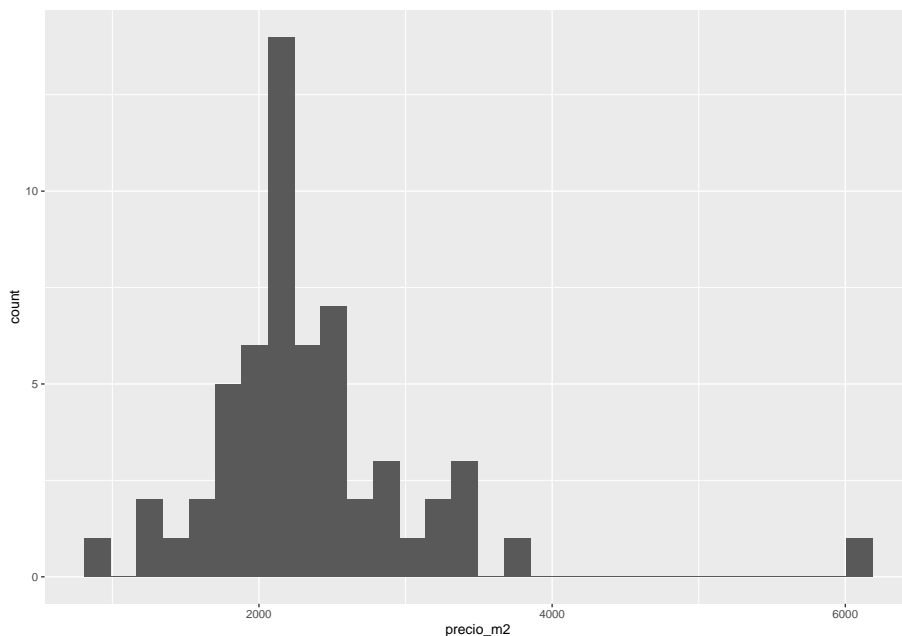
- Alto
- Medio
- Bajo

En particular, podemos notar de esta primera regresión que algunos barrios tienen un efecto significativo en subir el valor de la propiedad, como Belgrano o Recoleta.

Para construir la nueva variable, podemos ver el precio promedio del metro cuadrado por barrio

```
df_barrios <- df %>%
  group_by(l3) %>%
  summarise(precio_m2 = mean(price/surface_total))

ggplot(df_barrios, aes(precio_m2)) +
  geom_histogram()
```



```
summary(df_barrios$precio_m2)
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  871.2  2031.8  2147.3  2346.0  2560.0  6068.5
```

Con este gráfico vemos que hay muchos barrios con un precio promedio cercano a 2500 dólares el m^2 .

Podemos dividir los tres grupos al rededor de los cuartiles 1 y 3.

- <2000 bajo
- 2000-2500 medio
- 2500 alto

```
df_barrios <- df_barrios %>%
  mutate(barrio= case_when(precio_m2<2000 ~ 'bajo',
                             precio_m2>2000 & precio_m2<2500 ~ 'medio',
                             precio_m2>2500 ~ 'alto'))

df_barrios %>%
  sample_n(10)
```

```
## # A tibble: 10 x 3
##   l3          precio_m2 barrio
##   <chr>          <dbl> <chr>
## 1 Villa Riachuelo    1479. bajo
## 2 Tribunales        2238. medio
## 3 San Nicolás       2439. medio
## 4 Parque Chacabuco  1938. bajo
## 5 Villa Pueyrredón  2292. medio
## 6 Villa Luro        2147. medio
## 7 Caballito         2687. alto
## 8 Parque Avellaneda 1616. bajo
## 9 Parque Patricios  1925. bajo
## 10 Barrio Norte     3221. alto
```

Con esta nueva variable podemos modificar la tabla original.

```
df <- df %>%
  left_join(df_barrios, by='l3')
```

y volvemos a calcular el modelo

```
lm_fit <- lm(price~ barrio+ rooms + bathrooms + surface_total + property_type,data = df)

summary(lm_fit)

##
## Call:
## lm(formula = price ~ barrio + rooms + bathrooms + surface_total +
##     property_type, data = df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2145645  -71277  -11187   42472  5307946
##
```

```
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   -1.041e+05  6.349e+03 -16.396 < 2e-16 ***
## barriobajo    -1.097e+05  4.231e+03 -25.939 < 2e-16 ***
## barriomedio   -9.342e+04  2.150e+03 -43.445 < 2e-16 ***
## rooms         4.808e+04  9.293e+02  51.732 < 2e-16 ***
## bathrooms     1.602e+05  1.499e+03 106.867 < 2e-16 ***
## surface_total  5.485e+00  1.198e+00   4.580 4.67e-06 ***
## property_typeDepartamento 2.370e+04  5.352e+03   4.428 9.51e-06 ***
## property_typePH  -3.906e+04  5.920e+03  -6.598 4.22e-11 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 220800 on 52234 degrees of freedom
## Multiple R-squared:  0.4338, Adjusted R-squared:  0.4338
## F-statistic: 5718 on 7 and 52234 DF,  p-value: < 2.2e-16
```

Si queremos que compare contra 'barrio medio' podemos convertir la variable en factor y explicitar los niveles

```
df <- df %>%
  mutate(barrio = factor(barrio, levels = c('medio', 'alto', 'bajo')))

lm_fit <- lm(price ~ barrio + rooms + bathrooms + surface_total + property_type, data = df)

summary(lm_fit)
```

```
##
## Call:
## lm(formula = price ~ barrio + rooms + bathrooms + surface_total +
##     property_type, data = df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2145645   -71277   -11187    42472   5307946
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   -1.975e+05  6.215e+03 -31.783 < 2e-16 ***
## barriobajo    -1.632e+04  4.321e+03  -3.777 0.000159 ***
## barriomedio   -9.342e+04  2.150e+03 -43.445 < 2e-16 ***
## barriobajo    -1.632e+04  4.321e+03  -3.777 0.000159 ***
## rooms         4.808e+04  9.293e+02  51.732 < 2e-16 ***
## bathrooms     1.602e+05  1.499e+03 106.867 < 2e-16 ***
## surface_total  5.485e+00  1.198e+00   4.580 4.67e-06 ***
## property_typeDepartamento 2.370e+04  5.352e+03   4.428 9.51e-06 ***
## property_typePH  -3.906e+04  5.920e+03  -6.598 4.22e-11 ***
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 220800 on 52234 degrees of freedom
## Multiple R-squared:  0.4338, Adjusted R-squared:  0.4338
## F-statistic: 5718 on 7 and 52234 DF,  p-value: < 2.2e-16
```

8.2.1.1 Feature engineering.

Lo que hicimos arriba con los barrios se conoce como feature engineerin: Generamos una nueva variable a partir de las anteriores para mejorar nuestro modelo.

¿Qué otras modificaciones podemos hacer?

- Hay una que ya hicimos: En lugar de pensar en el precio total, podemos pensar en el precio por m^2 . De esta manera ya no tendría sentido agregar la variable `surface_total`

```
lm_fit <- lm(precio_m2 ~ barrio + rooms + bathrooms + property_type, data = df)
summary(lm_fit)
```

```
##
## Call:
## lm(formula = precio_m2 ~ barrio + rooms + bathrooms + property_type,
##     data = df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2071.97  -241.41    55.51   214.05  2993.52
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)    1935.419     13.106  147.670 < 2e-16 ***
## barrioalto       892.491      4.535  196.790 < 2e-16 ***
## barriobajo     -461.046      9.112  -50.595 < 2e-16 ***
## rooms          -22.684      1.959  -11.579 < 2e-16 ***
## bathrooms       133.009      3.161   42.084 < 2e-16 ***
## property_typeDepartamento  227.481     11.285   20.158 < 2e-16 ***
## property_typePH       99.545     12.485    7.973 1.58e-15 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 465.7 on 52235 degrees of freedom
## Multiple R-squared:  0.552, Adjusted R-squared:  0.5519
## F-statistic: 1.073e+04 on 6 and 52235 DF,  p-value: < 2.2e-16
```

que pasó con rooms?

Al normalizar el precio por los metros, rooms pasa de tomar valores positivos

a negativos. Eso significa que rooms estaba correlacionado con el tamaño, y por lo tanto cuantos más cuartos, mayor el valor. Al normalizar podemos ver que, dado un metraje, más cuartos reducen el precio: Preferimos ambientes más grandes tal vez?

predecir

Para predecir un nuevo caso, podemos construir un dataframe con las variables. Por ejemplo

```
caso_nuevo <- tibble(barrio='alto',  
  rooms=3,  
  bathrooms=2,  
  property_type='Departamento',  
  surface_total=78)  
  
predict(lm_fit,newdata = caso_nuevo)
```

```
##          1  
## 3253.356
```

Pero debemos recordar que este es el valor por metro cuadrado. Para obtener lo que realmente nos interesa, tenemos que hacer el camino inverso del feature engineering:

```
predict(lm_fit,caso_nuevo)*caso_nuevo$surface_total
```

```
##          1  
## 253761.8
```

8.2.1.2 Para seguir practicando

Un problema de lo que vimos en esta práctica es que las salidas de `summary(lm_fit)` es una impresión en la consola. Es muy difícil seguir trabajando con esos resultados. Para resolver esto hay un par de librerías que incorporan el modelado lineal al flujo del tidyverse:

- Broom
- Modelr

Chapter 9

Diseño y análisis de encuestas

- Introducción al diseño de encuestas
- Presentación de la Encuesta Permanente de Hogares
- Generación de estadísticos de resumen en muestras estratificadas
- Utilización de los ponderadores

9.1 Explicación

9.2 Práctica Guiada