Notas Curso Básico R

Tobías Chavarría

Actualizado el 01 Mar, 2021

Índice general

Capítulo 1

Introducción

Este documento contiene las bases del lenguaje de programación **R** para poder empezar a realizar análisis de datos, se va a iniciar con la instalación y las configuraciones básicas, y luego avanzaremos a los principios básicos del lenguaje y las herramientas necesarias que nos permitan realizar un análisis exploratorio de un conjunto de datos, construir funciones básicas y realizar visualizaciones.

Capítulo 2

Primeros pasos con R y RStudio

R es un entorno y lenguaje de programación con un enfoque al análisis estadístico

RStudio es un IDE por sus siglas en ingles Integrated Development Environment o Entorno De Desarrollo Integrado que facilita la interacción con el lenguaje de programación R y los procesos de carga de datos, instalación y administración de paquetes, exportación de gráficos y administración de archivos, entre otros.

El objetivo de este capítulo es conocer el entorno de trabajo que proporciona R y RStudio, además de aprender a instalar y cargar los paquetes que se necesiten para realizar análisis de datos.

2.1. Instalación

2.1.1. R

Para instalar R en Windows, la forma más simple es descargar la versión más reciente de R base desde el siguiente enlace de CRAN:

https://cran.r-project.org/bin/windows/base/

El archivo que necesitamos tiene la extensión **.exe** (por ejemplo 4.0.2-win.exe). Una vez descargado, lo ejecutamos como cualquier instalable.

Después de la instalación, estamos listos para usar R.

2.1.2. RStudio

Para instalar RStudio, es necesario descargar y ejecutar alguno de los instaladores disponibles en su sitio oficial. Están disponibles versiones para Windows, OSX y Linux.

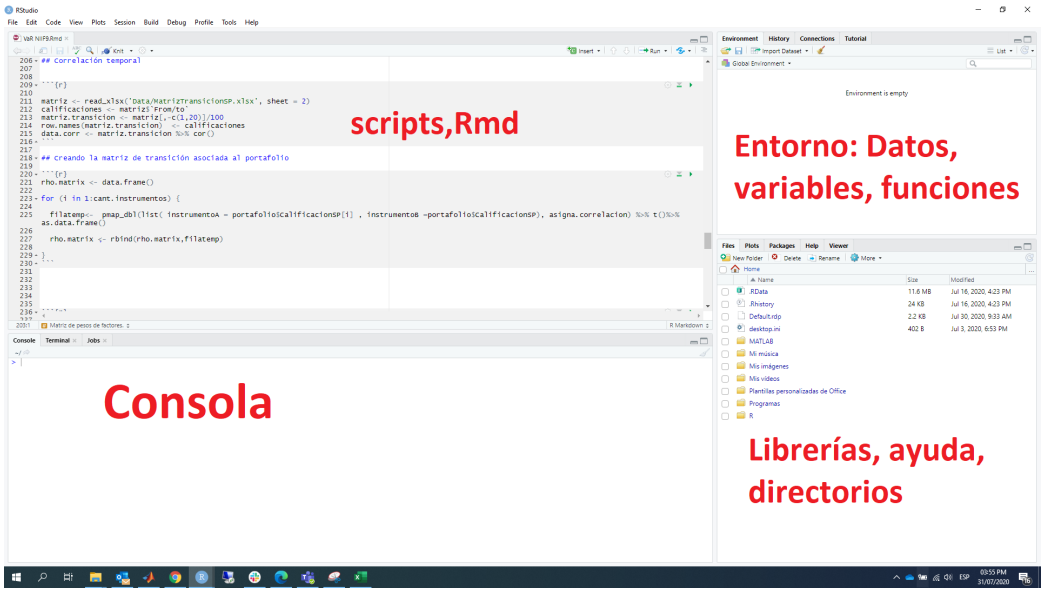
https://www.rstudio.com/products/rstudio/download/

Si ya hemos instalado R en nuestro equipo, RStudio lo detectará automáticamente y podremos utilizarlo desde este entorno. Si no instalamos RStudio antes que R, no hay problema, cada vez que iniciamos este programa, verificará la instalación de R.

2.2. Entorno de trabajo de RStudio.

En general se trabaja con la interfaz de RStudio antes que con la de R porque la primera es mucho "más amigable."

Al abrir RStudio veremos algo como esto:



Una vez estamos en RStudio, podemos escribir y ejecutar las órdenes de varias formas:

- Directamente en la consola
- A través de un script (.R)
- Con ficheros Rmarkdown (.Rmd)

2.2.1. Consola de R

La consola de RStudio nos permite interactuar con los comandos de R, es decir, ingresamos una instrucción en la consola y esta retornará el resultado de la ejecución de ese comando, aunque esta es una herramienta muy útil no es la mejor opción cuando nuestro código gana complejidad.

En la consola escribimos **expresiones**, el símbolo "<-" es el operador de asignación, aunque también se puede utilizar el símbolo "=."

Asignación de valores.

```
x <- 1  # Asignamos el valor 1 a la variable x

texto <- "Bienvenidos"  # Asignamos el valor "Bienvenidos" a la variable texto
```

En R el símbolo "#" indica que es un comentario, cualquier cosa que esté a su derecha (incluido el "#") será ignorado a la hora de ejecutar el código. Este es el único símbolo para hacer comentarios en R y además cabe mencionar que R no soporta comentarios en bloques o multilíneas.

Evaluación.

Cuando escribimos una expresión en la consola, podemos imprimir su valor sin una orden explícita.

```
x <- 13 # No imprime nada, solo asigna el valor
```

```
x # Se imprime el valor
```

```
## [1] 13
```

```
print(x) # Orden explicita
```

```
## [1] 13
```

2.2.2. Ayuda en R

Al comenzar a trabajar con R necesitaremos información sobre cada instrucción, función y paquete. Toda la documentación se encuentra integrada en RStudio, para accesar a esta información podemos usar la función **help()** o el signo de interrogación **?**, de la siguiente manera

```
help("funcion")

?funcion
??nombre_paquete
```

Al ejecutar estas instrucciones la información aparece en la pestaña de **help**.

```
help("read.table")

?read.table
```

2.2.3. Nombres en R

Al igual que la documentación de nuestro código, es importante el nombre que le demos a nuestros objetos (variables, funciones). En **R** los nombres de los objetos deben comenzar con una letra y solo pueden contener letras, números y los signos : " ", "." *Es bueno que los nombres sean descriptivos, es necesario adoptar una convención, la más común es la del guión bajo (snake_case) en la que los nombres se escriben en minúscula y separados por* .

```
yo_uso_guion_bajo  ## snake_case

OtraGenteUsaMayusculas

algunas.personas.usan.puntos ## Esto es peculiar de R, ya que en otros lenguajes el
## ya que tiene otras funciones

Y_algunasPocas.Personas_RENIEGANdelasconvenciones
```

Generalmente las variables son sustantivos y el nombre de las funciones verbos, se debe procurar que los nombres sean concisos y con significado.

```
## Correcto

dia_uno <- 10

## Incorrecto

primer_dia_del_mes <- 10
```

También se debe evitar utilizar nombres de funciones o variables comunes, esto causa confusión al leer el código.

```r
## Incorrecto

T <- FALSE
c <- 10

mean <- function(x) {
  sum(x)
}
```

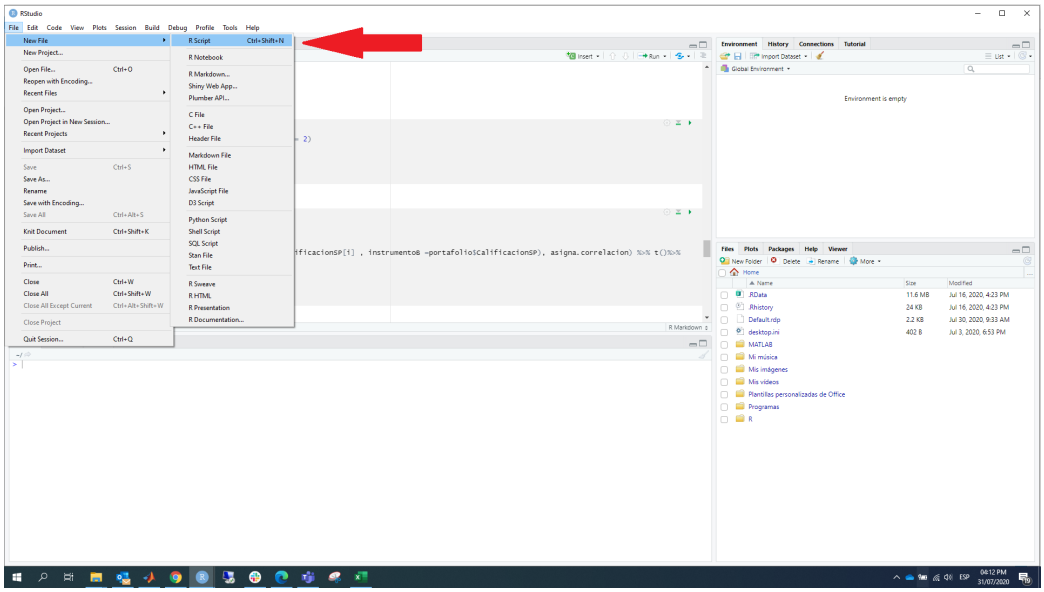
Existen muchas otras buenas prácticas a la hora de escribir código en **R**, el siguiente link contiene una guía del estilo *tidyverse*.

https://style.tidyverse.org/index.html

2.2.4. Scripts de R

Trabajar en la consola es muy limitado ya que las instrucciones se tienen que escribir una por una. Lo habitual es trabajar con scripts o ficheros de instrucciones. Estos ficheros tienen extensión **.R**.

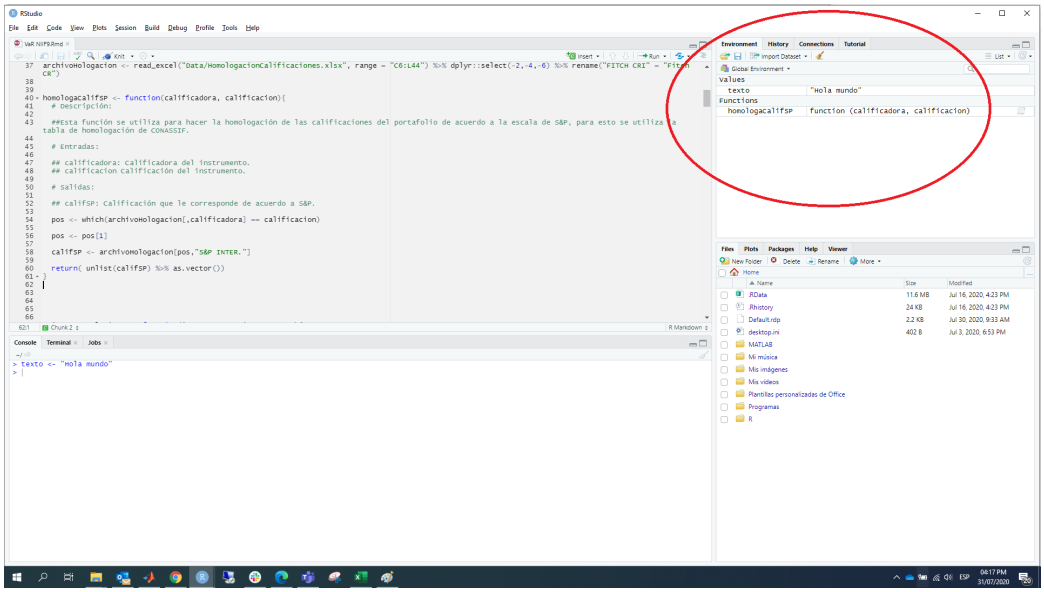
Se puede crear una script con cualquier editor de texto, pero nosotros lo haremos desde RStudio. Para hacer esto, seleccionamos la siguiente ruta de menús: File > New File > R script



2.2.5. Entorno

El panel de entorno esta compuesto de dos pestañas: Environment y History.

En el entorno se irán registrando los objetos que vayamos creando en la sesión de trabajo: datos, variables, funciones. También tenemos la opción de cargar y guardar una sesión de trabajo, importar datos y limpiar los objetos de la sesión. Estas opciones están accesibles a través de las de opciones de la pestaña.



2.2.6. Directorio de trabajo

El directorio o carpeta de trabajo es el lugar en la computadora en el que se encuentran los archivos con los que se van a trabajar en R. Este es el lugar donde R buscara archivos para importarlos y al que serán exportados, a menos que indiquemos otra cosa.

Para encontrar cuál es el directorio de trabajo actual se utiliza la función **getwd()**.

```
getwd()
```

```
## [1] "/Users/tchavarria/Documents/GitHub/programacion-r-basico"
```

Se mostrará en la consola la ruta del directorio que está usando R.

Se puede cambiar el directorio de trabajo usando la función setwd(), dando como argumento la ruta del directorio que se desea utilizar.

```
setwd("otra_ruta")
```

2.3. Paquetes

Cada paquete es una colección de funciones diseñadas para atender una tarea específica. Por ejemplo, hay paquetes para trabajo visualización, conexiones a bases de datos, minería de datos, interacción con servicios de internet, entre otros.

Estos paquetes se encuentran alojados en CRAN, así que pasan por un control riguroso antes de estar disponibles para su uso generalizado.

Se pueden instalar paquetes usando la función **install.packages()**, dando como argumento el nombre del paquete que deseamos instalar, entre comillas.

Por ejemplo, para instalar el paquete **dplyr**, ejecutamos lo siguiente.

```
install.packages("dplyr") ## En general se escribe  install.packages("nombre_paquete
```

Después de ejecutar esa instrucción, aparecerán algunos mensajes en la consola mostrando el avance de la instalación

Una vez concluida la instalación de un paquete, para poder utilizar sus funciones debemos ejecutar la función **library()** con el nombre del paquete que se quiere utilizar.

```
library(dplyr)   ## En general se escribe  library("nombre_paquete")
```

2.4. Scripts

Los scripts son documentos de texto con la extensión de archivo .R, por ejemplo mi_script.R.

Estos archivos son iguales a cualquier documentos de texto, pero R los puede leer y ejecutar el código que contienen.

Aunque R permite el uso interactivo, es recomendable guardar el código en un archivo .R, de esta manera se puede utilizar después y compartirlo con otras personas. En general, en proyectos complejos, es posible que sean necesarios múltiples scripts para distintos fines.

Se pueden abrir y ejecutar scripts en R usando la función **source()**, esta recibe como argumento la ruta del archivo .R en nuestra computadora, entre comillas.

Por ejemplo.

```
source("C:/Proyecto/limpiezaDatos.R")
```

Cuando usamos RStudio y abrimos un script con extensión .R, este programa abre un panel en el que se puede ver su contenido. De este modo se puede ejecutar todo el código que contiene o sólo partes de él.

2.5. Shortcuts

- Borrar toda la consola: CTRL + L.
- Ejecutar una línea o lo que se seleccione: CTRL+R

Capítulo 3

Objetos en R.

En R tenemos 5 clases de objeto básicos o atómicos:

- character
- numeric
- integer
- complex
- logical (TRUE/FALSE)

```
tipo.bien <- "Vivienda"   ## character

saldo <- 130500.34        ## numeric

meses <- 13               ## numeric

dias.mora <- 10L          ## integer

complejo <- 1 + 3i         ## complex

cobro.judicial <- TRUE    ## logical
```

Números.

- En R los número en general se tratan como objetos numeric (i.e números reales de doble precisión.)

- Existe el valor **Inf** que representa infinito y se asocia a operaciones como : 1/0.

```
1 / 0
```

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

```
-1 / 0
```

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

```
100 / Inf
```

```
## [1] 0
```

- El valor **NaN** significa not a number, este se asocia generalmente a datos ausentes pero también a una operación del tipo 0/0 que no está definida.

Atributos

Los objetos en R pueden tener los siguientes atributos

- names, dimnames (matrices, data frames)
- dimension (matrices, data frames)
- class
- length

más adelante veremos que con detalle el uso de estos.

Capítulo 4

Operadores

Operadores aritméticos

En R tenemos los siguientes operadores aritméticos:

Operador	Operación	Ejemplo	Resultado
+	Suma	3+1	4
-	Resta	4-6	-2
	Multiplicación	4*6	24
/	División	14/5	2.8
^	Potencia	2^3	8
%%	División entera	5%%2	1

Operadores relacionales

Los operadores relacionales son usados para hacer comparaciones y siempre devuelven como resultado TRUE o FALSE (verdadero o falso, respectivamente).

Operador	Operación	Ejemplo	Resultado
<	Menor estricto	10 < 3	FALSE
<=	Menor o igual	10 <= 3	FALSE
>	Mayor estricto	10 > 3	TRUE
>=	Mayor o igual	10 >= 3	TRUE
==	Igual	10 == 3	FALSE

Operador	Operación	Ejemplo	Resultado
!=	Distinto	10 != 3	TRUE

Operadores lógicos

Los operadores lógicos son usados para operaciones de álgebra Booleana, es decir, para describir relaciones lógicas, expresadas como verdadero (TRUE) o falso (FALSO).

Operador	Operación
|	or
&	and (conjunción)
!	negación

Los operadores | y & siguen estas reglas:

- | devuelve TRUE si alguno de los datos es TRUE
- & solo devuelve TRUE si ambos datos es TRUE
- | solo devuelve FALSE si ambos datos son FALSE
- & devuelve FALSE si alguno de los datos es FALSE

```
edad <- 16
notas <- 83

beca1 <- (edad > 18 & notas > 80)

beca1
```

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

```
beca2 <- (edad > 18 | notas > 80)

beca2
```

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

Capítulo 5

Estructuras de datos.

Las estructuras de datos básicas de R se pueden agrupar por su dimensionalidad y según si son homogéneas (todos los elementos son del mismo tipo) o heterogéneas (hay elementos de distintos tipos). En el siguiente cuadro se resumen estas:

Dimensión	Homogéneas	Heterogéneas
1d	Vector	Lista
2d	Matriz	Data Frame

5.1. Vectores

Los vectores en R son fundamentales ya que, es una de las estructuras de datos más utilizada, la propiedad más importante de los vectores, es que **solo pueden contener objetos de la misma clase**.

Vectores vacíos pueden crearse con la función **vector()**, por ejemplo:

```
vector("numeric", length = 10) ## Tiene los parámetros clase y longitud.
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

```
vector("character", 3)
```

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

Sin embargo, la forma más común para crear vectores es utilizando la función **c()** que hace referencia a la palabra concatenar.

```
vector.numerico <- c(1, 2, 3.5)   ## numeric

vector.logical <- c(TRUE, FALSE, T, T, F) ## logical

vector.char <- c("Azul", "Blanco", "Verde") ## character

vector.entero <- 1:13   ## integer
```

```
vector.numerico
```

```
## [1] 1.0 2.0 3.5
```

```
vector.logical
```

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

```
vector.char
```

```
## [1] "Azul"   "Blanco" "Verde"
```

```
vector.entero
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13
```

5.1.1. Coerción

Como dijimos anteriormente los vectores solo contienen una misma clase de datos, sin embargo, cuando mezclamos diferentes clases ocurre automáticamente un proceso que se llama **coerción**, esto básicamente transforma todos los elementos del vector a una misma clase.

```
vector.prueba1 <- c(1.3, "Diego") # character
vector.prueba1
```

```
## [1] "1.3"   "Diego"
```

```
vector.prueba2 <- c(TRUE, 13, FALSE) # numeric  #TRUE =1 , # FALSE =0
vector.prueba2
```

```
## [1]  1 13  0
```

Una función muy útil es **length** ya que nos devuelve el tamaño de nuestro vector.

```
length(vector.prueba2)
```

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

Reglas de coerción:

- Valores lógicos se convierten en numéricos: TRUE = 1, FALSE=0.
- El orden de coerción es el siguiente:
    - logical -> integer -> numeric -> character

Es decir todos los elementos del vector se van a transformar a la clase correspondiente siguiendo el orden anterior.

```
vector.prueba3 <- c(12, TRUE, "Azul")

vector.prueba3
```

```
## [1] "12"   "TRUE" "Azul"
```

Esto es lo que hace R de forma automática, sin embargo, podemos realizar la coerción de forma explícita utilizando la función **as.***

```
# Vector numerico 0,1,2,3,4,5.
x <- 0:5
# Se transforma a clase logical
as.logical(x) # 0 es FALSE, y cualquier otro número es TRUE.
```

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

```
# Se transforma a clase character
as.character(x)
```

```
## [1] "0" "1" "2" "3" "4" "5"
```

```
x <- as.character(x)
x
```

```
## [1] "0" "1" "2" "3" "4" "5"
```

Al hacer la coerción de forma explícita es común obtener el siguiente warning:

"Nas introduced by coercion"

Esto significa que dentro del vector hay valores que no tiene sentido convertirlos a la clase que queremos, por ejemplo, convertir

```
y <- c("A", "B", "C")
as.numeric(y)
```

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

Tal vez en el ejemplo anterior no tiene mucho sentido, sin embargo puede pasar que haya una variable que debería contener solo números, pero aparece una letra por error.

```
y <- c(1:3, "A", 5:7)
y
```

```
## [1] "1" "2" "3" "A" "5" "6" "7"
```

```
as.numeric(y)
```

```
## [1]  1  2  3 NA  5  6  7
```

5.2. Matrices

Las matrices son vectores pero con el atributo de dimensión. Este atributo es un vector de longitud 2 que contiene el número de filas y el número de columnas (nrow,ncol).

En R la función para crear matrices es **matrix** recibe cuatro parámetros (dos son opcionales)

- data : vector con los valores que contendrá la matriz.
- nrow : cantidad de filas de la matriz.
- ncol : cantidad de columnas de la matriz.
- byrow : si su valor es TRUE la lectura de los datos se realiza por filas sino se realiza por columnas.

```
matriz1 <- matrix(nrow = 2, ncol = 3) # No le doy los valores entonces coloca

matriz1
```

```
##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]   NA   NA   NA
```

Las matrices se construyen por defecto por columnas, empezando por el valor de la entrada $(1, 1)$.

```
matriz1 <- matrix(1:9, nrow = 3, ncol = 3)
matriz1
```

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

Esta función nos devuelve la cantidad de filas y columnas de nuestra matriz.

```
dimensiones <- dim(matriz1)

dimensiones
```

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

Es muy común utilizar solo uno de estos valores, estos se pueden obtener mediante la siguiente instrucción.

```
filas <-  dimensiones[1]
columnas <- dimensiones[2]

filas
```

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

```
columnas
```

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

Sin embargo se puede cambiar a que se construyan por filas asignando el parámetro **byrow** en TRUE.

```
matriz2 <- matrix(1:9, nrow = 3, ncol = 3, byrow = T)
matriz2
```

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

Un par de funciones útiles a la hora de crear matrices o conjuntos de datos son **rbind** y **cbind** que permiten concatenar objetos en forma de filas y columnas respectivamente.

```
x <- 1:5

y <- 11:15

rbind(x, y)
```

```
##   [,1] [,2] [,3] [,4] [,5]
## x    1    2    3    4    5
## y   11   12   13   14   15
```

```
cbind(x, y)
```

```
##      x  y
## [1,] 1 11
## [2,] 2 12
## [3,] 3 13
## [4,] 4 14
## [5,] 5 15
```

5.3. Data Frames

Los data frames constituyen la manera más eficiente mediante la cual R puede analizar un conjunto de datos estadísticos.

Habitualmente se configuran de tal manera que **cada fila se refiere a un individuo o unidad estadística, mientras que cada columna hace referencia a una variable estadística**, esa configuración hace que visualmente un data frame parezca una matriz. Sin embargo, como objetos de R, son cosas distintas.

- Los data frames tienen los atributos **row.names** y **col.names**.
- Usualmente se crean leyendo datos con las funciones **read.table()** y **read.csv()**, pero también podemos utilizar la función **data.frame()**.
- Se pueden convertir a matrices utilizando la función **data.matrix()** o **as.matrix()**.

```
# Creamos las variables de nuestro data frame.

emisor <- c("BL", "BARCL", "CSGF", "CVS", "DBK", "G", "NOMUR", "USTES", "BNSFI

monto.facial <- c(5000000, 2500000, 10000000, 40000000, 5000000, 40000000, 100

categoria <- c("COSTO AMORTIZADO", "COSTO AMORTIZADO", "COSTO AMORTIZADO", "VR

calificacion.SP <- c("BBB+", "AA+", "B+", "B+", "B", "B", "A+", "BB-", "BB")

isin <- c("US06738EAL92", "US225433AH43", "US126650CK42", "XS2127535131", "CRG


# Creamos el data frame.

portafolio <- data.frame(ISIN = isin, Emisor = emisor, Monto.Facial = monto.fa


portafolio
```

```
##           ISIN Emisor Monto.Facial       Categoria_NIIF Calificacion
## 1 US06738EAL92     BL      5.0e+06     COSTO AMORTIZADO         BBB+
## 2 US225433AH43  BARCL      2.5e+06     COSTO AMORTIZADO          AA+
## 3 US126650CK42   CSGF      1.0e+07     COSTO AMORTIZADO           B+
## 4 XS2127535131    CVS      4.0e+07 VR CON CAMBIO EN ORI           B+
## 5 CRG0000B82H3    DBK      5.0e+06     COSTO AMORTIZADO            B
## 6 US404280BJ78      G      4.0e+07 VR CON CAMBIO EN P/G            B
## 7 ONRBNCR00465  NOMUR      1.0e+07     COSTO AMORTIZADO           A+
## 8 US9127962Z13  USTES      5.6e+06     COSTO AMORTIZADO          BB-
## 9 US9128283G32  BNSFI      5.0e+07     COSTO AMORTIZADO           BB
```

```
dim(portafolio)
```

```
## [1] 9 5
```

5.4. Listas

Con los data frames vimos que se pueden guardar diferentes tipos de datos en columnas.

Ahora queremos ir un poco más allá y guardar diferentes objetos en una misma estructura de datos.

Las listas permiten agrupar o contener cosas como dataframes, matrices y vectores en una misma variable.

Para crear una lista podemos utilizar la función **list()**, por ejemplo

```
lista1 <- list(1, "A", TRUE) # integer, character, logical
lista1
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "A"
##
```

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

```
mi_vector <- 1:10
mi_matriz <- matrix(1:4, nrow = 2)

mi_dataframe <- data.frame("numeros" = 1:3, "letras" = c("a", "b", "c"))

mi_lista <- list("un_vector" = mi_vector, "una_matriz" = mi_matriz, "un_df" =

mi_lista
```

```
## $un_vector
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $una_matriz
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $un_df
##   numeros letras
## 1       1      a
## 2       2      b
## 3       3      c
```

Cuando creamos listas lo más común es nombrar cada una de las entradas para luego poder extraer esos datos con el signo "$."

```
lista_clientes <- list(saldo = runif(3, min = 1000, max = 2000), nombres = c("


lista_clientes
```

```
## $saldo
## [1] 1514.047 1169.385 1807.954
```

```
##
## $nombres
## [1] "Juan"   "Luis"   "Carlos"
##
## $tarjeta.credito
## [1]  TRUE  TRUE FALSE
```

```
# runif(cantidad, min, max) genera números aleatorios.
```

```
lista_clientes$saldo
```

```
## [1] 1514.047 1169.385 1807.954
```

Otro ejemplo:

```
datos.lucas <- list(Nombre = "Lucas", Edad = 33, Tarjeta.Credito = FALSE)

datos.lucas
```

```
## $Nombre
## [1] "Lucas"
##
## $Edad
## [1] 33
##
## $Tarjeta.Credito
## [1] FALSE
```

Una ventaja del objeto lista es que podemos accesar a cada uno de sus argumentos mediante el símbolo "$," por ejemplo si quisieramos ver su Edad y luego si posee tarjeta de crédito o no, podemos escribir

```
datos.lucas$Edad
```

```
## [1] 33
```

```
datos.lucas$Tarjeta.Credito
```

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

5.5. Factores

Esta clase de datos se utiliza para representar datos categóricos. Estos pueden ser ordenados y sin orden. Podemos pensar en los factores como un vector de enteros, donde cada número representa una categoría.

- Los factores tienen un tratamiento especial en las funciones de modelación como lm() y glm(). (Regresiones lineales)
- Es mejor utilizar factores que utilizar enteros, por ejemplo tener la variable Estado civil, con los valores "Casado," "Soltero" es mejor que utilizar los valores 1 y 2.

La función para crear variables categóricas es **factor()**

```
estado.deuda <- factor(c("NORMAL", "NORMAL", "VENCIDA", "COBRO JUDICIAL", "VEN
estado.deuda
```

```
## [1] NORMAL         NORMAL         VENCIDA        COBRO JUDICIAL VENCIDA
## Levels: COBRO JUDICIAL NORMAL VENCIDA
```

Como podemos ver, al imprimir nuestra variable categórica tenemos tres niveles : COBRO JUDICIAL NORMAL VENCIDA. Estos representan las categorías en nuestra variable. R automáticamente hace está asignación por orden alfabético, si queremos definir nosotros el orden, podemos hacerlo utilizando el parámetro **levels**.

```
estado.deuda <- factor(c("NORMAL", "NORMAL", "VENCIDA", "COBRO JUDICIAL", "VEN

estado.deuda
```

```
## [1] NORMAL         NORMAL         VENCIDA        COBRO JUDICIAL VENCIDA
## Levels: NORMAL VENCIDA COBRO JUDICIAL
```

Una forma de saber cuantos individuos hay en cada categoría es mediante la función **table()**.

```
table(estado.deuda)
```

```
## estado.deuda
##         NORMAL        VENCIDA COBRO JUDICIAL
##              2              2              1
```

5.6. Valores ausentes

Los valores ausentes se denotan por **NA** (not avaiable) o **NaN**(not a number), las siguientes funciones se utilizan para verificar y encontrar valores ausentes.

1. **is.na()**: Se utiliza para verificar y encontrar los valores **NA** en un objeto.
2. **is.nan()**: Se utiliza para verificar y encontrar los valores **NaN** en un objeto.
3. Un valor **NaN** es un **NA** pero la otra dirección no es cierta.

```
## Creamos un vector que contenga un NA
x <- c(1, 3, NA, 10, 3)

## Retorna un vector de la misma longitud de x, con TRUE donde hay un NA y FALSE do
is.na(x)
```

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

```
## Creamos un vector que contenga un NA y NaN.
x <- c(1, 3, NA, 10, 3, NaN)

## Retorna un vector de la misma longitud de x, con TRUE donde hay un NA y F
is.na(x)
```

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

```
## Retorna un vector de la misma longitud de x, con TRUE donde hay un NA y F
is.nan(x)
```

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

con el ejemplo anterior se puede verificar el punto **3.**.

Capítulo 6

Subsetting

Vamos a ver como podemos obtener subconjuntos de nuestros datos, existen tres tipos de operaciones para extraer subconjuntos de datos en R:

- []: Siempre retorna un objeto de la misma clase que el original.
- [[]]: Se utiliza para extraer elementos de una lista o un data frame, mediante índices lógicos o numéricos. No necesariamente retorna una lista o data frame.
- $: Se utiliza para extraer elementos de una lista por su nombre.

6.1. Subsetting vectores

Índices numéricos:

```
x <- c("A", "BB+", "CCC", "AA+", "B", "B+")

x[1]
```

```
## [1] "A"
```

```
x[3]
```

```
## [1] "CCC"
```

```
x[1:3]
```

```
## [1] "A"   "BB+" "CCC"
```

Algo peculiar y útil en R es que podemos obtener elementos, seleccionando los que **no** queremos

```
x[-1]  ## Todos excepto el primer elemento
```

```
## [1] "BB+" "CCC" "AA+" "B"   "B+"
```

```
x[-c(1, 3, 5)] ## Todos excepto los elementos e la posición 1,3,5.
```

```
## [1] "BB+" "AA+" "B+"
```

Índices lógicos.

```
# Vector de enteros del 1 al 10.
y <- sample(30, 10)
# sample(x,n) retorna n números aleatorios sin repeticiones menores o iguale
y
```

```
##  [1] 14 27 26 12 29 19 16 10 20 23
```

```
# Retorna un vector con los valores mayores que 4.
y[y > 10]
```

```
## [1] 14 27 26 12 29 19 16 20 23
```

Esto también es válido pero es más lardo de escribir.

```
# Guardamos un índice lógico que nos devuelve TRUE en las posiciones de y que hay e
index <- y > 20
index
```

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

```
# Extraemos los elementos utilizando el índice lógico.
y[index]
```

```
## [1] 27 26 29 23
```

6.2. Subsetting matrices

Podemos obtener los elementos de una matriz utilizando los índices usuales, es decir, para obtener de la matriz M el elemento que está en la fila i y en la columna j, escribimos

$$M[i, j]$$

Si queremos obtener la fila i o la columna j escribimos

$$M[i,] \quad ; \quad M[, j]$$

respectivamente.

```
M <- matrix(1:9, 3, 3)

M
```

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

```
## Obtenemos el elemento que está en la fila 1 y la columna 3.
M[1, 3]
```

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

```
## Obtenemos la fila 1
M[1, ]
```

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

```
## Obtenemos la columna 2
M[, 2]
```

```
## [1] 4 5 6
```

```
## La matriz menos la fila 1

M[-1, ]
```

```
##      [,1] [,2] [,3]
## [1,]    2    5    8
## [2,]    3    6    9
```

```
# Con drop igual FALSE obtenemos un objeto de tipo matrix.
M[3, , drop = F]
```

```
##      [,1] [,2] [,3]
## [1,]    3    6    9
```

6.3. Subsetting data frames

Para extraer "trozos" de un data frame por filas y columnas (funciona exactamente igual que en matrices) donde n y m pueden definirse como:

- intervalos
- condiciones
- números naturales
- no poner nada

```
mtcars[1, ]
```

```
##           mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Mazda RX4  21   6  160 110  3.9 2.62 16.46  0  1    4    4
```

```
mtcars[1:3, 1:4]
```

```
##                mpg cyl disp  hp
## Mazda RX4     21.0   6  160 110
## Mazda RX4 Wag 21.0   6  160 110
## Datsun 710    22.8   4  108  93
```

Para extraer a solo una variable (columna) del data frame podemos utilizar el simbolo de dólar "$."

```
mtcars$hp
```

```
##  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230  66  52
## [20]  65  97 150 150 245 175  66  91 113 264 175 335 109
```

Para hacer filtros podemos combinar los dos métodos anteriores:

```
mtcars[mtcars$hp > 100, ]
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

6.4. Subsetting listas

Para acceder a elementos de las listas podemos usar **$** o doble corchete [[ ]], ambos realizan la misma operación in embargo una usa índice y el otro el nombre del elemento.

```
datos.cliente <- list(Nombre = c("Lucas", "Luis", "Diego"), Edad = c(33, 50, 2

datos.cliente
```

```
## $Nombre
```

```
## [1] "Lucas" "Luis"  "Diego"
##
## $Edad
## [1] 33 50 20
##
## $Tarjeta.Credito
## [1]  TRUE FALSE  TRUE
```

```
datos.cliente$Nombre
```

```
## [1] "Lucas" "Luis"  "Diego"
```

```
datos.cliente$Edad
```

```
## [1] 33 50 20
```

```
datos.cliente$Tarjeta.Credito
```

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

```
datos.cliente[[1]]
```

```
## [1] "Lucas" "Luis"  "Diego"
```

```
datos.cliente[[2]]
```

```
## [1] 33 50 20
```

Si queremos el valor i del elemento j escribimos

$$lista[[j]][i]$$

```
datos.cliente[[3]][1] ## Valor 1 del elemento 3.
```

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

Capítulo 7

Operaciones vectorizadas.

La idea de las operaciones vectorizadas es que los cálculos se pueden hacer en paralelo.

Muchas de las operaciones en R son *vectorizadas*, esto hace que el código sea mucho más eficiente, fácil de escribir y leer.

Suma de dos vectores

```
x <- 1:4
y <- 6:9
x
```

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

```
y
```

```
## [1] 6 7 8 9
```

En otros lenguajes

```
z <- vector("numeric", length = length(x))
for (i in 1:length(x)) {
  z[i] <- x[i] + y[i]
}
z
```

```
## [1]  7  9 11 13
```

En R

```
x + y
```

```
## [1]  7  9 11 13
```

Otras operaciones

```
x > 2
```

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

```
y == 8
```

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

```
x * y
```

```
## [1]  6 14 24 36
```

```
x / y
```

```
## [1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Similar para matrices

```
x <- matrix(1:4, 2, 2)
y <- matrix(rep(10, 4), 2, 2)  ## rep(x, n) repite el objeto x n veces.

x
```

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

```
y # imprimir las matrices
```

```
##      [,1] [,2]
## [1,]   10   10
## [2,]   10   10
```

```
x * y ## multiplicación entrada por entrada
```

```
##      [,1] [,2]
## [1,]   10   30
## [2,]   20   40
```

```
x / y ## división entrada por entrada
```

```
##      [,1] [,2]
## [1,]  0.1  0.3
## [2,]  0.2  0.4
```

```
x %*% y ## multiplicación matricial
```

```
##      [,1] [,2]
## [1,]   40   40
## [2,]   60   60
```

Capítulo 8

Estructuras de control en R

Las estructuras de control nos permiten controlar el flujo de ejecución de una secuencia de comandos. De este modo, podemos poner «lógica» en el código de R y lograr así reutilizar fragmentos de código una y otra vez.

Las estructuras de control más utilizadas son:

- if, else: permite decidir si ejecutar o no un fragmento de código en función de una condición.
- for: ejecuta un bucle una cantidad fija de veces.
- while: ejecuta un bucle mientras sea verdadera una condición.
- repeat: ejecuta un bucle indefinidamente. (la única forma de detener esta estructura es mediante el comando break).
- break: detiene la ejecución de un bucle.
- next: salta a la siguiente ejecución de un bucle.
- return: permite salir de la función.

La mayoría de estas no son usadas escribimos código directo en la consola, sino cuando escribimos funciones o expresiones largas. En la próxima clase veremos como trabajar con funcionar en R, pero es necesario tener bases sólidas de estos conceptos pues son necesarias cada vez que queramos producir o leer código.

8.1. if-else:

La combinación if-else es muy utilizada a la hora de programar. Esta estructura de control permite actuar en función de una condición. La sintaxis es la siguiente

```
if(<condicion>) {
  ## bloque de código
}
```

```
if(<condicion>) {
  ## bloque de código
} else {
  ## otro bloque de código
}
```

```
if(<condition1>) {
  ## bloque de código
} else if(<condicion2>) {
  ## otro bloque de código
} else {
  ## otro bloque de código
}
```

Ejemplo

```
x <- runif(1, 1, 10)
y <- 0

if (x > 5) {
  y <- 10
}
x
```

```
## [1] 1.679069
```

```
y
```

```
## [1] 0
```

```
tipo.cambio <- 585.6

moneda.deuda <- sample(c("CRC", "USD"), 1)

saldo.deuda <- runif(1, 1, 1000)

saldo.deuda
```

```
## [1] 104.3849
```

```
moneda.deuda
```

```
## [1] "CRC"
```

```
if (moneda.deuda == "USD") {
  saldo.deuda <- saldo.deuda * tipo.cambio
}


saldo.deuda
```

```
## [1] 104.3849
```

```
estado.mora <- c("")

dias.mora <- sample(85:100, 1) # sample(x,m), genera m números aleatorios tomados de

dias.mora
```

```
## [1] 85
```

```
if (dias.mora > 90) {
  estado.mora <- "Mora 90"
} else {
  estado.mora <- "Normal"
}

estado.mora
```

```
## [1] "Normal"
```

```
estado.mora <- c("")

dias.mora <- sample(85:145, 1) # sample(x,m), genera m números aleatorios tom

dias.mora
```

```
## [1] 119
```

```
if (dias.mora > 120) {
  estado.mora <- "Cobro Judicial"
} else if (90 > dias.mora) {
  estado.mora <- "Normal"
} else {
  estado.mora <- "Mora 90"
}

estado.mora
```

```
## [1] "Mora 90"
```

ifelse() es una función que nos permite escribir de forma más compacta la estructura if-else.

```
saldo.deuda <- ifelse(moneda.deuda == "USD", saldo.deuda * tipo.cambio, saldo.deuda)

saldo.deuda
```

```
## [1] 104.3849
```

8.2. for loop

Los bucles **for** se utilizan para recorrer .

```
for(<variable> in <objeto iterable>) {
  # código
  ...
}
```

Recorrer por índice.

```
meses <- c("Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio", "Julio", "Agosto",

for (i in 1:6) {
  print(meses[i])
}
```

```
## [1] "Enero"
## [1] "Febrero"
## [1] "Marzo"
## [1] "Abril"
## [1] "Mayo"
## [1] "Junio"
```

La función **seq_along()** es muy utilizada en los ciclos for, para poder generar una secuencia de enteros basada en el tamaño del objeto sobre el que queremos iterar.

```
for (i in seq_along(meses)) {
  print(meses[i])
}
```

```
## [1] "Enero"
## [1] "Febrero"
## [1] "Marzo"
## [1] "Abril"
## [1] "Mayo"
## [1] "Junio"
## [1] "Julio"
## [1] "Agosto"
## [1] "Setiembre"
## [1] "Octubre"
## [1] "Noviembre"
## [1] "Diciembre"
```

Recorrer los elementos.

```
for (mes in meses) {
  print(mes)
}
```

```
## [1] "Enero"
## [1] "Febrero"
## [1] "Marzo"
## [1] "Abril"
## [1] "Mayo"
## [1] "Junio"
## [1] "Julio"
## [1] "Agosto"
## [1] "Setiembre"
## [1] "Octubre"
## [1] "Noviembre"
## [1] "Diciembre"
```

8.3. while loop

Los ciclos while comienzan revisando una condición, si se cumple inicia el ciclo y se repite hasta que la condición no se cumpla.

```
contador <- 0

while (contador < 5) {
  print(contador)
  contador <- contador + 1

}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

Caminata aleatoria

```
z <- 5

set.seed(1)

while (z >= 3 && z <= 10) {
  moneda <- rbinom(1, 1, 0.5)

  if (moneda == 1) { ## Paso hacia la derecha
    z <- z + 1
  } else {           ## Paso hacia la izquierda
    z <- z - 1
  }
}

z
```

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

8.4. repeat, next, break

repeat inicia un ciclo infinito. La única forma de terminar o de salir de un ciclo **repeat** es mediante la instrucción **break**. No son muy comunes a la hora de hacer análisis de datos, pero vale la pena mencionarlos pues se pueden utilizar para algoritmos que busquen una solución con cierto de nivel de tolerancia, ya que en estos casos no se puede saber de ante mano cuantas iteraciones se necesitan.

```
x0 <- 1

tol <- 1e-10

repeat{


  x1 <- algoritmoEstimacion() ## Se calcula el estimado

  if (abs(x1 - x0) < tol) { ## Se hace el test
    break
  } else {   ## Continua

    x0 <- x1
  }

}
```

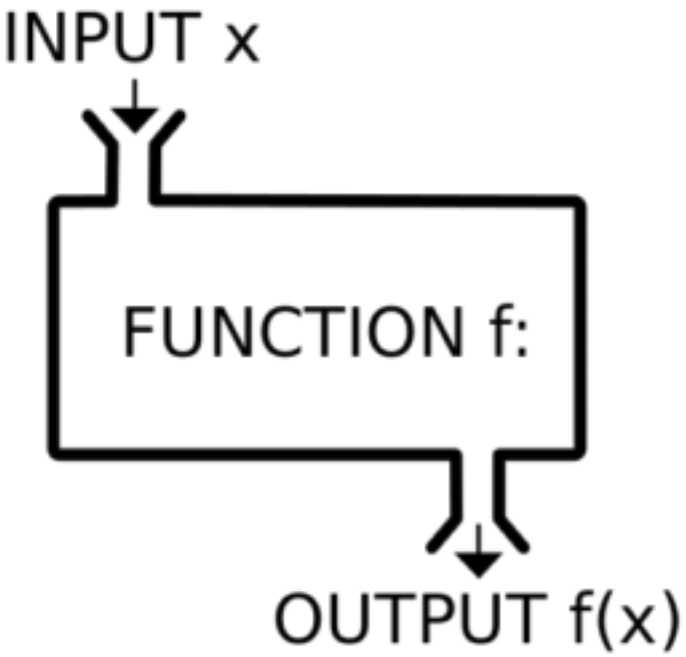
next: Se utiliza para avanzar a la siguiente iteración del ciclo. **break**: Se utiliza para salir del ciclo inmediatamente.

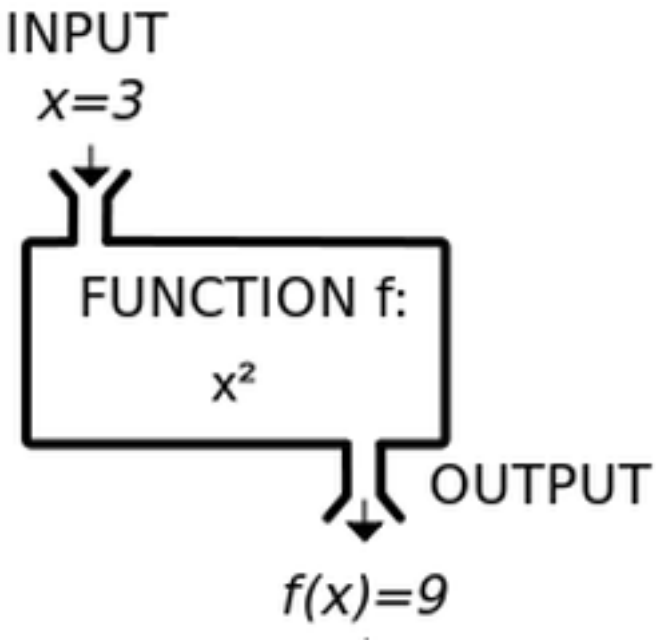
Capítulo 9

Funciones

Como analistas de datos escribir funciones es una de las mejores herramientas, ya que nos permiten automatizar y estandarizar tareas, además de que hace nuestro código más legible y mucho más sencillo de mantener.

La idea de este capítulo es introducir los conceptos básicos de las funciones y buenas prácticas a la hora de escribirlas.


INPUT x
FUNCTION f:
OUTPUT f(x)



Sintaxis

```
nombre.funcion <- function(<entradas>){

  ##Cuerpo de la función

  return <resultado>

}
```

Ejemplos:

```
## Función que calcula el area de un triángulo dada su base y su altura.


# Parámetros
## base : Base del triángulo.
## altura: Altura del triángulo
```

```
# Resultado
## area: Área del triángulo.

area.triangulo <- function(base, altura) {
  area <- (base * altura) / 2
  return(area)
}

area.triangulo(2, 5)
```

```
## [1] 5
```

Una persona desea sacar un préstamo, de P colones a una tasa de interés mensual i. El préstamo tiene que ser reembolsado en n cuotas mensuales de tamaño C, comenzando dentro de un mes. El problema es calcular C. La fórmula C es:

$$C = P \cdot \left(\frac{i}{1 - (1+i)^{-n}} \right)$$

Supongamos que $P = 150000$, que la tasa de interés es del $2\,\%$ y que le número de pagos es 10. EL código en **R** sería:

```
tasa.interes <- 0.02
n <- 10
principal <- 150000
pago <- principal * tasa.interes / (1 - (1 + tasa.interes)^(-n))
pago
```

```
## [1] 16698.98
```

Utilizando una función:

```r
# Parámetros
## tasa.interes : Tasa de interés mensual del préstamo
## n: Número de cuotas
## principal: Monto del préstamo

# Resultado
## pago: Cuota del préstamo.


calcula.cuota <- function(tasa.interes, n, principal) {
  pago <- principal * tasa.interes / (1 - (1 + tasa.interes)^(-n))
  return(pago)
}


calcula.cuota(0.02, 10, 150000) ## Orden por defecto
```

```
## [1] 16698.98
```

```r
calcula.cuota(n = 340, tasa.interes = 0.11, principal = 3500000) ## Para cambiar el o
```

```
## [1] 385000
```

Ejemplo de función que retorna una lista.

```r
# Parámetros
## DF : Data frame
## NC: Número de columna

# Resultado
## lista con nombre de la variable correspondiente al número de columna, la media, l

estadisticas <- function(DF, NC) {

  variable <- DF[, NC]
  nombre <- colnames(DF)[NC]
```

```
  media <- mean(variable)
  mediana <- median(variable)
  deviacion <- sd(variable)
  varianza <- var(variable)
  maximo <- max(variable)
  minimo <- min(variable)

  return(list(Variable = nombre, Media = media, Mediana = mediana, DesEst = de
}


estadisticas(portafolio.banco, 20)
```

```
## Error in estadisticas(portafolio.banco, 20): object 'portafolio.banco' not
```

Capítulo 10

Importación de datos

Algunas de las funciones base en R para la lectura datos son:

- **read.table** , **read.csv**, se utilizan para leer datos que tienen formato de tabla.

10.0.1. read.table

```
read.table(file = archivo[, header = TRUE | FALSE,
  sep = separadorDatos, dec = separadorDecimal,
  quote = delimitadorCadenas,
  stringsAsFactors = TRUE | FALSE])
```

Esta es la función genérica para leer datos en formato .csv y genera , algunos de sus argumentos son:

- **file**: El nombre del archivo o su ubicación.
- **header**: Variable lógica que indica si el archivo tiene encabezado.
- **sep**: String que indica como están separadas las columnas.
- **dec**: Para datos numéricos, establece cuál es el separador entre parte entera y decimal.
- **colClasses**: Vector con las clases de cada una de las columnas.

- **stringsAsFactors**: Indica si las variables de tipo character se deben leer como factor.

Leer documentación de la función read.table.

10.0.2. read.csv

Es una implementación especializada de read.table() en la que se asume que los parámetros header, sep y dec toman los valores TRUE, "," y "." respectivamente.

```
datos.credito <- read.csv("data/DeudaCredito.csv", sep = ";", dec = ".")

str(datos.credito)
```

```
## 'data.frame':    400 obs. of  13 variables:
##  $ X             : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ monto_ingreso: num  14891 106025 104593 148924 55882 ...
##  $ monto_limite : int  3606 6645 7075 9504 4897 8047 3388 7114 3300 6819 ..
##  $ CalifCredit   : int  283 483 514 681 357 569 259 512 266 491 ...
##  $ Tarjetas      : int  2 3 4 3 2 4 2 2 5 3 ...
##  $ Edad          : int  34 82 71 36 68 77 37 87 66 41 ...
##  $ Educacion     : int  11 15 11 11 16 10 12 9 13 19 ...
##  $ Genero        : chr  "Masculino" "Femenino" "Masculino" "Femenino" ...
##  $ Estudiante    : chr  "No" "Si" "No" "No" ...
##  $ Casado        : int  1 1 0 0 1 0 0 0 0 1 ...
##  $ Etnicidad     : chr  "Caucasico" "Asiatico" "Asiatico" "Asiatico" ...
##  $ monto_balance: int  333 903 580 964 331 1151 203 872 279 1350 ...
##  $ fecha_ini     : chr  "12/11/2019" "12/13/2019" "12/17/2019" "12/06/2019"
```

Podemos ver que varias variables que deberían ser categóricas se leyeron como strings, para corregir esto podemos colocar el parámetro **stringsAsFactors** igual a TRUE de la función read_excel(), para leer los caracteres como factores.

```
datos.credito <- read.csv("data/DeudaCredito.csv", sep = ",", dec = ".", stringsAsFac

str(datos.credito)
```

```
## 'data.frame':    400 obs. of  12 variables:
##  $ X          : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Ingreso    : num  14.9 106 104.6 148.9 55.9 ...
##  $ Limite     : int  3606 6645 7075 9504 4897 8047 3388 7114 3300 6819 ...
##  $ CalifCredit: int  283 483 514 681 357 569 259 512 266 491 ...
##  $ Tarjetas   : int  2 3 4 3 2 4 2 2 5 3 ...
##  $ Edad       : int  34 82 71 36 68 77 37 87 66 41 ...
##  $ Educacion  : int  11 15 11 11 16 10 12 9 13 19 ...
##  $ Genero     : Factor w/ 2 levels "Femenino","Masculino": 2 1 2 1 2 2 1 2 1 1 ...
##  $ Estudiante : Factor w/ 2 levels "No","Si": 1 2 1 1 1 1 1 1 1 2 ...
##  $ Casado     : int  1 1 0 0 1 0 0 0 0 1 ...
##  $ Etnicidad  : Factor w/ 3 levels "Afrodescendiente",..: 3 2 2 2 3 3 1 2 3 1 ...
##  $ Balance    : int  333 903 580 964 331 1151 203 872 279 1350 ...
```

10.0.3. read_excel

Esta se utiliza para leer datos de excel, algunos de sus argumentos son:

- **path**: Ruta del archivo.
- **sheet**: Hoja del excel que se desea leer, por defecto es la primera.
- **range**: Rango de celdas que se desean leer.
- **col_types**: Vector con las clases de cada una de las columnas.
- **col_names**: Indica si la primera fila corresponde al nombre de las columnas.

```
library(readxl)


tipo_cambio <- read_excel("data/tipo_cambio.xls")

tipo_cambio_top10 <- head(tipo_cambio, 10) ## head(data,n) retorna las primeras n fi
```

```
tipo_cambio_top10
```

```
## # A tibble: 10 x 3
##    'Tipo cambio de compra y de venta del dólar de lo~ ...2         ...3
##    <chr>                                              <chr>        <chr>
##  1 Referencia del Banco Central de Costa Rica         <NA>         <NA>
##  2 En colones costarricenses                          <NA>         <NA>
##  3 <NA>                                               <NA>         <NA>
##  4 <NA>                                               TIPO CAMBIO~ TIPO DE
##  5 1 Ene 2019                                         604.3899999~ 611.75
##  6 2 Ene 2019                                         604.3899999~ 611.75
##  7 3 Ene 2019                                         603.0099999~ 611.5499
##  8 4 Ene 2019                                         604.7699999~ 611.6799
##  9 5 Ene 2019                                         602.3999999~ 611.5499
## 10 6 Ene 2019                                         602.3999999~ 611.5499
```

Aplicando el head(), podemos ver que la lectura del archivo no es correcta, para hacerlo de forma correcta podemos utilizar el parámetro range de la función read_excel(), para decirle que celdas queremos leer, la notación es la mimsa que se utiliza en MS Excel.

```
tipo_cambio <- read_excel("data/tipo_cambio.xls", range = "A6:C787", col_names

head(tipo_cambio)
```

```
## # A tibble: 6 x 3
##   fecha      compra venta
##   <chr>       <dbl> <dbl>
## 1 1 Ene 2019   604.  612.
## 2 2 Ene 2019   604.  612.
## 3 3 Ene 2019   603.  612.
## 4 4 Ene 2019   605.  612.
## 5 5 Ene 2019   602.  612.
## 6 6 Ene 2019   602.  612.
```

10.0.4. Calculando requisitos de memoria.

Si queremos leer un archivo con 1.500.000 filas y 120 columnas, donde todas son de tipo numérico, realizamos el siguiente cálculo

$$
\begin{aligned}
1,500,000 \times 120 \times 8(bytes) & \\
&= 1{,}44 \times 10^9 (bytes) \\
&= 1{,}44 \times 10^9 / 2^{20} (MB) \\
&= 1,373(MB) \\
&= 1{,}34(GB)
\end{aligned}
$$

por lo general se necesita el doble de esto, por lo que necesitamos al menos 4GB de RAM en nuestra computadora.

Capítulo 11

Análisis exploratorio

Salvo que lo hayamos creado nosotros mismos o estemos familiarizados con el conjunto de datos que carguemos a **R** generalmente estamos interesados en obtener una idea general sobre su contenido. Con este fin se aplican funciones de estadística descriptiva, conservando la estructura de los datos (columnas que lo forman y su tipo, número de observaciones, etc.), para tener una idea general sobre cada variable.

11.0.1. Información general.

Asumiendo que comenzamos a trabajar con un conjunto de datos desconocido, lo primero que nos interesa es sera saber qué atributos contiene, cuántas observaciones hay, etc.

En secciones anteriores se definieron funciones como class() y typeof(), con las que podemos conocer la clase de un objeto y su tipo.

La función **str()** aporta más información, incluyendo el número de variables y observaciones y algunos detalles sobre cada una de las variables (columnas).

```
class(datos.credito) # Clase del objeto
```

```
## [1] "data.frame"
```

```
# Información sobre su estructura

str(datos.credito)
```

```
## 'data.frame':    400 obs. of  12 variables:
##  $ X          : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Ingreso    : num  14.9 106 104.6 148.9 55.9 ...
##  $ Limite     : int  3606 6645 7075 9504 4897 8047 3388 7114 3300 6819 ...
##  $ CalifCredit: int  283 483 514 681 357 569 259 512 266 491 ...
##  $ Tarjetas   : int  2 3 4 3 2 4 2 2 5 3 ...
##  $ Edad       : int  34 82 71 36 68 77 37 87 66 41 ...
##  $ Educacion  : int  11 15 11 11 16 10 12 9 13 19 ...
##  $ Genero     : Factor w/ 2 levels "Femenino","Masculino": 2 1 2 1 2 2 1 2
##  $ Estudiante : Factor w/ 2 levels "No","Si": 1 2 1 1 1 1 1 1 1 2 ...
##  $ Casado     : int  1 1 0 0 1 0 0 0 0 1 ...
##  $ Etnicidad  : Factor w/ 3 levels "Afrodescendiente",..: 3 2 2 2 3 3 1 2 3
##  $ Balance    : int  333 903 580 964 331 1151 203 872 279 1350 ...
```

11.0.2. Exploración del contenido.

Aunque la función **str()** facilita una muestra del contenido de cada variable, en general dicha información es insuficiente. Podemos recurrir a funciones como **head()** y **tail()** para obtener los primeros y últimos elementos, respectivamente, de un objeto en **R**. Asimismo, la función **summary()** ofrece un resumen global del contenido de cada variable: su valor mínimo, máximo y medio, mediana, cuartiles y, en el caso de las variables qualitativas, el número de elementos por categoría.

```
head(datos.credito) ## head(X,n) muestra los primeros n elementos del objeto
```

```
##   X Ingreso Limite CalifCredit Tarjetas Edad Educacion    Genero Estudiante
## 1 1  14.891   3606         283        2   34        11 Masculino         No
## 2 2 106.025   6645         483        3   82        15  Femenino         Si
## 3 3 104.593   7075         514        4   71        11 Masculino         No
## 4 4 148.924   9504         681        3   36        11  Femenino         No
## 5 5  55.882   4897         357        2   68        16 Masculino         No
```

```
## 6 6  80.180   8047         569        4   77        10 Masculino         No
##   Casado Etnicidad Balance
## 1      1 Caucasico     333
## 2      1  Asiatico     903
## 3      0  Asiatico     580
## 4      0  Asiatico     964
## 5      1 Caucasico     331
## 6      0 Caucasico    1151
```

```
head(datos.credito, 3)
```

```
##   X Ingreso Limite CalifCredit Tarjetas Edad Educacion    Genero Estudiante
## 1 1  14.891   3606         283        2   34        11 Masculino         No
## 2 2 106.025   6645         483        3   82        15  Femenino         Si
## 3 3 104.593   7075         514        4   71        11 Masculino         No
##   Casado Etnicidad Balance
## 1      1 Caucasico     333
## 2      1  Asiatico     903
## 3      0  Asiatico     580
```

```
tail(datos.credito)
```

```
##       X Ingreso Limite CalifCredit Tarjetas Edad Educacion    Genero Estudiante
## 395 395  49.794   5758         410        4   40         8 Masculino         No
## 396 396  12.096   4100         307        3   32        13 Masculino         No
## 397 397  13.364   3838         296        5   65        17 Masculino         No
## 398 398  57.872   4171         321        5   67        12  Femenino         No
## 399 399  37.728   2525         192        1   44        13 Masculino         No
## 400 400  18.701   5524         415        5   64         7  Femenino         No
##     Casado        Etnicidad Balance
## 395      0        Caucasico     734
## 396      1        Caucasico     560
## 397      0 Afrodescendiente     480
## 398      1        Caucasico     138
## 399      1        Caucasico       0
## 400      0         Asiatico     966
```

```
tail(datos.credito, 3)
```

```
##       X Ingreso Limite CalifCredit Tarjetas Edad Educacion    Genero Estudi
## 398 398  57.872   4171         321        5   67        12  Femenino
## 399 399  37.728   2525         192        1   44        13 Masculino
## 400 400  18.701   5524         415        5   64         7  Femenino
##     Casado Etnicidad Balance
## 398      1 Caucasico     138
## 399      1 Caucasico       0
## 400      0  Asiatico     966
```

```
summary(datos.credito)
```

```
##        X            Ingreso           Limite       CalifCredit
##  Min.   :  1.0   Min.   : 10.35   Min.   :  855   Min.   : 93.0
##  1st Qu.:100.8   1st Qu.: 21.01   1st Qu.: 3088   1st Qu.:247.2
##  Median :200.5   Median : 33.12   Median : 4622   Median :344.0
##  Mean   :200.5   Mean   : 45.22   Mean   : 4736   Mean   :354.9
##  3rd Qu.:300.2   3rd Qu.: 57.47   3rd Qu.: 5873   3rd Qu.:437.2
##  Max.   :400.0   Max.   :186.63   Max.   :13913   Max.   :982.0
##     Tarjetas          Edad         Educacion           Genero    Estudiante
##  Min.   :1.000   Min.   :23.00   Min.   : 5.00   Femenino :207   No:360
##  1st Qu.:2.000   1st Qu.:41.75   1st Qu.:11.00   Masculino:193   Si: 40
##  Median :3.000   Median :56.00   Median :14.00
##  Mean   :2.958   Mean   :55.67   Mean   :13.45
##  3rd Qu.:4.000   3rd Qu.:70.00   3rd Qu.:16.00
##  Max.   :9.000   Max.   :98.00   Max.   :20.00
##      Casado                  Etnicidad      Balance
##  Min.   :0.0000   Afrodescendiente: 99   Min.   :   0.00
##  1st Qu.:0.0000   Asiatico        :102   1st Qu.:  68.75
##  Median :1.0000   Caucasico       :199   Median : 459.50
##  Mean   :0.6125                          Mean   : 520.01
##  3rd Qu.:1.0000                          3rd Qu.: 863.00
##  Max.   :1.0000                          Max.   :1999.00
```

Como podemos observar la variable **Casado** se leyó como numérica, esto no tienen mucho sentido, para transformarla a categórica utilizamos la función factor().

```
## Transformamos la variable Casado a categorica

datos.credito$Casado <- factor(datos.credito$Casado, levels = c(1, 0), labels = c("si
summary(datos.credito)
```

```
##        X            Ingreso          Limite       CalifCredit
##  Min.   :  1.0   Min.   : 10.35   Min.   :  855   Min.   : 93.0
##  1st Qu.:100.8   1st Qu.: 21.01   1st Qu.: 3088   1st Qu.:247.2
##  Median :200.5   Median : 33.12   Median : 4622   Median :344.0
##  Mean   :200.5   Mean   : 45.22   Mean   : 4736   Mean   :354.9
##  3rd Qu.:300.2   3rd Qu.: 57.47   3rd Qu.: 5873   3rd Qu.:437.2
##  Max.   :400.0   Max.   :186.63   Max.   :13913   Max.   :982.0
##     Tarjetas          Edad         Educacion          Genero    Estudiante
##  Min.   :1.000   Min.   :23.00   Min.   : 5.00   Femenino :207   No:360
##  1st Qu.:2.000   1st Qu.:41.75   1st Qu.:11.00   Masculino:193   Si: 40
##  Median :3.000   Median :56.00   Median :14.00
##  Mean   :2.958   Mean   :55.67   Mean   :13.45
##  3rd Qu.:4.000   3rd Qu.:70.00   3rd Qu.:16.00
##  Max.   :9.000   Max.   :98.00   Max.   :20.00
##  Casado              Etnicidad     Balance
##  si:245   Afrodescendiente: 99   Min.   :   0.00
##  no:155   Asiatico        :102   1st Qu.:  68.75
##           Caucasico       :199   Median : 459.50
##                                  Mean   : 520.01
##                                  3rd Qu.: 863.00
##                                  Max.   :1999.00
```

Por otro lado la variable fecha_ini se leyó como factor, en lugar de como fecha, para corregir esto usamos la función as.Date().

```
datos.credito$fecha_ini <- as.Date(datos.credito$fecha_ini, format = "%m/%d/%Y")
```

```
## Error in `$<-.data.frame`(`*tmp*`, fecha_ini, value = structure(numeric(0), class
```

11.0.3. Funciones básicas.

Recordemos que **R** cuenta con multitud de funciones de tipo estadístico, entre ellas las que permiten obtener información descriptiva sobre la distribución de valores en un vector. Estas funciones pueden también aplicarse a objetos más complejos, como comprobaremos después. La sintaxis de las funciones de estadística descriptiva más comunes se presentan a continuación.

```
min(vector, na.rm = T / F) # Devuelve el valor mínimo existente en el vector

# El resultado será NA si el vector contiene algún valor ausente, a menos qu
# entregue el parámetro na.rm con el valor TRUE.
```

```
max(vector, na.rm = T / F) # Devuelve el valor máximo existente en el vector

# El resultado será NA si el vector contiene algún valor ausente, a menos qu
# entregue el parámetro na.rm con el valor TRUE.
```

```
range(vector, na.rm = T / F) # Devuelve un vector de dos elementos con el val
# existentes en el vector facilitado como parámetro
```

```
range(vector, na.rm = T / F) # Devuelve un vector de dos elementos con el val
# existentes en el vector facilitado como parámetro
```

```
saldo <- c(1000, 2000, 3000, 4500)

min(saldo)
```

```
## [1] 1000
```

```
max(saldo)
```

```
## [1] 4500
```

```
range(saldo)
```

```
## [1] 1000 4500
```

```
mean(saldo)
```

```
## [1] 2625
```

```
var(saldo)
```

```
## [1] 2229167
```

```
sd(saldo)
```

```
## [1] 1493.039
```

```
median(saldo)
```

```
## [1] 2500
```

```
quantile(saldo)
```

```
##   0%  25%  50%  75% 100%
## 1000 1750 2500 3375 4500
```

A fin de obtener un resultado más compacto, se crea una lista con el valor devuelto por cada operación y, finalmente, se usa la función **unlist()** para generar un vector con la información a mostrar:

```
valores <- saldo
unlist(list(media = mean(valores), desviacion = sd(valores), varianza = var(valores),
```

```
##          media     desviacion       varianza         minimo         maximo
##       2625.000       1493.039    2229166.667       1000.000       4500.000
##        mediana         rango1         rango2   quartiles.0%  quartiles.25%
##       2500.000       1000.000       4500.000       1000.000       1750.000
##  quartiles.50%  quartiles.75% quartiles.100%
##       2500.000       3375.000       4500.000
```

11.0.4. Aplicación a estructuras complejas.

Las anteriores funciones pueden aplicarse sobre estructuras más complejas que los vectores, como matrices y data frames. En la mayoría de los casos no nos interesan las medidas estadísticas de todo el conjunto de datos, sino de cada una de las variables (columnas) por separado.

```
mean(datos.credito$Ingreso)
```

```
## [1] 45.21889
```

```
max(datos.credito$Limite)
```

```
## [1] 13913
```

Capítulo 12

Data Frames con el paquete dplyr.

Como vimos en la clase anterior los data frames son las estructuras más importantes en R, recordemos que básicamente un data frame es una tabla, donde cada fila representa una observación o individuo, y cada columna una variable o característica de esta observación.

Dada la importancia de estas estructuras, es muy importante conocer las mejores herramientas para trabajar con ellas, en la sección de subsetting vimos como obtener subconjuntos de nuestros datos, sin embargo cuando tenemos que hacer varios filtros o agrupaciones el uso de "[]," "$," no es tan recomendable, pues es más fácil equivocarse y el código es más complicado de leer.

El paquete **dplyr** está diseñado para mitigar estas complicaciones y optimizado para realizar estas tareas.

12.1. Paquete dplyr:

El paquete **dplyr** fue desarrollado por Hadley Wickham de RStudio y us una versión mejorada del paquete **plyr**. Una de las ventajas de este paquete es que tiene cierta gramática en sus funciones, lo que facilita escribir y leer código. Además sus funciones son muy rápidas y algunas de sus operaciones están programadas en C++.

Gramática de dplyr

Algunos de los "verbos" que tiene el paquete **dplyr** son los siguientes:

- **select**: Retorna un subconjunto de columnas.
- **filter**: Extrae subconjuntos de filas basado en condiciones lógicas.
- **arrange**: Reordena las filas de un data frame.
- **rename**: Renombra las variables del data frame.
- **mutate**: Agrega columnas o transforma las existentes.
- **summarise / summarize**: Genera un resumen estadístico de las variables del data frame.
- %>%: El operador "pipe" es usado para conectar varios "verbos" en una sola ejecución.

Es importante notar que, el primer argumento de todas estas funciones es un data frame y su resultado también es un data frame, por eso es fácil y útil combinarlas.

Instalación

```
## Para instalarlo basta ejecutar lo siguiente en la consola
install.packages("dplyr")

## Para utilizar las funciones se debe cargar la librería mediante la instru
library(dplyr)
```

Cargamos el archivo de datos

```
datos.credito <- read.csv("data/DeudaCredito.csv", sep = ";", dec = ".", strin

datos.credito$Casado <- factor(datos.credito$Casado, levels = c(1, 0), labels

datos.credito$fecha_ini <- as.Date(datos.credito$fecha_ini, format = "%m/%d/%
```

12.2. select():

Normalmente trabajamos con data frames que tienen muchas variables y necesitamos enfocarnos en solo algunas de estas, la función **select()** como

su nombre lo sugiere, sirve para obtener las columnas deseadas de nuestro conjunto de datos.

Primero vamos a ver de forma general la estructura de nuestros datos, utilizando las funciones **dim()** y **str()**.

```
dim(datos.credito) ## Obtenemos la dimensiones de nuestro data frame [filas, columna
```

```
## [1] 400  13
```

```
str(datos.credito) ## Presenta un resumen las variables del data frame y su clase.
```

```
## 'data.frame':    400 obs. of  13 variables:
##  $ X             : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ monto_ingreso: num  14891 106025 104593 148924 55882 ...
##  $ monto_limite : int  3606 6645 7075 9504 4897 8047 3388 7114 3300 6819 ...
##  $ CalifCredit  : int  283 483 514 681 357 569 259 512 266 491 ...
##  $ Tarjetas     : int  2 3 4 3 2 4 2 2 5 3 ...
##  $ Edad          : int  34 82 71 36 68 77 37 87 66 41 ...
##  $ Educacion     : int  11 15 11 11 16 10 12 9 13 19 ...
##  $ Genero        : Factor w/ 2 levels "Femenino","Masculino": 2 1 2 1 2 2 1 2 1 1 .
##  $ Estudiante    : Factor w/ 2 levels "No","Si": 1 2 1 1 1 1 1 1 1 2 ...
##  $ Casado        : Factor w/ 2 levels "si","no": 1 1 2 2 1 2 2 2 2 1 ...
##  $ Etnicidad     : Factor w/ 3 levels "Afrodescendiente",..: 3 2 2 2 3 3 1 2 3 1 ..
##  $ monto_balance: int  333 903 580 964 331 1151 203 872 279 1350 ...
##  $ fecha_ini     : Date, format: "2019-12-11" "2019-12-13" ...
```

Suponga que queremos las columnas Edad, Educación, Género, Estudiante, Casado

```
datos.credito.info_personal <- select(datos.credito, c("Edad", "Educacion", "Genero",

head(datos.credito.info_personal)
```

```
##   Edad Educacion    Genero Estudiante Casado Etnicidad
## 1   34        11 Masculino         No     si Caucasico
```

```
## 2   82        15  Femenino         Si     si  Asiatico
## 3   71        11 Masculino         No     no  Asiatico
## 4   36        11  Femenino         No     no  Asiatico
## 5   68        16 Masculino         No     si Caucasico
## 6   77        10 Masculino         No     no Caucasico
```

También podemos utilizar el **select()** eligiendo las columnas que no queremos.

```
datos.credito.info_personal.sinEtnia <- select(datos.credito.info_personal, -E

head(datos.credito.info_personal.sinEtnia)
```

```
##   Edad Educacion    Genero Estudiante Casado
## 1   34        11 Masculino         No     si
## 2   82        15  Femenino         Si     si
## 3   71        11 Masculino         No     no
## 4   36        11  Femenino         No     no
## 5   68        16 Masculino         No     si
## 6   77        10 Masculino         No     no
```

Otra forma es seleccionar las columnas que tengan inicien o terminen con ciertos caracteres, por ejemplo si queremos todas las columnas que tienen el prefijo "monto."

```
datos.credito_montos <- select(datos.credito, starts_with("monto")) ## Podemos

head(datos.credito_montos)
```

```
##   monto_ingreso monto_limite monto_balance
## 1      14891.00         3606           333
## 2     106025.00         6645           903
## 3     104593.00         7075           580
## 4     148924.00         9504           964
## 5      55882.00         4897           331
## 6         80.18         8047          1151
```

12.3. filter():

Esta función se utiliza para extraer filas de nuestro data frame utilizando condiciones.

```
datos.credito_enero <- filter(datos.credito, months(fecha_ini) == "January")

dim(datos.credito_enero)
```

```
## [1]  0 13
```

```
head(datos.credito_enero)
```

```
##  [1] X             monto_ingreso monto_limite  CalifCredit   Tarjetas
##  [6] Edad          Educacion     Genero        Estudiante    Casado
## [11] Etnicidad     monto_balance fecha_ini
## <0 rows> (or 0-length row.names)
```

```
datos.credito_enero_fem <- filter(datos.credito, months(fecha_ini) == "January" & Gen

dim(datos.credito_enero_fem)
```

```
## [1]  0 13
```

```
head(datos.credito_enero_fem)
```

```
##  [1] X             monto_ingreso monto_limite  CalifCredit   Tarjetas
##  [6] Edad          Educacion     Genero        Estudiante    Casado
## [11] Etnicidad     monto_balance fecha_ini
## <0 rows> (or 0-length row.names)
```

12.4. arrange():

Esta función se utiliza para ordenar las filas de un data frame de acuerdo a una de sus variables.

Ordenamos de acuerdo la columna que contiene las fechas de inicio.

```
datos.credito_ordenado <- arrange(datos.credito, fecha_ini) ## De forma ascend

head(select(datos.credito_ordenado, fecha_ini, monto_balance))
```

```
##    fecha_ini monto_balance
## 1 2011-01-17          1448
## 2 2016-06-01             0
## 3 2016-10-21           962
## 4 2016-10-21           345
## 5 2016-10-21             0
## 6 2016-10-21           480
```

```
datos.credito_ordenado <- arrange(datos.credito, desc(fecha_ini)) ## De forma

head(select(datos.credito_ordenado, fecha_ini, monto_balance))
```

```
##    fecha_ini monto_balance
## 1 2020-06-30           250
## 2 2020-06-30           295
## 3 2020-06-29           637
## 4 2020-06-29           209
## 5 2020-06-29           531
## 6 2020-06-26             0
```

12.5. rename():

Renombrar variables puede ser de mucha utilidad para poder escribir código y hacerlo más legible. Sin la función **rename()** esta tarea puede ser bastante tediosa.

```
datos.credito <- rename(datos.credito, cant_tarjetas = "Tarjetas", calif_credit = Cal

head(datos.credito)
```

```
##   id monto_ingreso monto_limite calif_credit cant_tarjetas Edad Educacion
## 1  1      14891.00         3606          283             2   34        11
## 2  2     106025.00         6645          483             3   82        15
## 3  3     104593.00         7075          514             4   71        11
## 4  4     148924.00         9504          681             3   36        11
## 5  5      55882.00         4897          357             2   68        16
## 6  6         80.18         8047          569             4   77        10
##      Genero Estudiante Casado Etnicidad monto_balance  fecha_ini
## 1 Masculino         No     si Caucasico           333 2019-12-11
## 2  Femenino         Si     si  Asiatico           903 2019-12-13
## 3 Masculino         No     no  Asiatico           580 2019-12-17
## 4  Femenino         No     no  Asiatico           964 2019-12-06
## 5 Masculino         No     si Caucasico           331 2019-10-17
## 6 Masculino         No     no Caucasico          1151 2020-02-07
```

12.6. mutate():

Esta función nos permite crear variables a partir de las que ya existen, de una forma muy sencilla.

```
datos.credito <- mutate(datos.credito, razon = monto_limite / monto_ingreso)

head(select(datos.credito, monto_ingreso, monto_limite, razon))
```

```
##   monto_ingreso monto_limite        razon
## 1      14891.00         3606   0.24215969
## 2     106025.00         6645   0.06267390
## 3     104593.00         7075   0.06764315
## 4     148924.00         9504   0.06381779
## 5      55882.00         4897   0.08763108
## 6         80.18         8047 100.36168621
```

También funciona para agregar variables de forma manual

```
datos.credito <- mutate(datos.credito, fecha_actual = Sys.Date(), dif_fechas =

head(select(datos.credito, fecha_ini, fecha_actual, dif_fechas))
```

```
##    fecha_ini fecha_actual dif_fechas
## 1 2019-12-11   2021-02-23   440 days
## 2 2019-12-13   2021-02-23   438 days
## 3 2019-12-17   2021-02-23   434 days
## 4 2019-12-06   2021-02-23   445 days
## 5 2019-10-17   2021-02-23   495 days
## 6 2020-02-07   2021-02-23   382 days
```

12.7. group_by():

Esta función se utiliza para generar subconjuntos de los datos a partir de ciertas propiedades, luego de hacer esto podemos generar resúmenes estadísticos de esos subconjuntos.

La estrategia en general es separar el data frame en partes de acuerdo a una o más variables y luego aplicar un summary en cada una de esas partes.

```
datos.credito_genero <- group_by(datos.credito, Genero) ## Agrupamos por gene

summarize(datos.credito_genero, mean(Edad))
```

```
## # A tibble: 2 x 2
##   Genero    `mean(Edad)`
## * <fct>            <dbl>
## 1 Femenino          55.7
## 2 Masculino         55.6
```

```
summarize(datos.credito_genero, "Media Ingreso (CRC)" = mean(monto_ingreso) *
```

```
## # A tibble: 2 x 2
##   Genero    `Media Ingreso (CRC)`
## * <fct>                     <dbl>
## 1 Femenino              24622638.
## 2 Masculino             25218931.
```

```
summarize(datos.credito_genero, "Media Ingreso (CRC)" = mean(monto_ingreso) * 619, "M
```

```
## # A tibble: 2 x 3
##   Genero    `Media Ingreso (CRC)` `Media edad`
## * <fct>                     <dbl>        <dbl>
## 1 Femenino              24622638.         55.7
## 2 Masculino             25218931.         55.6
```

```
datos.credito_gen_casado <- group_by(datos.credito, Genero, Estudiante) ## Agrupamos

resumen_gen_casado <- summarize(datos.credito_gen_casado, "Cantidad de individuos" = 

resumen_gen_casado
```

```
## # A tibble: 4 x 3
## # Groups:   Genero [2]
##   Genero    Estudiante `Cantidad de individuos`
##   <fct>     <fct>                           <int>
## 1 Femenino  No                                183
## 2 Femenino  Si                                 24
## 3 Masculino No                                177
## 4 Masculino Si                                 16
```

```
library(lubridate) ## Se utiliza la función year()

datos.credito_anho <- group_by(datos.credito, "Fecha inicio" = year(fecha_ini), Gener

resumen.anho <- summarize(datos.credito_anho, "Media balance" = mean(monto_balance),

resumen.anho
```

```
## # A tibble: 11 x 4
## # Groups:   Fecha inicio [6]
##    `Fecha inicio` Genero    `Media balance` `Cantidad de individuos`
##             <dbl> <fct>               <dbl>                    <int>
##  1           2011 Femenino            1448                        1
##  2           2016 Femenino             666.                       5
##  3           2016 Masculino            371.                      16
##  4           2017 Femenino             501.                      18
##  5           2017 Masculino            456.                      17
##  6           2018 Femenino             630.                      25
##  7           2018 Masculino            564.                      16
##  8           2019 Femenino             591.                      83
##  9           2019 Masculino            459.                      70
## 10           2020 Femenino             414.                      75
## 11           2020 Masculino            588.                      74
```

```
resumen.anho <- arrange(resumen.anho, desc(`Fecha inicio`))  ## Ordenamos por

resumen.anho
```

```
## # A tibble: 11 x 4
## # Groups:   Fecha inicio [6]
##    `Fecha inicio` Genero    `Media balance` `Cantidad de individuos`
##             <dbl> <fct>               <dbl>                    <int>
##  1           2020 Femenino             414.                      75
##  2           2020 Masculino            588.                      74
##  3           2019 Femenino             591.                      83
##  4           2019 Masculino            459.                      70
##  5           2018 Femenino             630.                      25
##  6           2018 Masculino            564.                      16
##  7           2017 Femenino             501.                      18
##  8           2017 Masculino            456.                      17
##  9           2016 Femenino             666.                       5
## 10           2016 Masculino            371.                      16
## 11           2011 Femenino            1448                        1
```

12.8. Operador pipe %>%:

El operador %>% es muy útil a la hora de utilizar funciones del paquete **dplyr** de forma consecutiva, primero recordemos que el resultado de un función de este paquete siempre es un data frame, por lo que es posible (y muy usual) aplicar varias funciones, pero si lo hacemos de forma anidada es un poco confuso de leer, pues se vería de esta forma

```
> tercera(segunda(primera(dataframe)))
```

Esta lógica anidada no es la forma más natural de pensar, por lo el operador %>% no es permite escribir las operaciones en forma de secuencia de izquierda a derecha, es decir

```
> primera(dataframe) %>% segunda %>% tercera
```

```
datos.credito %>% select(monto_balance, monto_ingreso, monto_limite, calif_credit)
```

```
##   monto_balance monto_ingreso monto_limite calif_credit
## 1          1999        182728        13913          982
## 2          1809        186634        13414          949
## 3          1779        152298        12066          828
```

```
## También se pueden ver los últimos usando tail(n)
datos.credito %>% select(monto_balance, monto_ingreso, monto_limite, calif_credit)
```

```
##     monto_balance monto_ingreso monto_limite calif_credit
## 398             0         13444          886          121
## 399             0         14084          855          120
## 400             0         12414          855          119
```

El último ejemplo que hicimos en la sección de **group_by()** se puede reescribir de forma más sencilla utilizando %>% de la siguiente forma:

```
group_by(datos.credito, "Fecha inicio" = year(fecha_ini), Genero) %>% summari
```

```
## # A tibble: 11 x 4
## # Groups:   Fecha inicio [6]
##    `Fecha inicio` Genero    `Media balance` `Cantidad de individuos`
##             <dbl> <fct>               <dbl>                    <int>
##  1           2020 Femenino             414.                       75
##  2           2020 Masculino            588.                       74
##  3           2019 Femenino             591.                       83
##  4           2019 Masculino            459.                       70
##  5           2018 Femenino             630.                       25
##  6           2018 Masculino            564.                       16
##  7           2017 Femenino             501.                       18
##  8           2017 Masculino            456.                       17
##  9           2016 Femenino             666.                        5
## 10           2016 Masculino            371.                       16
## 11           2011 Femenino            1448                         1
```

Es bueno aclarar que este operador **no** es exclusivo para funciones del paquete **dplyr** se puede usar siempre que escribamos código siempre y cuando tenga sentido, es decir, que el resultado de la operación anterior sea compatible con el insumo de la función siguiente.

Para escribir este operador de forma rápida utilizamos el shortcut: **CTRL+SHIFT+M**.

Capítulo 13

Visualización: Paquete ggplot2

En el proceso de análisis de datos la visualización de datos está presente en varias de sus etapas entre las principales están: la exploración de los datos y la presentación de los resultados, los gráficos permiten, de forma intuitiva, encontrar y entender patrones, grupos o valores atípicos que puedan existir en los datos, una vez terminado el proceso de exploración o predicción los gráficos nos permiten comunicar los resultados a nuestra audiencia sin importar si esta tiene o no conocimientos en técnicos.

13.1. Gramática de los gráficos

Un gráfico realizado con ggplot2 presenta, al menos, tres elementos:

1. Datos (Data) que queremos representar (que serán un data frame).
2. Características estéticas (aesthetic mappings) que describen cómo queremos que los datos se vean en el gráfico. Como se ve más adelante, se introducen con la función aes() y se refieren a:

- posición (en los ejes)
- color exterior (color) y de relleno (fill)
- forma de puntos (shape)
- tipo de línea (linetype)

- tamaño (size)

3. Objetos geométricos (Geom) representan lo que vemos en un gráficos (puntos, líneas, etc.). Todo gráfico tiene, como mínimo, una geometría. La geometría determina el tipo de gráfico:

- geom_point (para puntos)
- geom_lines (para lineas)
- geom_histogram (para histograma)
- geom_boxplot (para boxplot)
- geom_bar (para barras)
- geom_smooth (líneas suavizadas)
- geom_polygons (para polígonos en un mapa)

Por tanto, para construir un gráfico con ggplot2 comenzamos con la siguiente estructura de código:

** ggplot(datos, aes()) + geom_tipo() **

A partir de esta estructura básica puede mejorarse la presentación de los gráficos introduciendo, por ejemplo, características estéticas en los objetos geométricos, agregando títulos a los gráficos, etc.

Otros elementos que conviene tener presente en un gráfico de ggplot2 son:

- Stat (Stat), transformaciones estadísticas para, generalmente, resumir datos (por ejemplo: contar frecuencias, número de intervalos en los histogramas, etc.).

- Escalas (Scale). Las escalas, por ejemplo, convierten datos en características estéticas (colores, etc.), crean leyendas... . - Coordenadas (coord): sistema de coordenadas cartesianas, polares, proyecciones, etc.

- Faceting (Faceting), permite representar gráficos separados para subconjuntos de los datos originales.

Para a realizar algunos gráficos con ggplot primero cargamos la librería ggplot2. Si no está instalado el paquete lo instalamos.

```
# install.packages("ggplot2")
library(ggplot2)
```

En los ejemplos que siguen tratamos de ir introduciendo poco a poco distintos elementos y argumentos para mejorar la apariencia de los gráficos.

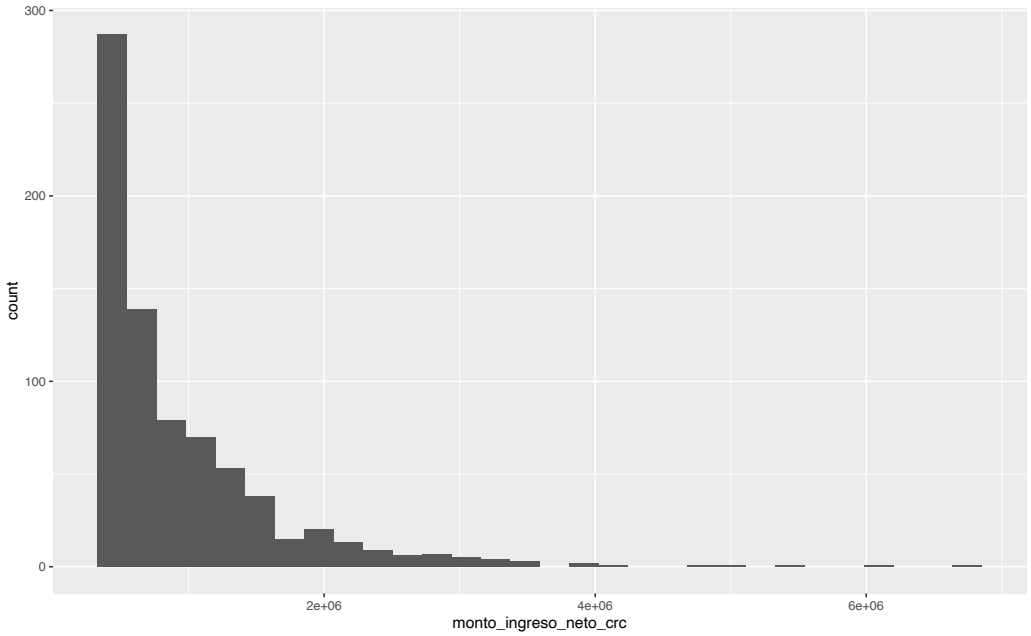
13.2. Histogramas

Vamos a comenzar haciendo un histograma muy sencillo de os instrumentos del portafolio del banco. Para esto, recordemos que la instrucción comienza con la función ggplot(), en la que incluimos los datos y la estética con la que queremos que se presenten en el gráfico. Seguidamente le añadimos (+) la geometría (tipo histograma) con la función geom_histogram().

Muy importante: con ggplot2 añadimos capas (layers) con el símbolo +.

El histograma es el siguiente:

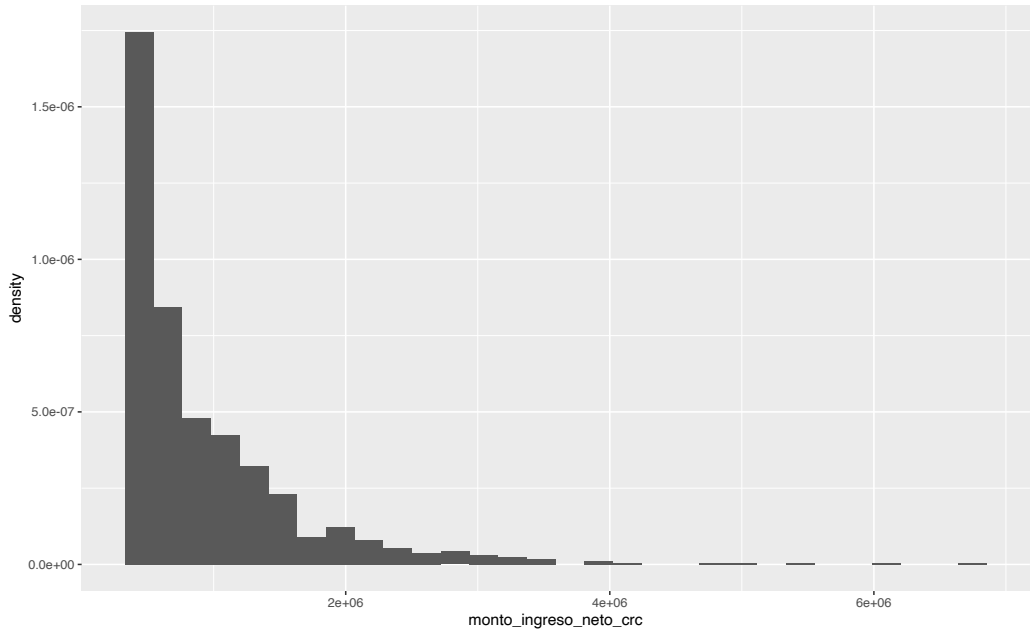
```
ggplot(datos.ingresos, aes(x = monto_ingreso_neto_crc)) +
  geom_histogram()
```



Dos cosas a considerar del histograma anterior:

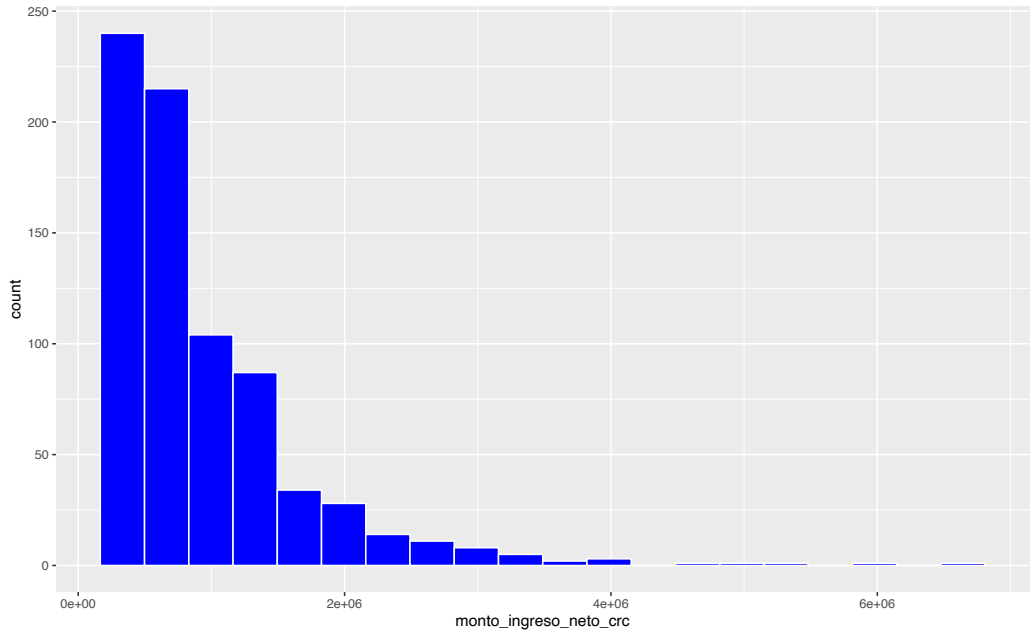
- Por defecto, el número de intervalos es de 30. Es posible establecer el número de intervalos (bins), la amplitud del intervalo (binwidth) o fijar los puntos de corte de los intervalos (breaks).
- El eje Y corresponde al número de observaciones (frecuencias absolutas). Si estamos interesados en representar un histogramas de forma que el área del mismo sume 1, entonces tenemos que cambiar la estética de la siguiente forma:

```
ggplot(datos.ingresos, aes(x = monto_ingreso_neto_crc)) +
  geom_histogram(aes(y = ..density..))
```



A partir de esta estructura básica, podemos ir añadiendo elementos para mejorar la presentación.

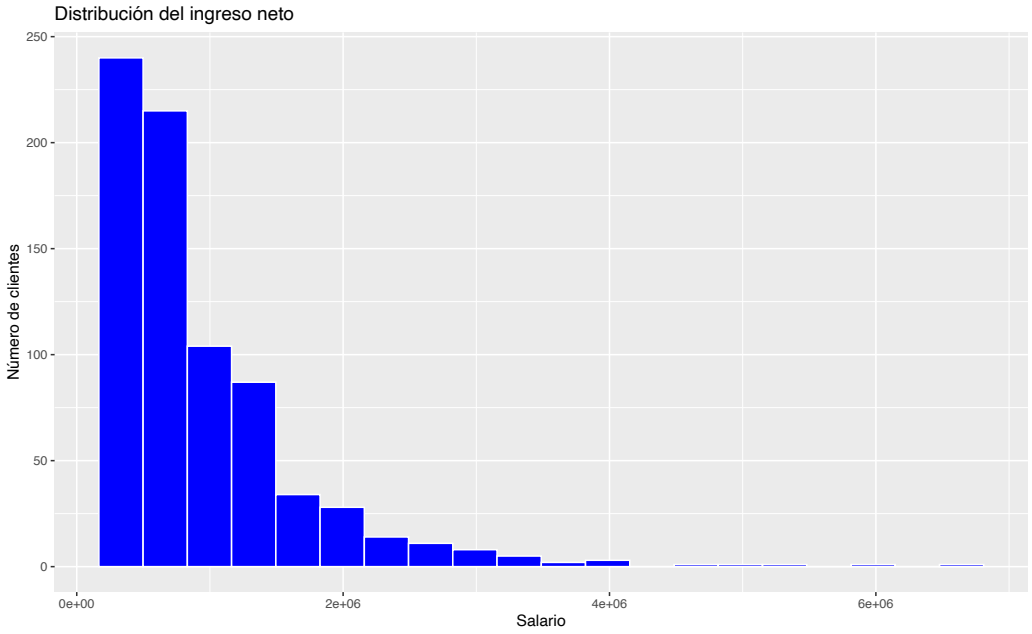
```
# Histograma con 20 intervalos
ggplot(datos.ingresos, aes(x = monto_ingreso_neto_crc)) +
  geom_histogram(bins = 20, color = "white", fill = "blue")
```



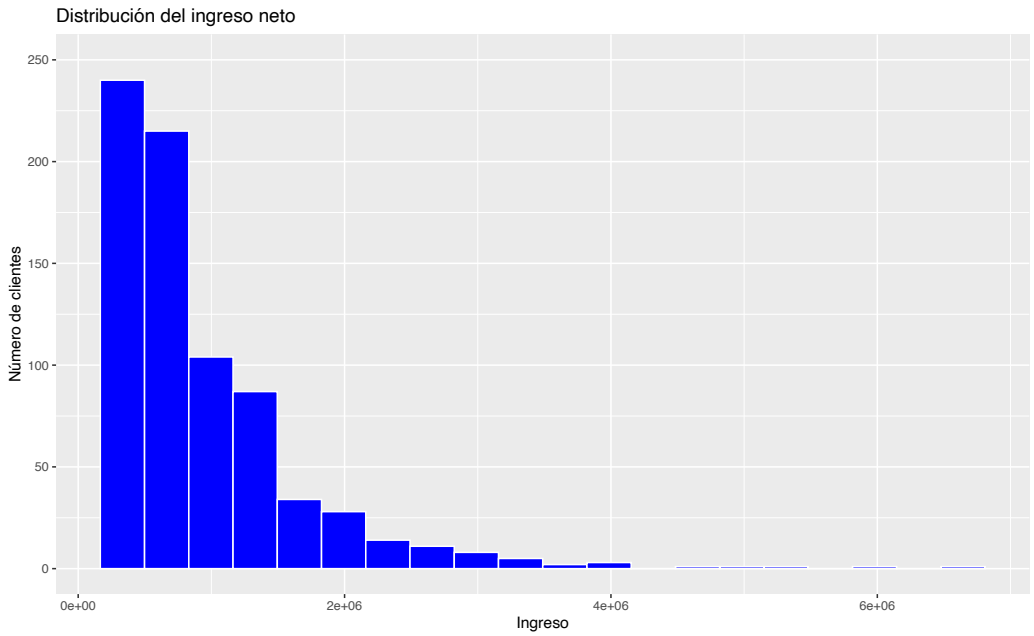
Ahora vamos a insertar un título al gráfico y también rotularemos los ejes. Para modificar las etiquetas de los ejes se utilizan las funciones xlab() y ylab(). Si, por ejemplo, quisiéramos omitir la etiqueta del eje Y agregamos: ylab(NULL).

También se pueden modificar los límites de los ejes, para esto se utilizan las funciones xlim() y ylim().

```
ggplot(datos.ingresos, aes(x = monto_ingreso_neto_crc)) +
  geom_histogram(bins = 20, color = "white", fill = "blue") +
  ggtitle("Distribución del ingreso neto ") +
  xlab("Salario") +
  ylab("Número de clientes")
```



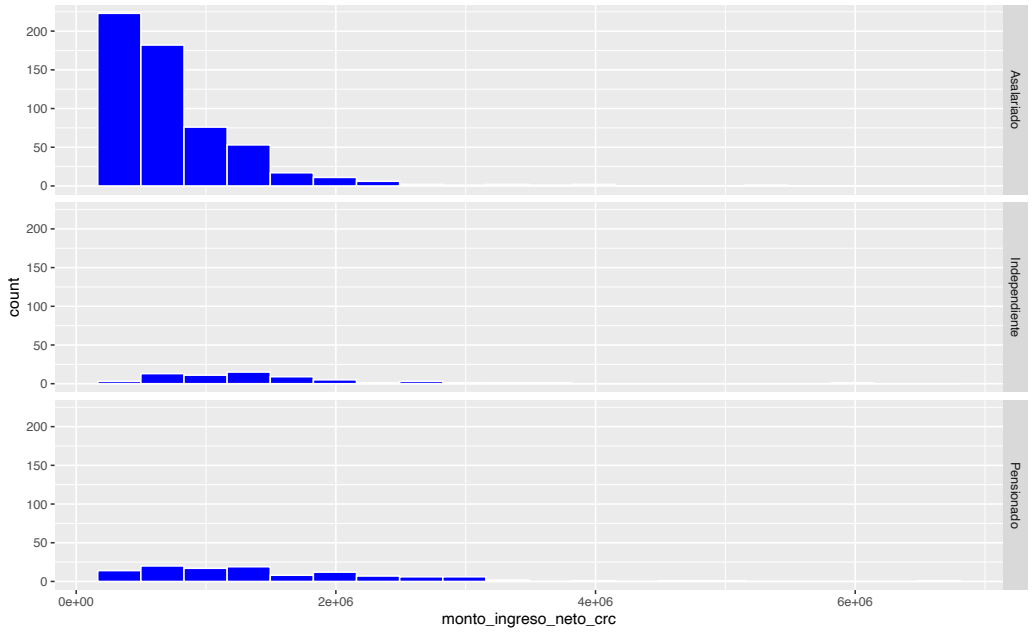

```
# Una alternativa al anterior sería:

ggplot(datos.ingresos, aes(x = monto_ingreso_neto_crc)) +
  geom_histogram(bins = 20, color = "white", fill = "blue") +
  labs(title = "Distribución del ingreso neto",
    x = "Ingreso",
    y = "Número de clientes") +
  ylim(c(0, 250))
```



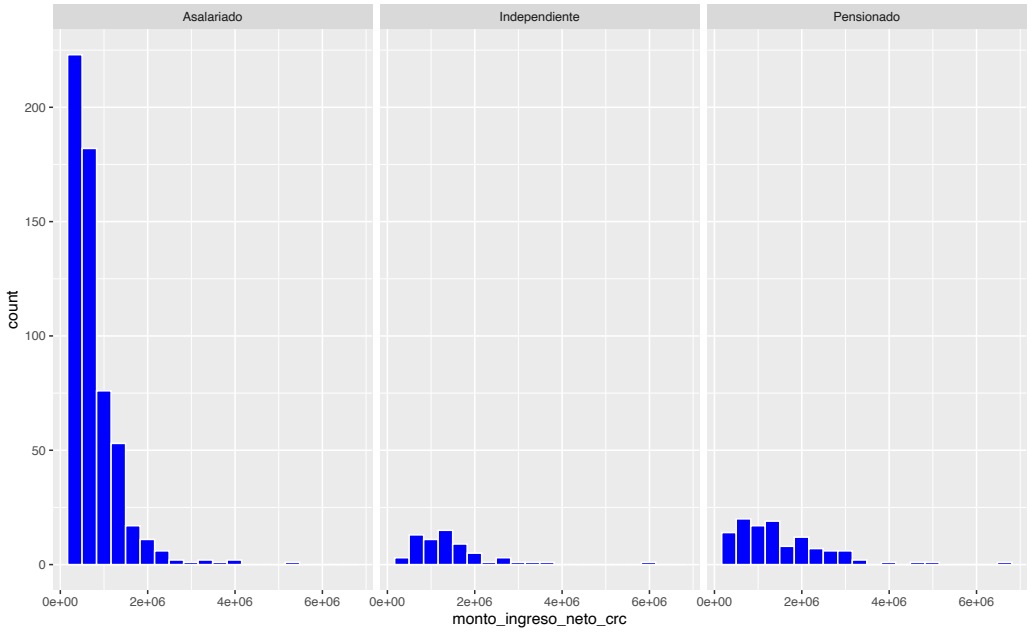
Para tratar de observar las diferencias en la distribución del salario según distintos grupo, por ejemplo, el género podemos:

Visualizar cada subconjunto de datos (ingreso neto para asalariados e independientes) en distintos paneles. Para ello, utilizamos el elemento facet.

```
ggplot(datos.ingresos, aes(x = monto_ingreso_neto_crc)) +
  geom_histogram(bins = 20, color = "white", fill = "blue") +
  facet_grid(tipo_empleado ~ .)
```

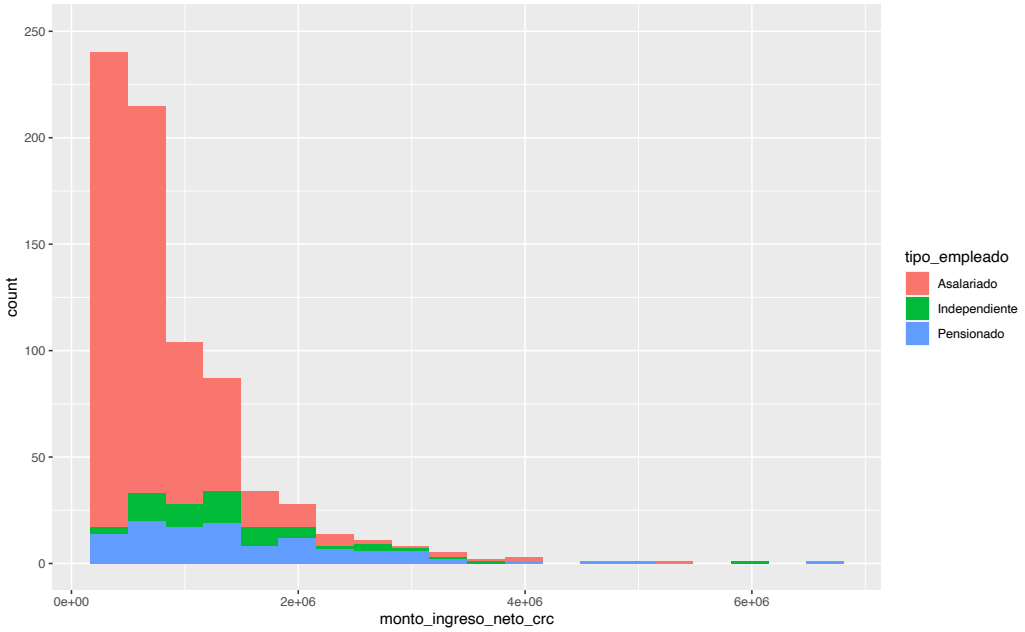



```
ggplot(datos.ingresos, aes(x = monto_ingreso_neto_crc)) +
  geom_histogram(bins = 20, color = "white", fill = "blue") +
  facet_wrap(~tipo_empleado)
```



Hacemos el histograma pero usamos el tipo_empleado de cliente para colorear las partes que de cada intervalo corresponden a asalariados, independientes y a pensionados.

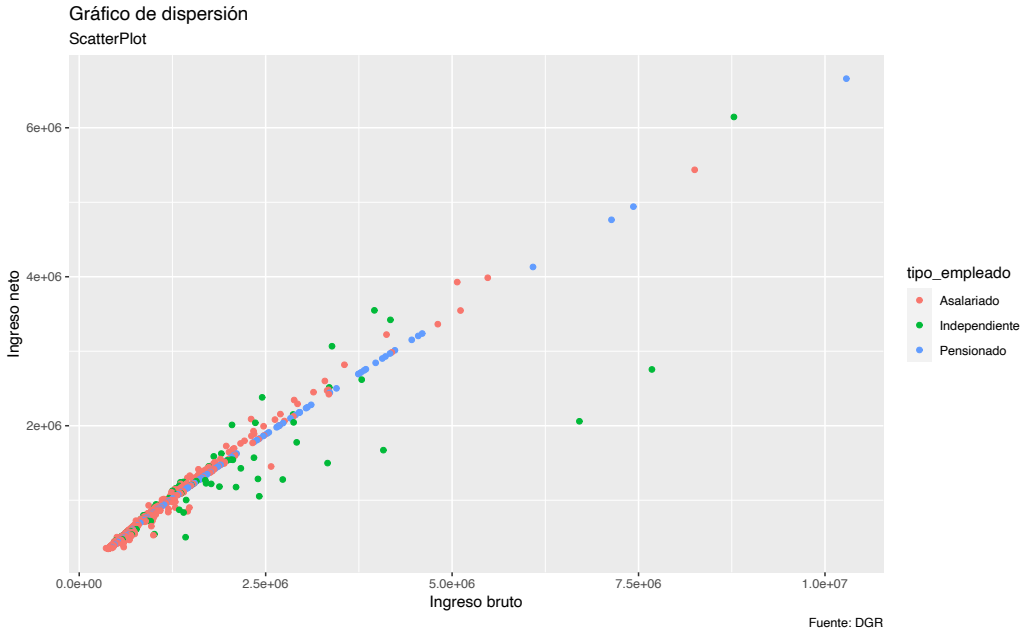
```
ggplot(datos.ingresos, aes(x = monto_ingreso_neto_crc, fill = tipo_empleado)) +
  geom_histogram(bins = 20) +
  ylim(c(0, 250))
```



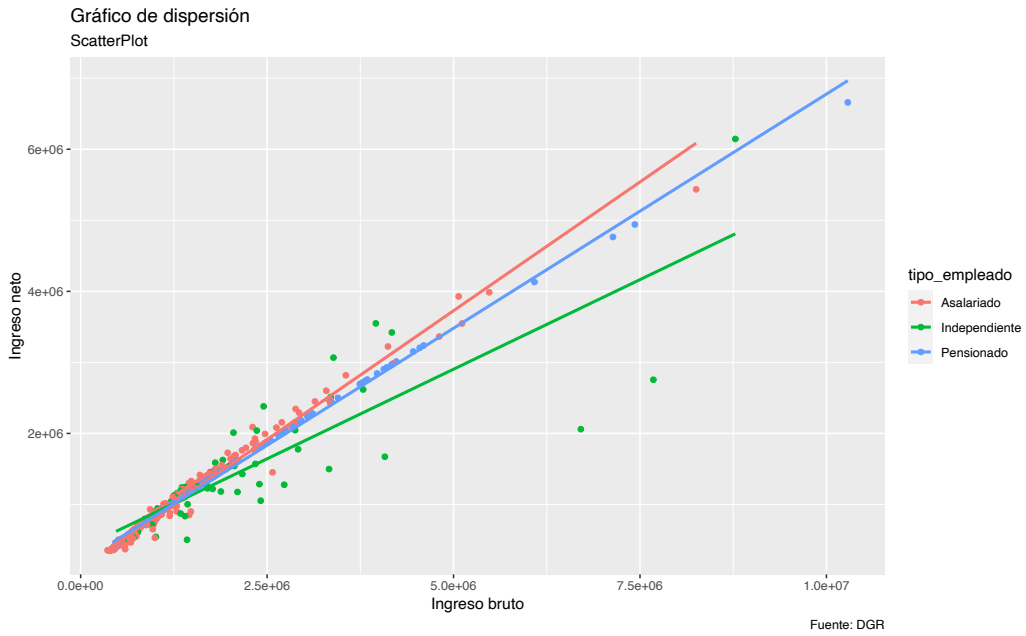

```
ggplot(datos.ingresos, aes(x = monto_ingreso_neto_crc)) +
  geom_histogram(bins = 10, aes(fill = tipo_empleado), position = "fill", alph
  labs(x = "Ingreso neto", y = "Clientes", fill = "tipo_empleado") +  # título
  scale_fill_discrete(labels = c("Asalariado", "Independiente", "Pensionado"))
```

13.3. Gráficos de dispersión

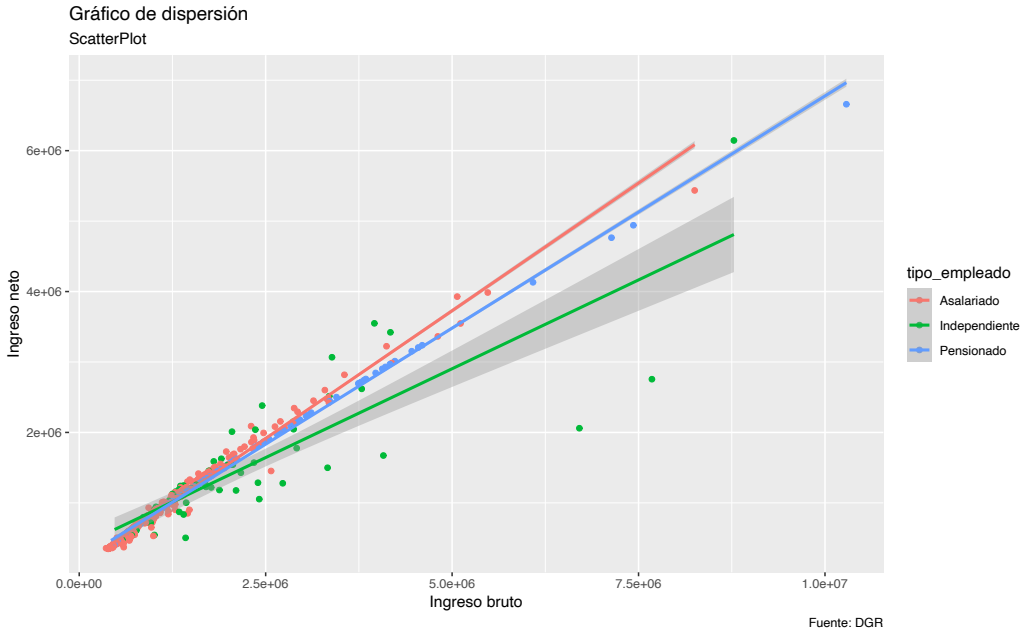
```
# Opción 1
ggplot(datos.ingresos, aes(monto_ingreso_bruto_crc, monto_ingreso_neto_crc, color = t
  geom_point() +
  labs(title = "Gráfico de dispersión",
    subtitle = "ScatterPlot",
    caption = "Fuente: DGR",
    x = "Ingreso bruto",
    y = "Ingreso neto") +
  scale_color_discrete(name = "tipo_empleado", labels = c("Asalariado", "Independient
```




```
ggplot(datos.ingresos, aes(monto_ingreso_bruto_crc, monto_ingreso_neto_crc, co
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  labs(title = "Gráfico de dispersión",
    subtitle = "ScatterPlot",
    caption = "Fuente: DGR",
    x = "Ingreso bruto",
    y = "Ingreso neto") +
  scale_color_discrete(name = "tipo_empleado", labels = c("Asalariado", "Indep
```

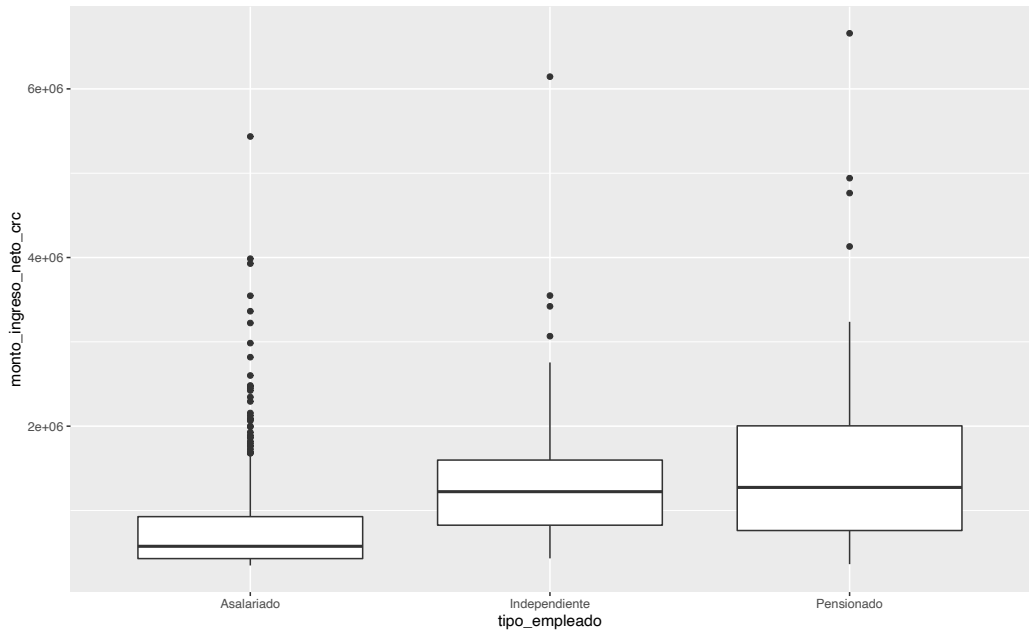



```
ggplot(datos.ingresos, aes(monto_ingreso_bruto_crc, monto_ingreso_neto_crc, color = t
  geom_point() +
  geom_smooth(method = "lm") +
  labs(title = "Gráfico de dispersión",
    subtitle = "ScatterPlot",
    caption = "Fuente: DGR",
    x = "Ingreso bruto",
    y = "Ingreso neto") +
  scale_color_discrete(name = "tipo_empleado", labels = c("Asalariado", "Independient
```

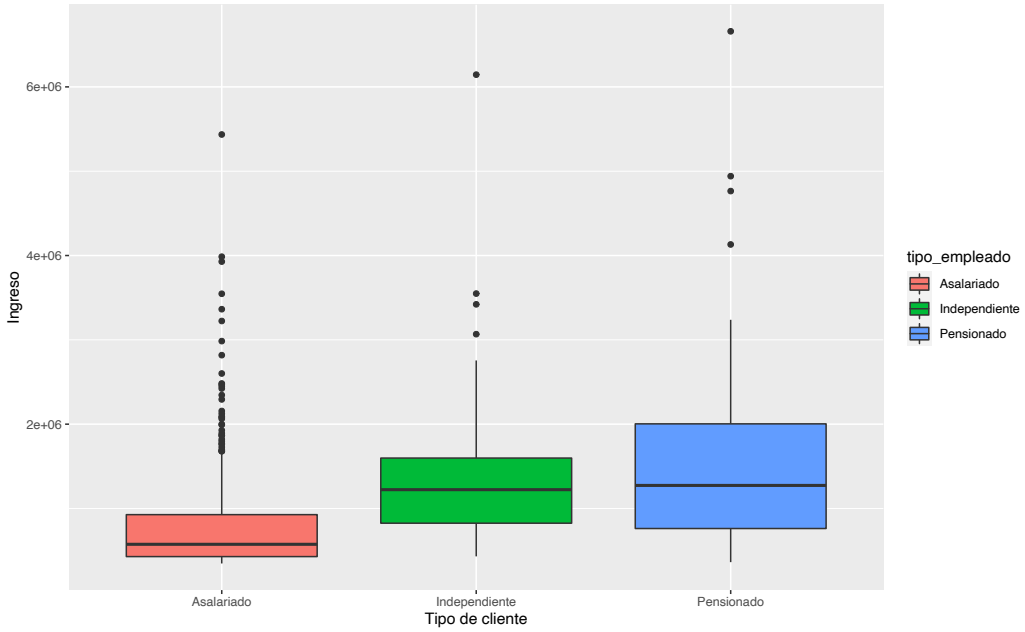


13.4. Gráficos de cajas

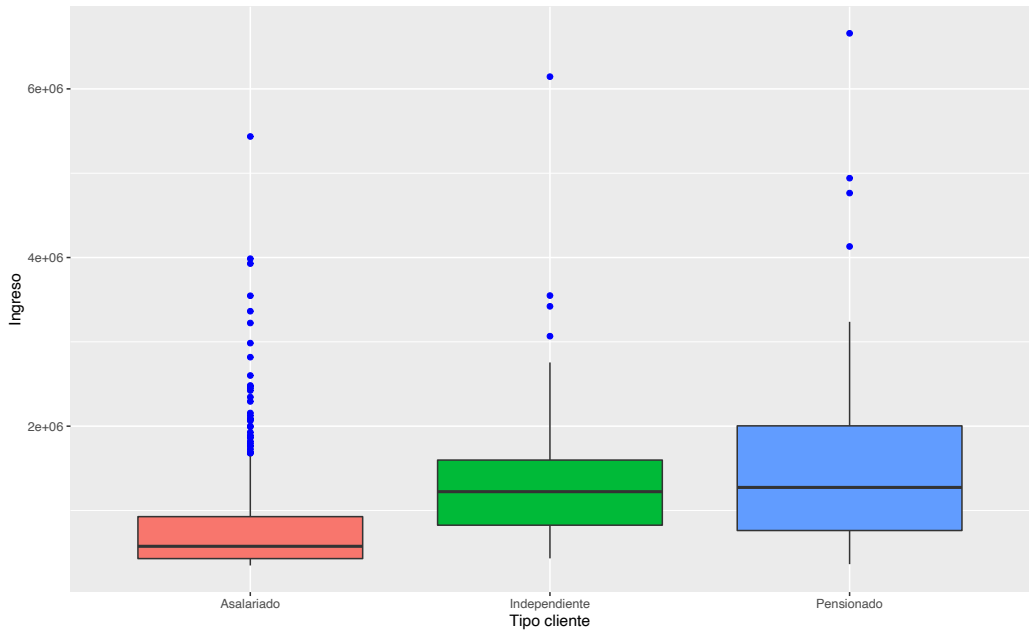
```
ggplot(datos.ingresos, aes(x = tipo_empleado, y = monto_ingreso_neto_crc)) +
  geom_boxplot()
```




```
# Diagrama de caja con color de relleno
ggplot(datos.ingresos, aes(x = tipo_empleado, y = monto_ingreso_neto_crc, fill = tipo
  geom_boxplot() +
  labs(x = "Tipo de cliente", y = "Ingreso", fill = "tipo_empleado") +   # titulo eje
  scale_x_discrete(labels = c("Asalariado", "Independiente", "Pensionado")) +   # eti
  scale_fill_discrete(labels = c("Asalariado", "Independiente", "Pensionado"))  # eti
```

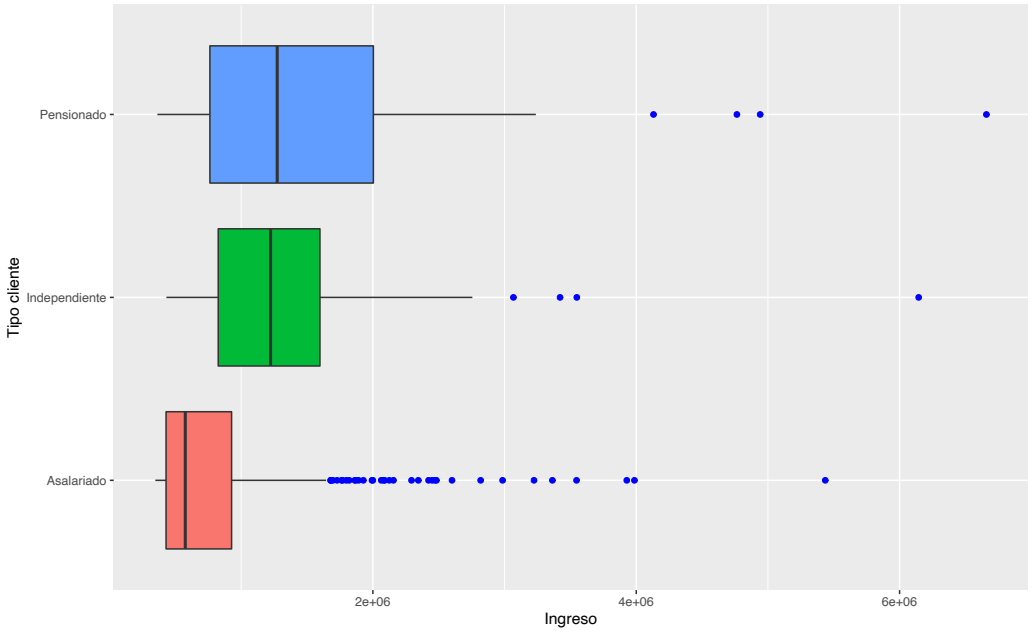



```
# Diagrama de caja sin leyenda por considerar redundante la información

ggplot(datos.ingresos, aes(x = tipo_empleado, y = monto_ingreso_neto_crc, fill
  geom_boxplot(outlier.colour = "blue") +             # color de los outliers
  labs(x = "Tipo cliente", y = "Ingreso") +
  scale_x_discrete(labels = c("Asalariado", "Independiente", "Pensionado")) +
  guides(fill = FALSE)
```




```
# eliminamos la leyenda
```

```
ggplot(datos.ingresos, aes(x = tipo_empleado, y = monto_ingreso_neto_crc, fill = tipo
  geom_boxplot(outlier.colour = "blue") +
  labs(x = "Tipo cliente", y = "Ingreso") +
  scale_x_discrete(labels = c("Asalariado", "Independiente", "Pensionado")) +
  guides(fill = FALSE) +
  coord_flip()                          # cambio dirección de las cajas
```

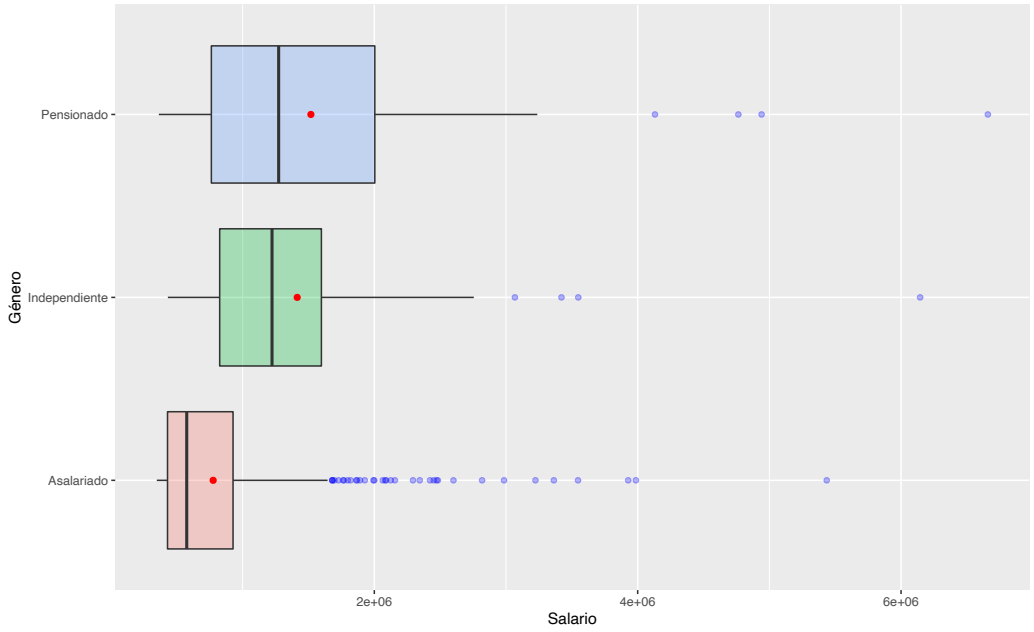


Puede resultar interesante localizar la media de cada grupo en los diagramas de caja. Para ello, hacemos uso del elemento stat (transformación estadística). Se puede añadir stat, básicamente, de dos formas:

1. Añadir directamente una función stat_() y de esa forma anular su valor por defecto en la geometría.
2. Añadir una función geom_() e introducir el elemento stat para anular el valor por defecto.

Veamos las dos formas comentadas de añadir.

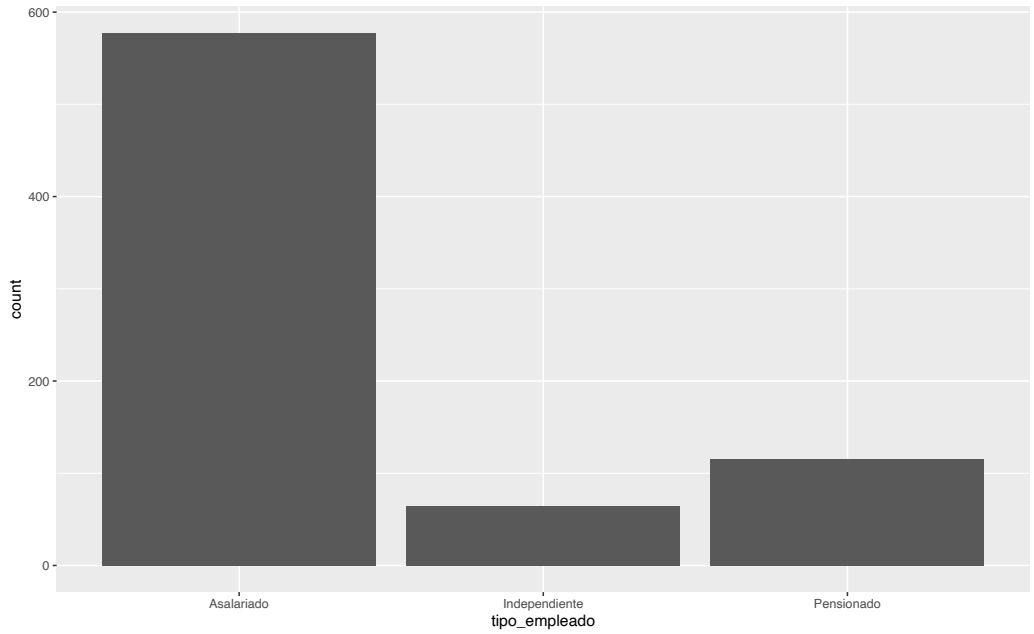
```
# Añadimos la media utilizando la función geom_() y modificando el elemento
ggplot(datos.ingresos, aes(x = tipo_empleado, y = monto_ingreso_neto_crc, fill
  geom_boxplot(alpha = 0.3, outlier.colour = "blue") +
  labs(x = "Género", y = "Salario") +
  scale_x_discrete(labels = c("Asalariado", "Independiente", "Pensionado")) +
  guides(fill = FALSE) +
  coord_flip() +
  geom_point(stat = "summary", fun.y = mean, shape = 16, size = 2, color = "re
```



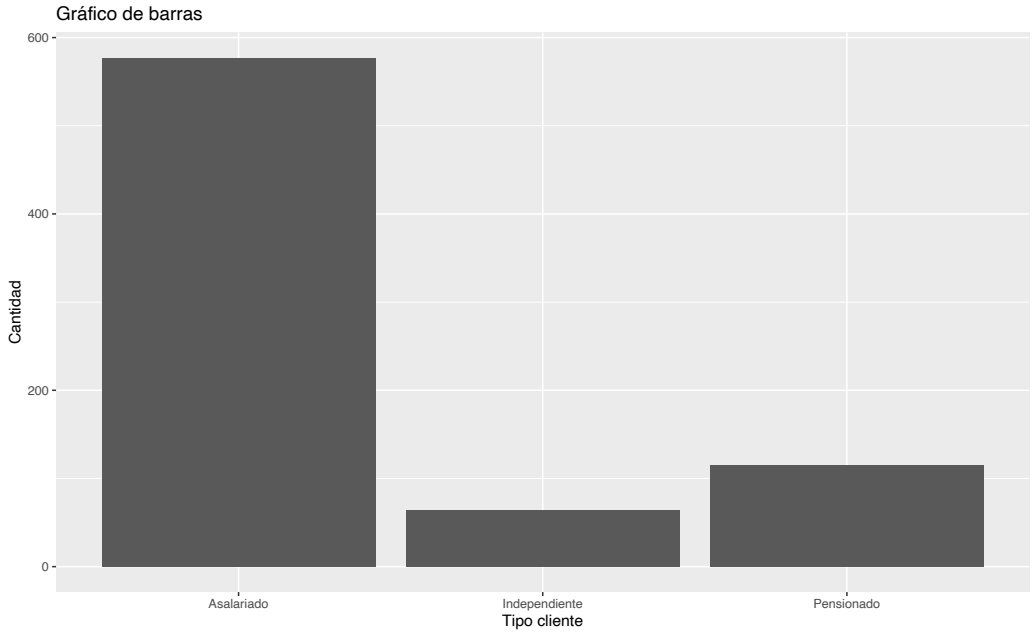
13.5. Gráficos de barras

Los gráficos de barras, se utilizan comúnmente para representar variables categóricas (atributos u ordinales) y variables cuantitativas discretas.

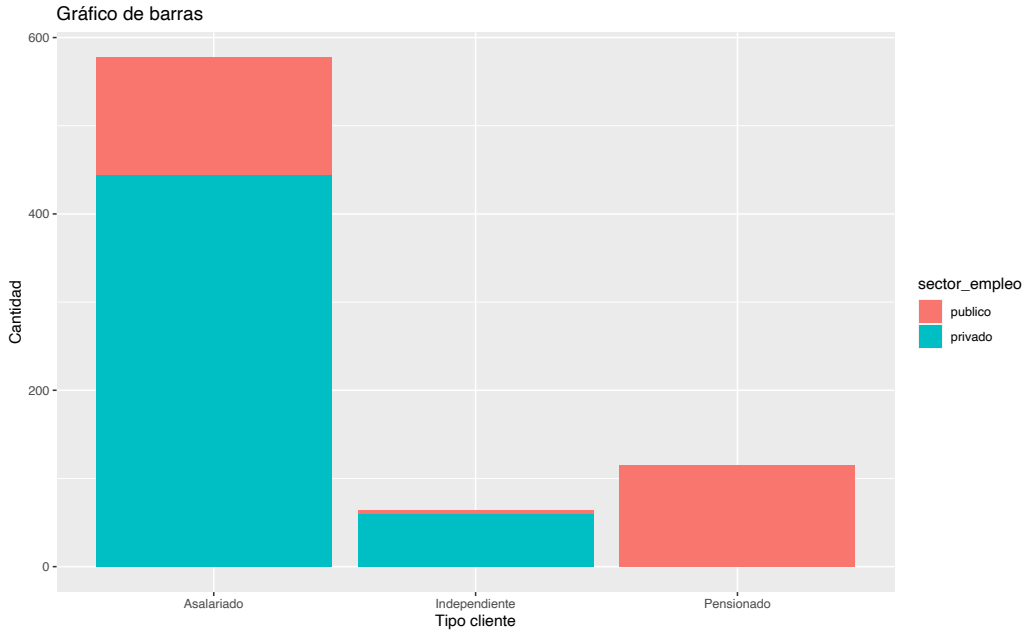
```
ggplot(datos.ingresos, aes(tipo_empleado)) +
  geom_bar()
```



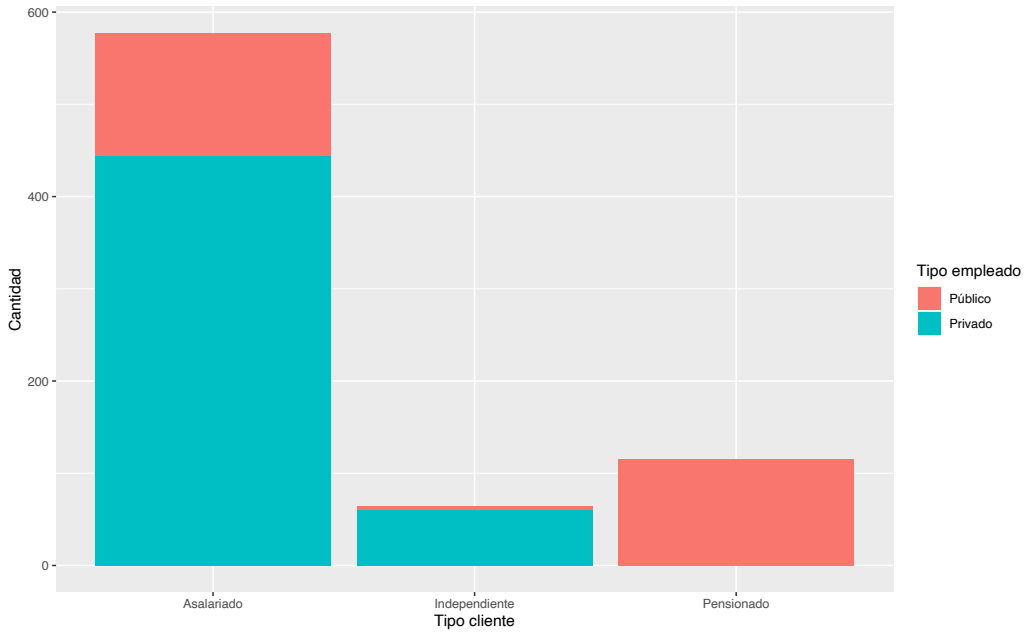

```
ggplot(datos.ingresos, aes(tipo_empleado)) +
  geom_bar() +
  labs(title = "Gráfico de barras",
    x = "Tipo cliente",
    y = "Cantidad")
```



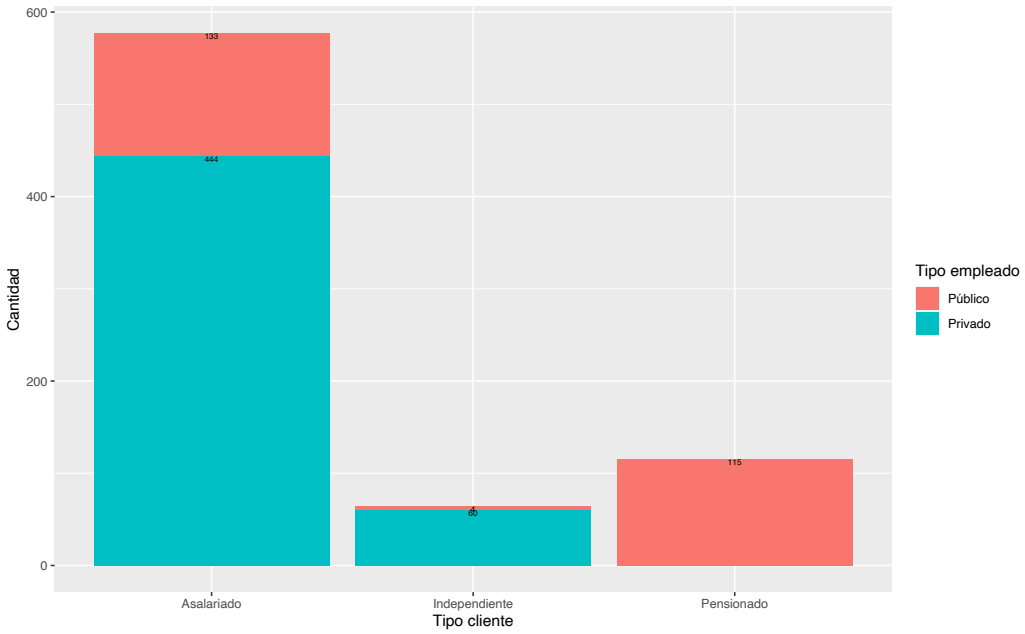

```
ggplot(datos.ingresos, aes(tipo_empleado, fill = sector_empleo)) +
  geom_bar() +
  labs(title = "Gráfico de barras",
    x = "Tipo cliente",
    y = "Cantidad")
```



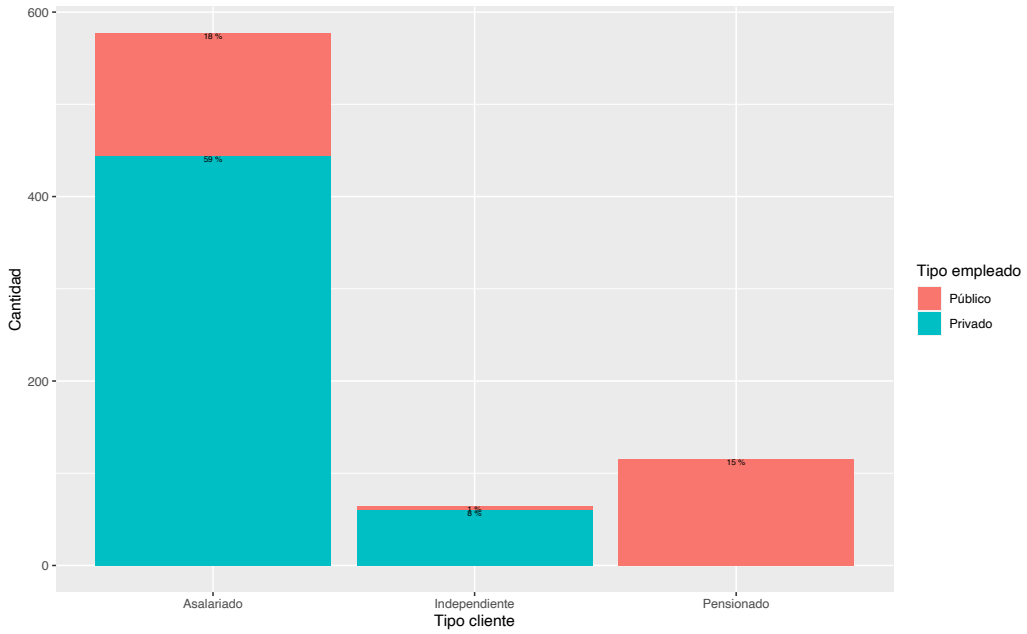

```
ggplot(datos.ingresos, aes(x = tipo_empleado, fill = sector_empleo))  +
  geom_bar() +
  labs(x = "Tipo cliente", y = "Cantidad") +
  scale_x_discrete(labels = c("Asalariado", "Independiente", "Pensionado")) +
  scale_fill_discrete("Tipo empleado",
    labels = c("Público", "Privado"))
```



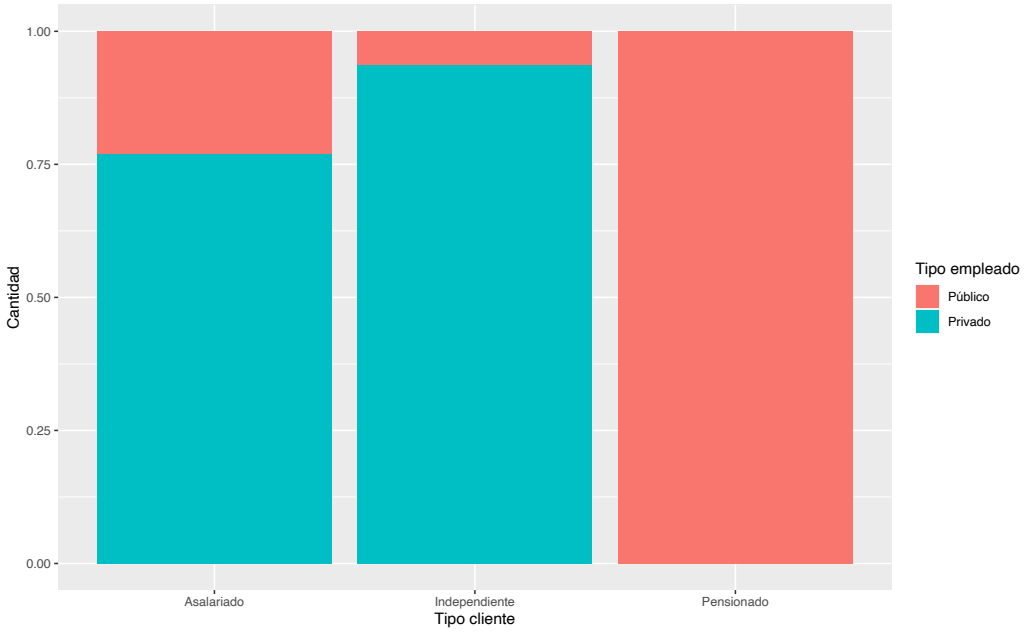

```
ggplot(datos.ingresos, aes(x = tipo_empleado, fill = sector_empleo))  +
  geom_bar() +
  labs(x = "Tipo cliente", y = "Cantidad") +
  scale_x_discrete(labels = c("Asalariado", "Independiente", "Pensionado")) +
  scale_fill_discrete("Tipo empleado",
    labels = c("Público", "Privado")) +
  geom_text(stat = "count", aes(label = ..count..),
    position = "stack",
    vjust = 1,
    size = 2,
    color = "black")
```



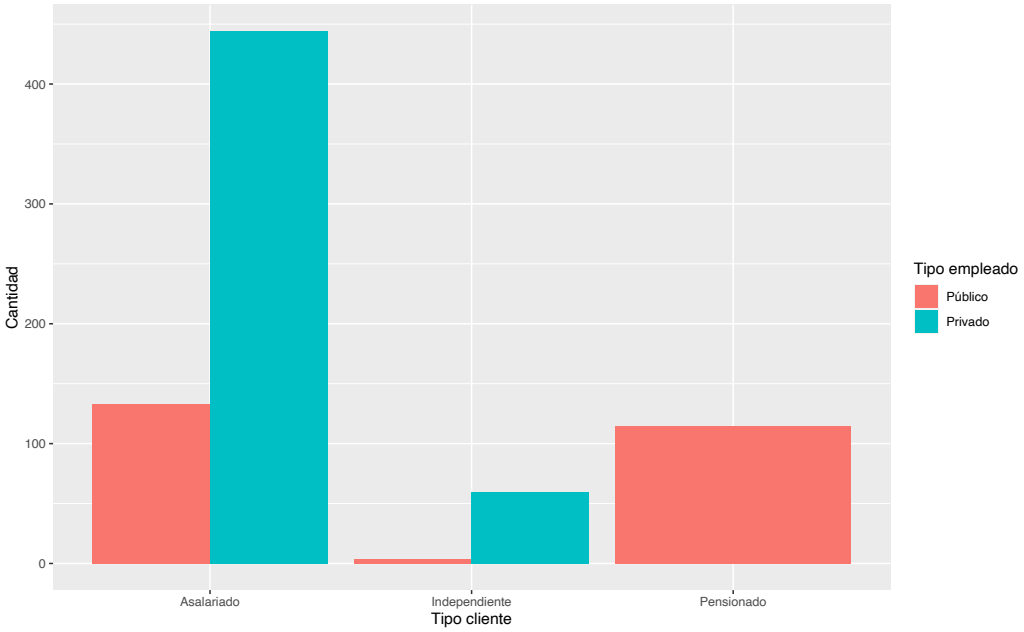

```
ggplot(datos.ingresos, aes(x = tipo_empleado, fill = sector_empleo))  +
  geom_bar() +
  labs(x = "Tipo cliente", y = "Cantidad") +
  scale_x_discrete(labels = c("Asalariado", "Independiente", "Pensionado")) +
  scale_fill_discrete("Tipo empleado",
    labels = c("Público", "Privado")) +
  geom_text(stat = "count",
    aes(label = paste(round((..count..) / sum(..count..) * 100), "%")),
    position = "stack",
    vjust = 1,
    size = 2,
    color = "black")
```




```
ggplot(datos.ingresos, aes(x = tipo_empleado, fill = sector_empleo))  +
  geom_bar(position = "fill") +
  labs(x = "Tipo cliente", y = "Cantidad") +
  scale_x_discrete(labels = c("Asalariado", "Independiente", "Pensionado")) +
  scale_fill_discrete("Tipo empleado",
    labels = c("Público", "Privado"))
```




```
ggplot(datos.ingresos, aes(x = tipo_empleado, fill = sector_empleo))  +
  geom_bar(position = "dodge") +
  labs(x = "Tipo cliente", y = "Cantidad") +
  scale_x_discrete(labels = c("Asalariado", "Independiente", "Pensionado")) +
  scale_fill_discrete("Tipo empleado",
    labels = c("Público", "Privado"))
```


400
300
200
100
0
Cantidad
Asalariado
Independiente
Pensionado
Tipo cliente
Tipo empleado
Público
Privado