

Introducción al manejo de datos con R

Curso: Manejo de datos y reportería con R

Néstor Montaña

Sociedad Ecuatoriana de Estadística

Enero-2021



Nota:

Con *Alt + F* o *Option + F* puede hacer que estas diapositivas ocupen todo el navegador (es decir que se ignore el aspecto de diapositiva que tiene por default la presentación)

Introducción a R y RStudio

Curso: Manejo de datos y reportería con R

Néstor Montaña

Instalar

R

- En windows y Mac descargar R desde [CRAN](#)
- Instalarlo como cualquier otro software
- En Linux, distribuciones basadas en Debian/Ubuntu tienen R en los repositorios oficiales
- En Linux, distribuciones basadas en Fedora/RedHat deben habilitar EPEL para tener R
- En Debian Estable para tener nuevas versiones se debe utilizar un ["backports"](#)

RStudio

- Debe estar R ya instalado
- Descargar según Sistema Operativo [web oficial](#)
- En Windows/Mac es next&next
- En linux Debian/Ubuntu/Mint se puede instalar desde el .deb y next&next
- En linux desde consola hay que seguir las instrucciones de la web oficial
- En linux Red Hat/CentOS/Fedora tener cuidado con las dependencias

¿Por qué R?



**¡Corre!, de nuevo
viene a hablarnos de**



R: Algo de historia

Mientras S cambiaba de dueño y denominación , Ross Ihaka y Robert Gentleman, decidieron implementar su propio dialecto, era 1991 cuando estos dos neozelandeses crearon R.

Tardaron dos años en anunciarlo públicamente y otros dos años más en licenciarlo bajo GPL, ésta última decisión fue posiblemente la responsable de que hoy R tenga tanta repercusión.

Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. Journal of Computational and Graphical Statistics, 5(3):299–314, 1996

¿Por qué R?

- Software Libre (Open Source), gratuito y de desarrollo independiente,
- Uno de los lenguajes más usados para Data Science,
- Posee uno de los mejores sistemas gráficos existentes,
- Gran y creciente cantidad de usuarios y desarrolladores,
- CRAN: +12000 paquetes disponibles y aparte hay paquetes instalables desde GitHub
- Curva de aprendizaje simple (desde que apareció el tidyverse),
- Actualmente la mayoría de U en las carreras no computacionales enseñan análisis de datos con R.

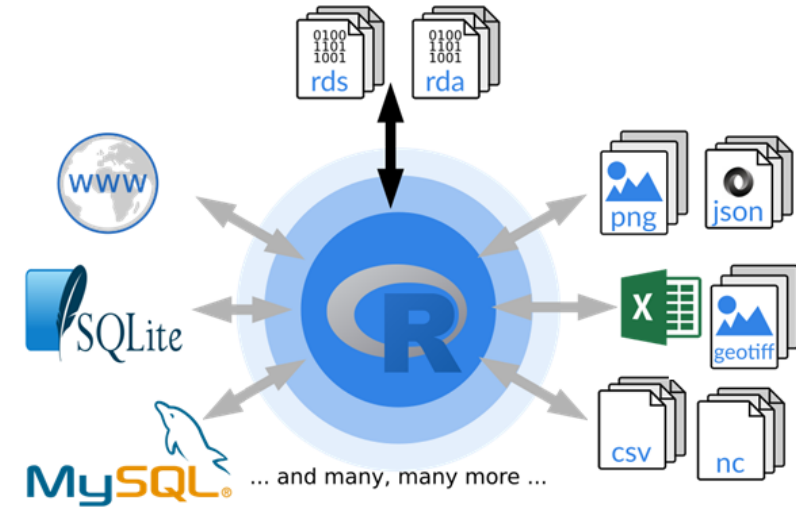


Comunidad R

fuelle: [Open Data Science - Medium](#)

¿Por qué R?

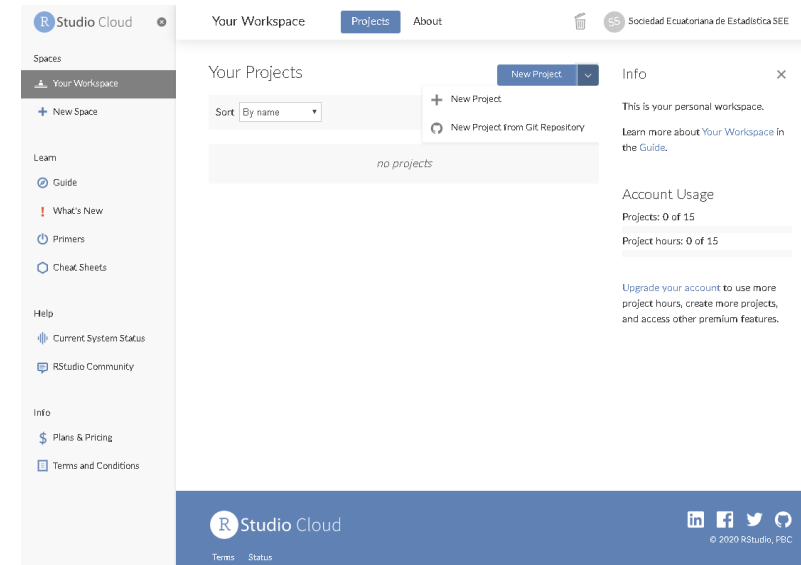
- Flexible y se integra bien con otros sistemas u aplicaciones: Bases de datos, software de inteligencia de negocios, etc.
- Rico ecosistema de paquetes libres y pagados que extienden las capacidades de R
- La mayoría de los nuevos métodos estadísticos se desarrollan primero en R,
- Para programar en R se pueden usar algunas IDEs (Interfaz de desarrollo) como **Eclipse**, **Emmacs** y **Rstudio** que es la más usada,
- Puedes probar las diferentes GUI disponibles como **Rcmdr** fuerte en investigación de mercado, **Rattle** útil para data mining o **rkward** en linux.



Algunas de las integraciones de R

Derivados de R

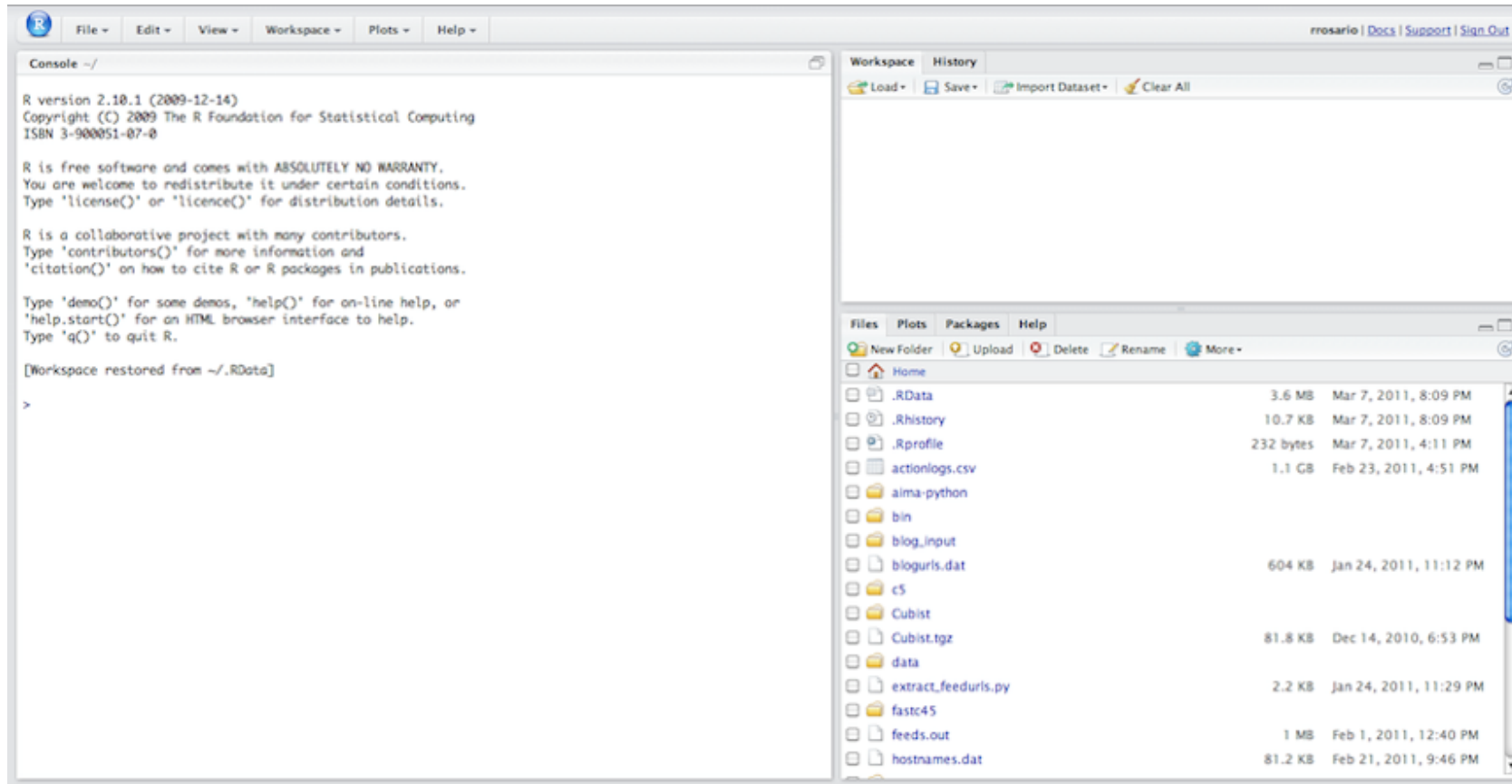
- **BIOCONDUCTOR** Herramientas para analizar Genoma con base R
- **ORACLE R ENTERPRISE** Integra R con Oracle Data Base
- **Microsoft R Application Network** Microsoft R Application Network
- **RAPPORTER** Reportería y análisis en la nube
- **RSTUDIO** Desarrollos como RStudio Server, Shiny server y paquetes en general
- **RSTUDIO Cloud** Ejecutar R online
- **Kernels de Kaggle** Kaggle tiene datasets, concursos y también kerneles que permiten ejecutar R en sus servidores
- **Google Colab** Permite ejecutar R online.



RStudio Cloud

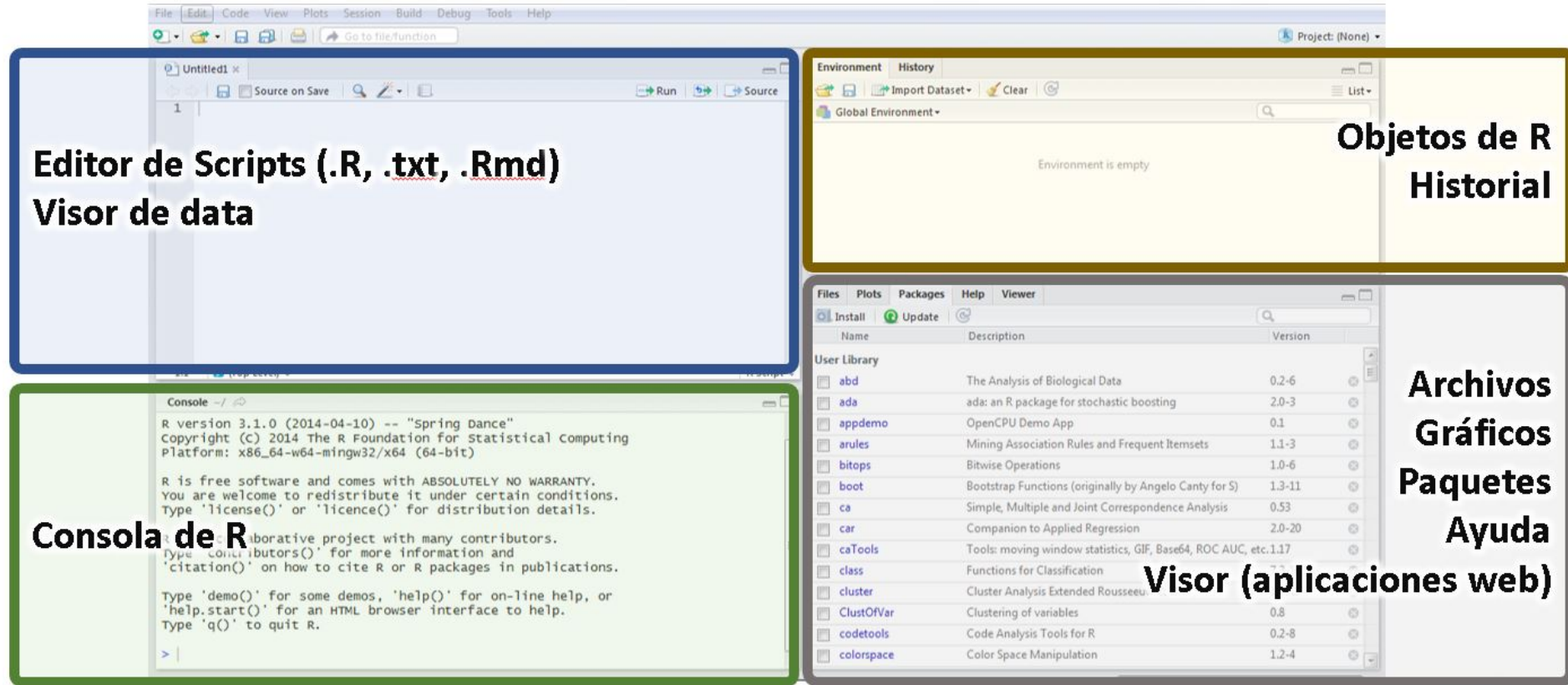
RStudio

Abrir RStudio



RStudio

Interfaz de RStudio se divide en 4 paneles:



The screenshot shows the RStudio interface with four panels highlighted and labeled:

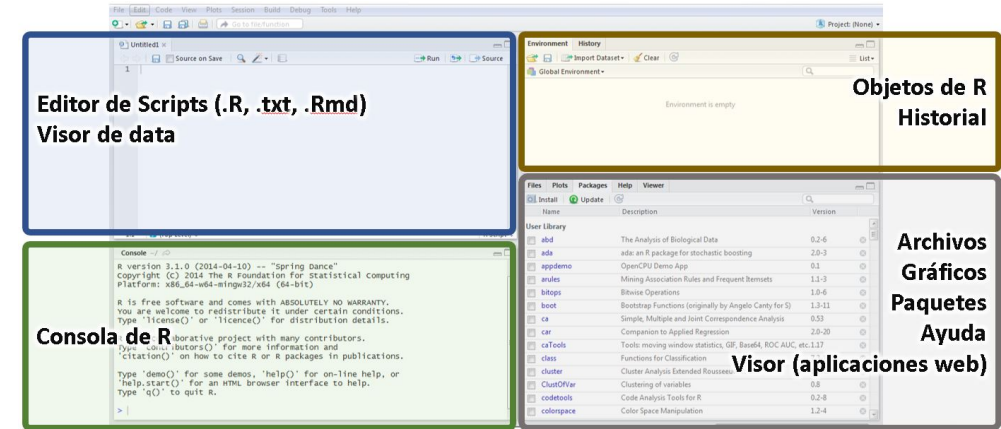
- Editor de Scripts (.R, .txt, .Rmd)**: The top-left panel, highlighted with a blue border, showing a script editor with a menu bar (File, Edit, Code, View, Plots, Session, Build, Debug, Tools, Help) and a toolbar. It contains a single line of code: `1`.
- Objetos de R Historial**: The top-right panel, highlighted with a yellow border, showing the Environment and History tabs. The Environment tab is active, displaying "Global Environment" and "Environment is empty".
- Archivos Gráficos Paquetes Ayuda**: The bottom-right panel, highlighted with a grey border, showing the Files, Plots, Packages, Help, and Viewer tabs. The Packages tab is active, displaying a list of installed packages with columns for Name, Description, and Version.
- Consola de R**: The bottom-left panel, highlighted with a green border, showing the R console output. It displays the R version (3.1.0), copyright information, and a welcome message.

Visor de data: This label is positioned over the top-left panel (Editor de Scripts).

Visor (aplicaciones web): This label is positioned over the bottom-right panel (Archivos Gráficos Paquetes Ayuda).

RStudio

1. Script: Pantalla donde se escriben las líneas de código
2. Consola: Donde se muestra el código ejecutado y el resultado
3. Environment/History: Pantalla donde se puede observar la data almacenada, los valores determinados.
4. File/plots/packages/help/viewer: en esta pantalla esta particionada en varias pestañas como:
 - Files.- Explorador de carpetas y archivos
 - Plots/Viewer.- Visor de gráficos o aplicaciones
 - Packages/Ayuda.- Muestra paquetes instalados del R y la ayuda de R
 - Help.- las ayudas internas del sistema



¿Por qué RStudio?

RStudio es la IDE más usada de R, a lo largo de este y otros cursos irán descubriendo características de este software, por ahora podemos numerar lo siguiente:

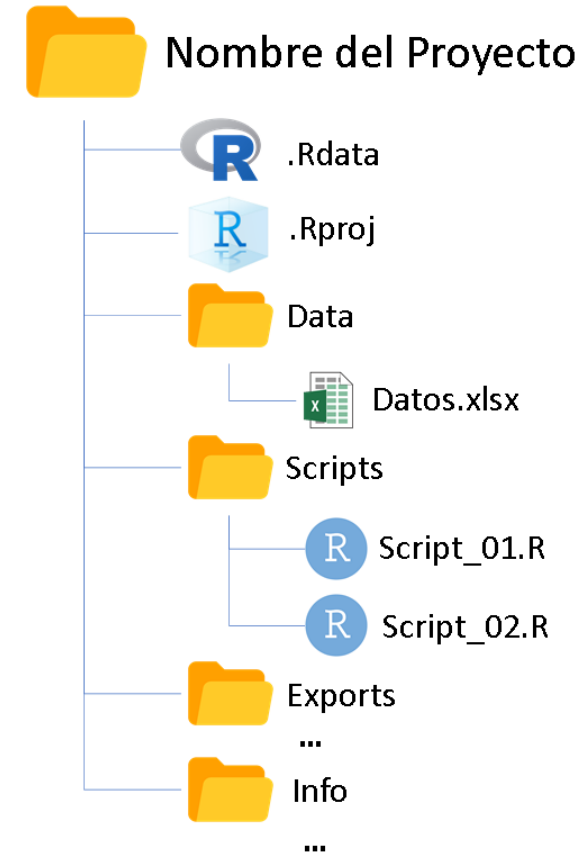
- Todo en 1-ventana: Console, Workspace, History, Working directory, Files, Plot, Packages y Help
- Integración de la consola de R
- Ejecutar código desde script
- Resaltado de sintaxis
- Completado de sintaxis
- Manejo de proyectos con soporte para Git y Subversion
- Herramientas para Investigación Reproducible (knitr)

¿IDE vs GUI?

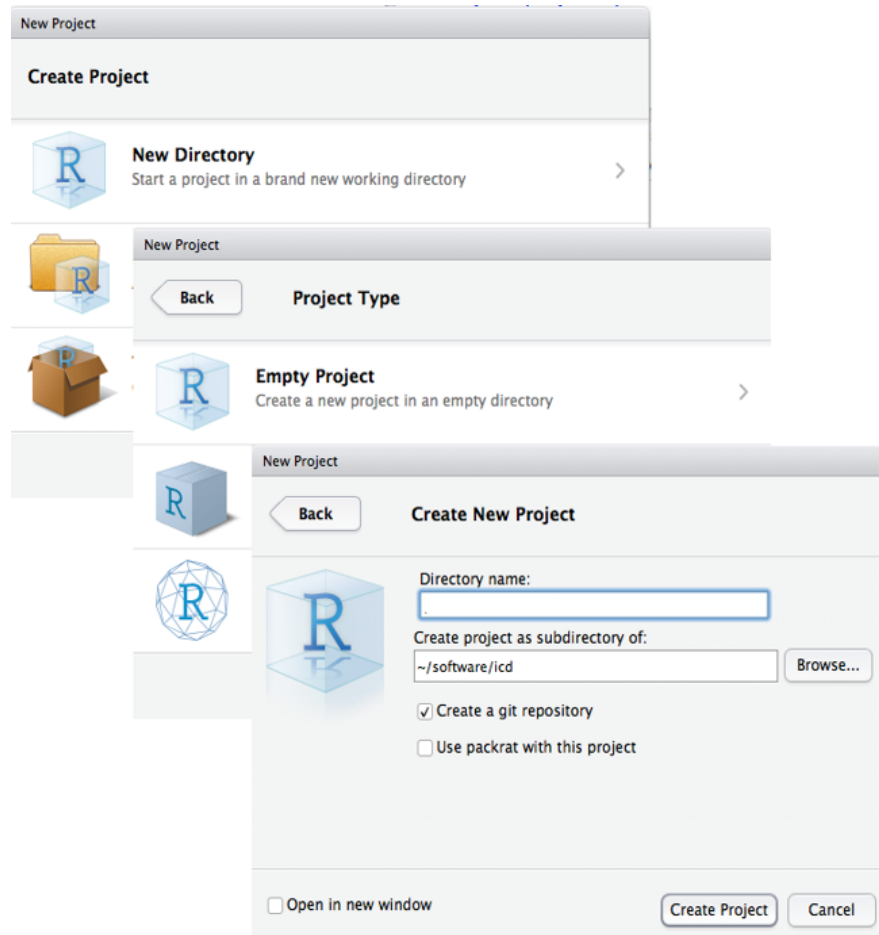
Una IDE es una interfaz que facilita la escritura de código, en nuestro caso facilita hacer los scripts que usaremos para nuestros análisis. Una GUI en cambio disminuye o elimina el uso de código, se realizan las operaciones con una serie de clicks en lugar de escribir código.

Proyectos en R, RStudio

- Un **Proyecto** es una carpeta que contiene todos los scripts, los archivos desde donde se importan los datos y los archivos de proyecto como el .RData (que contiene los objetos con los que se está trabajando) y .Rhistory (que contiene la historia de comandos ejecutados)
- Permite tener nuestros análisis ordenados,
- Al abrir un proyecto antiguo RStudio lo abre con las pestañas que se tenía activas,
- Permite colaboración utilizando GIT o Subversion,
- Se sugiere tener una estructura interior, por ejemplo:
 - Scripts, Data, Exports, Info



Iniciar un Proyecto en RStudio



- Crear un Nuevo Proyecto.-
Project >
..New Project >
...New Directory >
....Empty Project >
.....Poner nombre al Proyecto (se creará una carpeta con ese nombre) >
.....Create Project
- En la carpeta del proyecto crear las carpetas:
Data, Exports, Scripts, Info (estructura recomendada, no obligatoria)

Realizar un script en RStudio

- Nuevo script: `ctrl + shift + n`
- Completado de comando: `tab`, `ctrl + barra espaciadora`
- Ejecutar selección o línea actual: `ctrl + enter`
- Ir al source editor: `ctrl + 1`
- Ir a la consola: `ctrl + 2`
- Insertar símbolo de asignación `<-`: `alt + -`
- Comentar/des-comentar: `ctrl + shift + c`
- Reformatear línea: `ctrl + i`
- RStudio permite "plegar" código
- Crear secciones de código: `ctrl + shift + r o #### nombre ####`
- Saltar a (función o sección): `alt + shift + J`
- Ir a una función: `ctrl + .`

Generalidades

- Case sensitivity (Abc es diferente de abc)
- R, aparte de objetos, tiene:
 - Expresión.- Se evalúa, se imprime y el valor se pierde

```
5+5 # Expresión
```

```
## [1] 10
```

- Asignación.- Evalúa la expresión y guarda el resultado en una variable (no lo imprime)

```
a <- 5+5 # Asigna el valor a la variable "a"
```

Generalidades

- Comandos se separan por ; o enter
- Comandos pueden ser agrupados por { }
- Para comentar se usa #
- SAS y SPSS presentan extensos resultados, mientras que en R la salida es mínima (*En R un análisis se realiza mediante una serie de pasos, con resultados intermedios guardados en objetos*)

R como calculadora

```
2 + 3*5
```

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

```
log((1+2+3)/4) # log natural
```

```
## [1] 0.4054651
```

```
pi^2 # pi y potencia
```

```
## [1] 9.869604
```

R como calculadora

```
abs(-2) # valor absoluto
```

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

```
factorial(3) # factorial
```

```
## [1] 6
```

```
floor(5.7) # funcion piso
```

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

Generar secuencias, repeticiones y aleatorios

```
1:10 # secuencia de 1 a 10, de 1 en 1
```

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

```
seq(from= 0, to= 20, by= 5) # función seq
```

```
## [1] 0 5 10 15 20
```

```
seq(from= 5, by= 5, length.out= 5) # función seq
```

```
## [1] 5 10 15 20 25
```

Generar secuencias, repeticiones y aleatorios

```
rep(x= 3, times= 5) # repetir 5 veces el # 3
```

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

```
runif(n= 10, min= 1, max= 5) # Genera aleatorios uniformes
```

```
## [1] 2.388274 1.588528 2.497113 3.047155 1.991190 3.748902 1.976221 3.315052  
## [9] 1.356430 3.823720
```

```
rnorm(n= 10, mean= 100, sd= 10) # Genera aleatorios normales
```

```
## [1] 105.63652 113.50640 107.85083 101.35477 99.22578 92.47994 94.91715  
## [8] 106.32035 104.68566 94.94715
```

Asignaciones

- asigna el valor 5 a la variable a :
 - `a <- 5`
 - `5 -> a`
 - `assign("a", 5)`
- asigna globalmente el valor 5 a la variable a, (dentro de una función a seguirá valiendo 5)
 - `a <<- 5`
 - `5 ->> a`
- No se recomienda usar `a = 5`
- En RStudio verificar en la pestaña Environment la variable a

Asignaciones

- El resultado de una función de un objeto X puede ser asignada al mismo objeto X en la misma sentencia, es decir

```
a <- 5 # Expresión  
a
```

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

```
a <- 2*a  
a
```

```
## [1] 10
```


Workspace, environments y objetos

- Environment es donde se guardan las variables y objetos de R
- Tecnicamente es un conjunto de objetos y un puntero
- El environment por defecto es el workspace o `.GlobalEnv`
- Objetos en el workspace: `ls()` y `objects()`
- Para eliminar objetos: `rm`
- Obtener los objetos de un environment específico: `ls(envir= name_env)`
 - ejemplo: `ls(envir= globalenv())`
- El workspace se graba predeterminado con el nombre `.RData`
- RStudio permite el acceso desde su pestaña "Environment"

Workspace, environments y objetos

- Guardar workspace `save.image()`
- Guardar workspace bajo nombre/ruta definida `save.image(file= ruta.RData)`
- Guardar algunos objetos `save(lista_objetos, file = 'file_name.RData')`
- Guardar un único objeto `saveRDS(object, file = 'file_name.rds')`
- Cargar un archivo .RData `load(file = 'filename.Rdata')`
- Cargar un archivo .RData `obj <- readRDS(file = 'filename.rds')`

Historico de comandos (History)

- En la consola se puede acceder a los comandos anteriores con las flechas del teclado
- RStudio, abrir un histórico desplegable en la consola: `Ctrl + Up`
- Obtener el histórico: `history(max.show = 25)`
 - En la GUI oficial para Windows/Mac abrirá una nueva ventana, en linux se presentará en el mismo terminal
 - En RStudio `history()` nos lleva a la pestaña "History"
- Guardar el historico: `savehistory(file = '.Rhistory')`
- Cargar el historico desde un archivo: `loadhistory(file = '.Rhistory')`

Manejo de paquetes

- Instalación: `install.packages('nombre_paquete')`
- Ver paquetes instalados: `installed.packages()`
- Activar/Cargar: `library('nombre_paquete')`
- Desactivar/Des-cargar: `detach('package:nombre_paquete')`
- Paquetes cargados: `search()`
- RStudio tiene pestaña Packages que permite instalación visual

Paquetes a usar

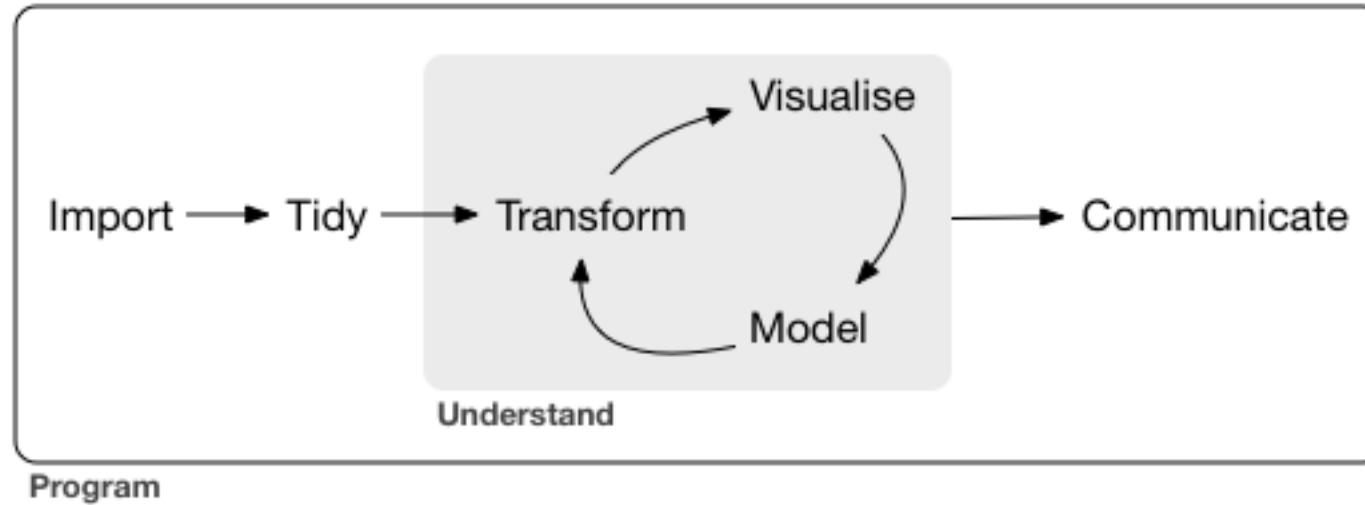
```
install.packages("openxlsx") # Para importar desde excel  
install.packages("tidyverse") # Manipulación de datos y ggplot2  
install.packages("magrittr") # pipe  
install.packages('modeest') # Obtener moda  
install.packages('fdth') # Calcular tabla frecuencias  
install.packages("prettyR") # Opcion de Descriptivas
```

Introducción a los análisis estadísticos

Curso: Manejo de datos y reportería con R

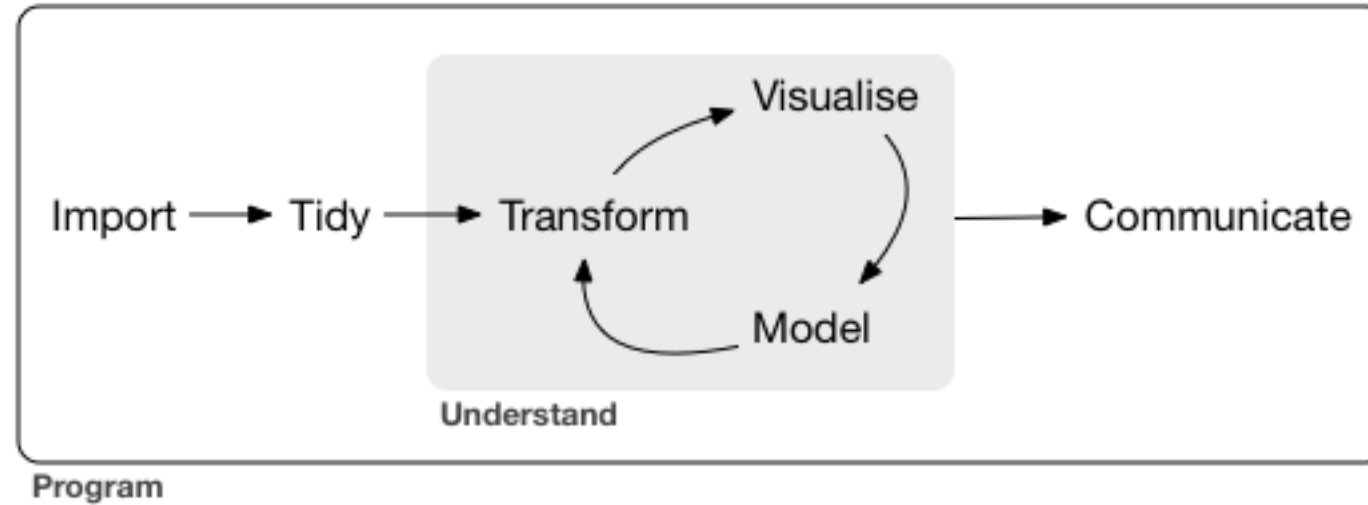
Néstor Montaña

Workflow de un análisis estadístico



- Import: Obtener y entender los datos
- Tidy: Ordenar los datos de tal manera que sea sencillo transformarlos, sumarlos, visualizarlos o realizar un modelo con ellos
- Transform: Manipular los datos hasta obtener el input que el análisis o técnica estadística necesita
- Visualise: Realizar el análisis exploratorio de datos
- Model: Aplicar técnicas estadísticas para el entendimiento del problema o tomar decisiones
- Communicate: Tratar de mostrar los resultados de tal forma que el resto del mundo los entienda, usando reportes, gráficos, visualizaciones interactivas, integración con herramientas de BI, web apps, etc.

Workflow de un análisis estadístico



- Import
- Tidy
- **Repetir mientras sea necesario**
- **Transform:** Manipular los datos, obtener el input del modelo
- **Visualise:** Realizar el análisis exploratorio de datos
- **Model:** Aplicar técnicas estadísticas
- Communicate

Ejemplo: Data de transacciones bancarias

El Banco del Pacífico requiere mejorar los tiempos de atención al cliente en ventanilla, para ello ha recolectado esta información anónimamente para cada cajero y transacción realizada.

Le suministran un excel con dos hojas:

1. Tiene los datos de las transacciones, columnas: Sucursal, Cajero, ID_Transaccion, Transaccion, Tiempo_Servicio_seg, Nivel de satisfacción, Monto de la transaccion.
2. Otra hoja que indica si en la sucursal se ha puesto o no el nuevo sistema.



Ejemplo: Data de transacciones bancarias

Crear un proyecto en RStudio, con las carpetas Data, Exports, etc

Poner en la carpeta Data, el excel y el .csv suministrado

Importar datos

Introducción al análisis de datos

Importar csv

- Desde RStudio (R-base o paquete readr)
Import Dataset > From Text File > Escoger archivo > Abrir > Escribir nombre a la variable > Import
- Con comando
`read.csv(file, sep = "," , dec = "." , stringsAsFactors= FALSE)`
- Para grandes volúmenes de datos usar paquete `data.table`
`fread()`
- Paquete `vroom` en desarrollo,
`vroom()`

Importar desde excel

- Copiando desde un archivo de excel abierto
`read.table("clipboard", sep="\t", header=TRUE)`
- Desde RStudio
Rstudio > Import Dataset > From Excel > Escoger archivo > Abrir > Escribir nombre a la variable > Import
- Usando el paquete `openxlsx`
`read.xlsx(xlsxFile , sheet , startRow , colNames , skipEmptyRows, rowNames)`
`data_tiempo_espera <- read.xlsx(xlsxFile = 'Data/Data_Banco.xlsx')`
- Otros paquetes
`excel.link`, `XLConnect`, `xlsx`, `readxl`, `rio`

Ejemplo - Importar

```
# Cargar la libreria a utilizar  
library(openxlsx)  
# Leer el archivo de excel y asignarlo al objeto data_banco  
data_banco <- read.xlsx(xlsxFile = "Data/Data_Banco.xlsx", sheet = "Data")  
data_sucursal <- read.xlsx(xlsxFile = "Data/Data_Banco.xlsx", sheet = "Data_Sucursal")
```

Exportar a excel

- Descargar [Rtools] (<https://cran.r-project.org/bin/windows/Rtools/>)
- Instalar Rtools
 - Se debe escoger "agregar al path"
 - Si la computadora ya tiene CYGWIN, se tiene un tratamiento especial
- Usando el paquete openxlsx

```
write.xlsx(x, file, asTable = FALSE, ...)
```
- Se puede usar los paquetes XLConnect, xlsx, etc.



Importar desde SPSS, SAS, Stata, etc

- Desde RStudio
Rstudio > Import Dataset > From SPSS/SAS/STATA
- Usando el paquete `foreign`
SAS: `read.xport()`
SPSS: `read.spss()` Stata: `read.dta()`
Soporta otros formatos
- Usando el paquete `haven`
SAS: `read_sas()` y `read_xpt()`
SPSS: `read_sav()` y `read_por()`
Stata: `read_dta()`
- Se puede usar el paquete `rio`

Exportar a SPSS, SAS, Stata, etc

- Usango el paquete foreign
`write.foreign(df, datafile, codefile, package = c("SPSS", "Stata", "SAS"), ...)`
- Usango el paquete haven
SAS: `write_sas()`
SPSS: `write_sav()`
Stata `write_dta()`
- Se puede usar el paquete rio

Interacción con Bases de Datos

- Utilizando ODBC RODB
(Recomendado para Microsoft SQL)
- Utilizando JDBC RJDBC
(Usa java DBC)
- Paquetes para bases específicas
RMySQL, ROracle, RPostgreSQL, RSQLite, mongolite, RMongo, MonetDB.R, rmongodb

Otros

- GIS sistemas de información geográfica con `rgdal` y `raster`
- GoogleSpreadSheets con `googlesheets`
- Archivos Open Document Spreadsheets con `readODS`
- JSON data con `rjson` o `jsonlite` o `RJSONIO`

Ejemplo - Importar

Bien, al importar hemos creado dos objetos en nuestro 'environment', pero ¿Qué son estos objetos? ¿Qué contienen?.

Entendamos un poco las estructuras de datos en R para poder explorar estos objetos

Estructuras de datos | Objetos

Introducción a R

Estructuras de datos | Objetos

R cuenta con un sinnúmero de estructuras de dato (clases de objetos), los más usados son:

- Vector
- Matriz
- Data.frame
- Litas
- Serie de Tiempo
- Data.table

Vectores

- En R no existen escalares, sino vectores de $\text{dim} = 1$

```
x <- 1  
is.vector(x)
```

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

- Los vectores se crean con la función `c()`

```
x <- c(11, 12, 13, 14) # crea x  
x # presenta x
```

```
## [1] 11 12 13 14
```

Vectores

- concatenar vectores

```
z <- c('a', 'b', 'c') # crea z
z # presenta z
```

```
## [1] "a" "b" "c"
```

```
y <- c(x, 21, 31, x) # crea y
y # presenta y
```

```
## [1] 11 12 13 14 21 31 11 12 13 14
```


Vectores

- Repetir vectores

```
rep(z, times=5) # repetir todo el vector 5 veces
```

```
## [1] "a" "b" "c" "a" "b" "c" "a" "b" "c" "a" "b" "c" "a" "b" "c"
```

```
rep(z, each=5) # repetir cada elemento 5 veces
```

```
## [1] "a" "a" "a" "a" "a" "b" "b" "b" "b" "b" "c" "c" "c" "c" "c"
```

Vectores

- Operaciones entre vectores

```
x # presenta x
```

```
## [1] 11 12 13 14
```

```
y <- c(10, 20, 30, 40) # Crea y  
x + 3*y - 1
```

```
## [1] 40 71 102 133
```

Data.frames

- Data.frame es una lista de vectores, cumple:
 - Las componentes son vectores
 - Cada vector puede ser de un tipo de dato distinto
 - Cada elemento, columna es una variable
 - Las columnas tienen el mismo largo
- Se podría decir que un data.frame es como una tabla en una hoja de excel

Data.frames

Crear un data.frame

```
Nombre <- c('Ana', 'Berni', 'Carlos')
Edad <- c(20,19,20)
Ciudad <- factor(c('Gye', 'Uio', 'Cue'))
df_1 <- data.frame(Nombre, Edad, Ciudad)
df_1
```

```
##   Nombre Edad Ciudad
## 1   Ana   20    Gye
## 2  Berni   19    Uio
## 3 Carlos   20    Cue
```

Data.frames

Crear un data.frame

```
df_2 <- data.frame( a= Nombre, b= Edad, c= Ciudad)  
df_2
```

```
##           a  b   c  
## 1      Ana 20 Gye  
## 2    Berni 19 Uio  
## 3   Carlos 20 Cue
```

Data.frames

Crear un data.frame

```
df_3 <- data.frame( Nombre= c('Ana', 'Berni', 'Carlos'),  
                     Edad = c(20,19,20),  
                     Ciudad= factor(c('Gye', 'Uio', 'Cue'))) )  
  
df_3
```

```
##   Nombre Edad Ciudad  
## 1    Ana   20    Gye  
## 2  Berni   19    Uio  
## 3 Carlos   20    Cue
```

Data.frames

Rownames

```
rownames(df_3) <- paste('id_',1:3,sep='')  
df_3
```

```
##      Nombre Edad Ciudad  
## id_1    Ana   20     Gye  
## id_2  Berni   19     Uio  
## id_3 Carlos   20     Cue
```

Data.frames

Modificar nombre de las variables

```
names(df_3) <- c('Name', 'Age', 'City')  
df_3
```

```
##      Name Age City  
## id_1   Ana  20  Gye  
## id_2 Berni  19  Uio  
## id_3 Carlos 20  Cue
```


Data.frames

Visualizar primeras filas

```
head(df_3, n=2)
```

```
##           Name Age City
## id_1     Ana  20  Gye
## id_2  Berni  19  Uio
```

Data.frames

Visualizar últimas filas

```
tail(df_3, n=2)
```

```
##           Name Age City  
## id_2   Berni  19  Uio  
## id_3 Carlos  20  Cue
```

Data.frames

Visualizar la estructura

`str` permite ver la estructura de cualquier objeto en R.

```
str(df_3)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ Name: Factor w/ 3 levels "Ana","Berni",...: 1 2 3
##  $ Age : num  20 19 20
##  $ City: Factor w/ 3 levels "Cue","Gye","Uio": 2 3 1
```

Listas

Imaginen un tren de carga donde en cada vagón tienen cosas diferentes, eso es una lista, un objeto que permite tener cualquier clase de objeto en sus posiciones (vagones).

```
list(1, c(2,3), df_1)
```

```
## [[1]]  
## [1] 1  
##  
## [[2]]  
## [1] 2 3  
##  
## [[3]]  
##   Nombre Edad Ciudad  
## 1    Ana   20    Gye  
## 2  Berni   19    Uio  
## 3 Carlos   20    Cue
```

Listas

Las posiciones pueden tener nombre y se pueden acceder con `lista$posicion`

```
lista_1 <- list(A= 1, B= c(2,3), C= df_1)
lista_1$B
```

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

```
lista_1$C
```

```
##   Nombre Edad Ciudad
## 1   Ana   20    Gye
## 2  Berni   19    Uio
## 3 Carlos   20    Cue
```

Estructuras de datos | Objetos

VECTOR

<i>names</i>			
[1]	[2]	[...]	[n]

SERIE DE TIEMPO

tiempo	$t1$	$t2$	tn	
data	[1]	[2]	[...]	[n]

MATRIZ

<i>rownames</i>	<i>colnames</i>			
	[1,1]	[1,2]	[...]	[1,m]
	[2,1]	[2,2]	[...]	[2,m]
	[...]	[...]	[...]	[...]
	[n,1]	[n,2]	[...]	[n,m]

DATAFRAME

		names		
rownames	Var_1	Var_2		Var_r
	[1,1]	[1,2]	[...]	[1,m]
	[2,1]	[2,2]	[...]	[2,m]
	[...]	[...]	[...]	[...]
	[n,1]	[n,2]	[...]	[n,m]

LISTA

[1,1] [1,2]		[1]	Var_1	Var_2		Var_m
[2,1] [2,2]		[2]	[1,1]	[1,2]	[...]	[1,m]
[3,1] [3,2]			[2,1]	[2,2]	[...]	[2,m]
			[...]	[...]	[...]	[...]

Ejemplo: Transacciones bancarias

Entonces, ¿qué tipo de estructura hemos importado?

R. Un data.frame

```
str(data_banco)
```

```
## 'data.frame':    24299 obs. of  7 variables:
## $ Sucursal      : num  62 62 62 62 62 62 62 62 62 62 ...
## $ Cajero        : num  4820 4820 4820 4820 4820 4820 4820 4820 4820 4820 ...
## $ ID_Transaccion : chr   "2" "2" "2" "2" ...
## $ Transaccion    : chr   "Cobro/Pago (Cta externa)" "Cobro/Pago (Cta externa)" "Cobro/Pago (Cta externa)" ...
## $ Tiempo_Servicio_seg: num  311 156 248 99 123 172 140 247 183 91 ...
## $ Satisfaccion   : chr   "Muy Bueno" "Malo" "Regular" "Regular" ...
## $ Monto          : chr   "2889,3" "1670,69" "3172,49" "1764.92" ...
```

Entender los datos

Introducción a los análisis estadísticos

Entender los datos

Luego de importar se debe entender los datos

- ¿Qué representa cada columna?
- ¿Qué tipo de dato debería tener cada columna?
- ¿Qué granularidad o atomicidad tiene la data?
- Si es que se tiene varios conjuntos de datos ¿Cómo se relacionan los datos?
- A qué periodo de tiempo corresponde la data
- Muchas veces se obtiene la información desde una base de datos y por tanto toca entender la base y el query que genera los datos

Ejemplo - Entender los datos

Podríamos ver las primeras filas

```
# ver las primeras 5 filas  
head(data_sucursal, n = 5)
```

##	ID_Sucursal	Sucursal	Nuevo_Sistema
## 1	62	Riocentro Sur	No
## 2	85	Centro	Si
## 3	267	Alborada	Si
## 4	443	Mall del Sol	Si
## 5	586	Via Daule	No

Ejemplo - Entender los datos

Listar los nombres de las columnas

```
# Listar los nombres de las columnas  
names(data_banco)
```

```
## [1] "Sucursal"          "Cajero"             "ID_Transaccion"  
## [4] "Transaccion"       "Tiempo_Servicio_seg" "Satisfaccion"  
## [7] "Monto"
```

```
names(data_sucursal)
```

```
## [1] "ID_Sucursal"  "Sucursal"      "Nuevo_Sistema"
```

Tipos de datos

Tipos datos en R

```
1 # Entero
```

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

```
3.5 # Numérico
```

```
## [1] 3.5
```

```
im <- 3.5 - 8i # Complejo  
Im(im) # Parte imaginaria
```

```
## [1] -8
```

```
Re(im) # Parte real
```

```
## [1] 3.5
```

Tipos de datos

```
'a' # Caracter
```

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

```
fecha <- lubridate::ymd("2010-01-01") # Fecha  
data_serie <- ts(1:24, start = 2014) # Serie de tiempo  
data_serie
```

```
## Time Series:  
## Start = 2014  
## End = 2037  
## Frequency = 1  
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

Tipos de datos - factor

- Util para tipos de datos ordinales
- `x`: es el vector de información
- `levels`: los niveles del factor labels: nombre de los niveles
- El factor puede tener un orden específico

```
# Crear un factor ordenado  
data_factor <- factor(x= c('alto', 'bajo', 'alto', 'alto'),  
                      levels = c('bajo', 'mediano', 'alto'), ordered = T)  
data_factor # Mostrar el factor
```

```
## [1] alto bajo alto alto  
## Levels: bajo < mediano < alto
```

Tipos de datos

Datos lógicos y valores perdidos

```
TRUE # LOGICO
```

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

```
FALSE # LOGICO
```

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

```
NA # No disponible, perdido
```

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

Nota: Los NA requieren un tratamiento especial

Tipos de datos

Casos especiales

```
1/0 # Infinito
```

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

```
-1/0 # Infinito negativo
```

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

```
Inf/Inf # No un Numero
```

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


Tipos de datos

Qué creen que resulte de lo siguiente?

```
(0:3)^Inf
```

Tipos de datos

Qué creen que resulte de lo siguiente?

```
0:3 # es una secuencia
```

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

```
(0:3)^Inf # ahora se eleva cada elemnto a Inf
```

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

Tipos de variables

Tipos de variables y su correspondencia en R

Tipos de datos

Al importar los datos es posible que no estén en el formato correcto, podemos pasar de un tipo de dato a otro (siempre que sea posible), usando R-base

```
as.numeric(x)
```

```
## [1] 11 12 13 14
```

```
as.integer(im)
```

```
## Warning: partes imaginarias descartadas en la coerción
```

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

```
as.character(data_factor)
```

```
## [1] "alto" "bajo" "alto" "alto"
```

```
as.factor(x)
```

```
## [1] 11 12 13 14
```

```
## Levels: 11 12 13 14
```

Tipos de datos

También se puede usar el paquete *readr* del *tidyverse* (estas funciones son las que se usan automáticamente al importar desde el menú de RStudio, *ojo*: Parten de vectores de tipo character).

Para datos numéricos tenemos varias funciones, `parse_number` es la menos restrictiva (elimina lo que no sea número y convierte), `parse_integer` y `parse_double` esperan valores en el formato indicado

```
library(tidyverse)
parse_number(c("$1,234.5", "$12.45"))
```

```
## [1] 1234.50 12.45
```

```
parse_double(c("1234.5", "12.45"))
```

```
## [1] 1234.50 12.45
```

```
parse_integer(c("$1,234.5", "$12.45")) # Error
```

```
## [1] NA NA
## attr(,"problems")
## # A tibble: 2 x 4
##   row   col expected actual
##   <int> <int> <chr>    <chr>
```

Tipos de datos

Para datos de tipo character

```
as.character(data_factor)
```

```
## [1] "alto" "bajo" "alto" "alto"
```

```
as.character(lubridate::now())
```

```
## [1] "2021-01-23 01:07:16"
```

```
## parse_character(data_factor) #ERROR
```

Tipos de datos

Para datos de tipo factor se requiere especificar los niveles, y si un dato no está en los niveles se muestra un warning

```
factor(c("Alto", "Bajo", "Medio", "Alto"),  
       levels = c("Alto", "Medio", "Bajo"), ordered = T)
```

```
## [1] Alto Bajo Medio Alto  
## Levels: Alto < Medio < Bajo
```

```
factor(c("Alto", "Bajo", "Medio bajo", "Alto"),  
       levels = c("Alto", "Medio", "Bajo")) #ERROR
```

```
## [1] Alto Bajo <NA> Alto  
## Levels: Alto Medio Bajo
```

Tipos de datos

Para datos de tipo factor se requiere especificar los niveles, y si un dato no está en los niveles se muestra un warning

```
parse_factor(c("Alto", "Bajo", "Medio", "Alto"),  
             levels = c("Alto", "Medio", "Bajo"), ordered = T)
```

```
## [1] Alto Bajo Medio Alto  
## Levels: Alto < Medio < Bajo
```

```
parse_factor(c("Alto", "Bajo", "Medio bajo", "Alto"),  
             levels = c("Alto", "Medio", "Bajo")) #ERROR
```

```
## Warning: 1 parsing failure.  
## row col          expected      actual  
##   3  -- value in level set Medio bajo  
  
## [1] Alto Bajo <NA> Alto  
## attr(,"problems")  
## # A tibble: 1 x 4  
##   row col expected      actual  
##   <int> <int> <chr>      <chr>  
## 1     3 NA value in level set Medio bajo
```


Tipos de datos

Para datos de tipo fecha tenemos:

```
parse_date("2010-10-01")
```

```
## [1] "2010-10-01"
```

```
parse_date("2010/10/01")
```

```
## [1] "2010-10-01"
```

Sin embargo los tipos de dato fecha tienen muchas características que veremos más adelante.

Entender los datos

Luego de importar se debe entender los datos

- ¿Qué representa cada columna?
- ¿Qué tipo de dato debería tener cada columna?
- ¿Qué granularidad o atomicidad tiene la data?
- Si es que se tiene varios conjuntos de datos ¿Cómo se relacionan los datos?
- A qué periodo de tiempo corresponde la data
- Muchas veces se obtiene la información desde una base de datos y por tanto toca entender la base y el query que genera los datos

Entender los datos - Ejemplo

¿Están bien nuestros tipos de datos?

...

```
# Ver la estructura del data.frame  
str(data_banco)
```

```
## 'data.frame':    24299 obs. of  7 variables:  
##  $ Sucursal      : num  62 62 62 62 62 62 62 62 62 62 ...  
##  $ Cajero        : num  4820 4820 4820 4820 4820 4820 4820 4820 4820 4820 ...  
##  $ ID_Transaccion : chr  "2" "2" "2" "2" ...  
##  $ Transaccion   : chr  "Cobro/Pago (Cta externa)" "Cobro/Pago (Cta externa)" "Cobro/Pago (Cta  
##  $ Tiempo_Servicio_seg: num  311 156 248 99 123 172 140 247 183 91 ...  
##  $ Satisfaccion   : chr  "Muy Bueno" "Malo" "Regular" "Regular" ...  
##  $ Monto         : chr  "2889,3" "1670,69" "3172,49" "1764.92" ...
```

Entender los datos - Ejemplo

¿Está bien nuestros tipos de datos?

Si no lo están entonces debemos transformarlos, para esto aprenderemos sobre manipulación de datos.

```
# Ver la estructura del data.frame  
str(data_banco)
```

```
## 'data.frame':    24299 obs. of  7 variables:  
## $ Sucursal      : num  62 62 62 62 62 62 62 62 62 62 ...  
## $ Cajero        : num  4820 4820 4820 4820 4820 4820 4820 4820 4820 4820 ...  
## $ ID_Transaccion : chr   "2" "2" "2" "2" ...  
## $ Transaccion   : chr   "Cobro/Pago (Cta externa)" "Cobro/Pago (Cta externa)" "Cobro/Pago (Cta  
## $ Tiempo_Servicio_seg: num  311 156 248 99 123 172 140 247 183 91 ...  
## $ Satisfaccion   : chr   "Muy Bueno" "Malo" "Regular" "Regular" ...  
## $ Monto          : chr   "2889,3" "1670,69" "3172,49" "1764.92" ...
```

Manipulacion de datos - Basico

Introducción a R

Manipulación de datos

R tiene sus comandos predeterminados para manipular datos, esto se conoce como *R Base*, sin embargo existen varios paquetes que simplifican esta tarea, en este curso veremos como hacerlo con el paquete *dplyr* (y *magrittr*) que están dentro del conjunto de paquetes llamado **tidyverse**

```
# Cargar la librería  
library(tidyverse)
```

Tibbles (un dataframe mejorado):

Tibble es un objeto del paquete dplyr, entre las mejoras que da es que no imprime todo el objeto en pantalla, sino un resumen del mismo. (más información tipeando ?tibble)

```
# Convertir el data_banco a un tibble
data_banco <- as_tibble( data_banco)
# Muestra data_banco
data_banco
```

```
## # A tibble: 24,299 x 7
##   Sucursal Cajero ID_Transaccion Transaccion Tiempo_Servicio~ Satisfaccion
##   <dbl>   <dbl> <chr>          <chr>          <dbl> <chr>
## 1      62    4820 2              Cobro/Pago~      311 Muy Bueno
## 2      62    4820 2              Cobro/Pago~      156 Malo
## 3      62    4820 2              Cobro/Pago~      248 Regular
## 4      62    4820 2              Cobro/Pago~       99 Regular
## 5      62    4820 2              Cobro/Pago~      123 Muy Bueno
## 6      62    4820 2              Cobro/Pago~      172 Bueno
## 7      62    4820 2              Cobro/Pago~      140 Regular
## 8      62    4820 2              Cobro/Pago~      247 Bueno
## 9      62    4820 2              Cobro/Pago~      183 Muy Bueno
## 10     62    4820 2              Cobro/Pago~       91 Muy Bueno
## # ... with 24,289 more rows, and 1 more variable: Monto <chr>
```

Seleccionar columnas: select()

Seleccionar las columnas Transaccion, Tiempo_Servicio_seg del data.frame data_banco

```
# Seleccionar las columnas Transaccion, Tiempo_Servicio_seg del data.frame data_banco  
# Note que como no se asignó, R evalúa la expresión y presenta el resultado  
select( data_banco, Transaccion, Tiempo_Servicio_seg)
```

```
## # A tibble: 24,299 x 2  
##   Transaccion      Tiempo_Servicio_seg  
##   <chr>          <dbl>  
## 1 Cobro/Pago (Cta externa)      311  
## 2 Cobro/Pago (Cta externa)      156  
## 3 Cobro/Pago (Cta externa)      248  
## 4 Cobro/Pago (Cta externa)       99  
## 5 Cobro/Pago (Cta externa)      123  
## 6 Cobro/Pago (Cta externa)      172  
## 7 Cobro/Pago (Cta externa)      140  
## 8 Cobro/Pago (Cta externa)      247  
## 9 Cobro/Pago (Cta externa)      183  
## 10 Cobro/Pago (Cta externa)       91  
## # ... with 24,289 more rows
```


Operador Pipe: %>%

El operador Pipe %>% del paquete magrittr (y del tidyverse) permiten que el código sea más legible porque:

- Permite secuencias estructurantes de operaciones de datos de izquierda a derecha (a diferencia de dentro y fuera),

```
# Con el operador pipe  
data_banco %>% names
```

```
## [1] "Sucursal"           "Cajero"              "ID_Transaccion"  
## [4] "Transaccion"        "Tiempo_Servicio_seg" "Satisfaccion"  
## [7] "Monto"
```

```
# Sin el operador pipe  
names(data_banco)
```

```
## [1] "Sucursal"           "Cajero"              "ID_Transaccion"  
## [4] "Transaccion"        "Tiempo_Servicio_seg" "Satisfaccion"  
## [7] "Monto"
```

Operador Pipe: %>%

El operador Pipe %>% del paquete magrittr (y del tidyverse) permiten que el código sea más legible porque:

- Permite secuencias estructurantes de operaciones de datos de izquierda a derecha (a diferencia de dentro y fuera),
- Evita llamadas a funciones anidadas,

```
# Con Pipe  
data_banco %>% names %>% length
```

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

```
# Sin Pipe  
length(names(data_banco))
```

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

Operador Pipe: %>%

El operador Pipe %>% reemplaza el primer argumento del comando siguiente, es decir $x \%>\% f(y)$ es equivalente a $f(x, y)$.

```
head(data_banco, n= 3) # Sin Pipe
```

```
## # A tibble: 3 x 7
##   Sucursal Cajero ID_Transaccion Transaccion Tiempo_Servicio~ Satisfaccion
##   <dbl> <dbl> <chr>          <chr>          <dbl> <chr>
## 1      62   4820 2          Cobro/Pago~      311 Muy Bueno
## 2      62   4820 2          Cobro/Pago~      156 Malo
## 3      62   4820 2          Cobro/Pago~      248 Regular
## # ... with 1 more variable: Monto <chr>
```

```
data_banco %>% head(n= 3) # Con Pipe
```

```
## # A tibble: 3 x 7
##   Sucursal Cajero ID_Transaccion Transaccion Tiempo_Servicio~ Satisfaccion
##   <dbl> <dbl> <chr>          <chr>          <dbl> <chr>
## 1      62   4820 2          Cobro/Pago~      311 Muy Bueno
## 2      62   4820 2          Cobro/Pago~      156 Malo
## 3      62   4820 2          Cobro/Pago~      248 Regular
## # ... with 1 more variable: Monto <chr>
```

Luego retornaremos a ver más beneficios de este operador.

Seleccionar columnas: select()

Seleccionar las columnas Transaccion, Tiempo_Servicio_seg del data.frame data_banco pero usando %>%, lo que permite programar como si se escribiese "del data_banco, selecciona las columnas Transaccion y Tiempo_Servicio_seg"

```
# Seleccionar las columnas Transaccion, Tiempo_Servicio_seg del data.frame data_banco
# Note que como no se asignó, R evalúa la expresión y presenta el resultado
# Se lee, del data_banco, selecciona las columnas Transaccion y Tiempo_Servicio_seg
# data_banco[ , c("Transaccion", "Tiempo_Servicio_seg") ] ## Base de R
data_banco %>% select( Transaccion, Tiempo_Servicio_seg)
```

```
## # A tibble: 24,299 x 2
##   Transaccion      Tiempo_Servicio_seg
##   <chr>          <dbl>
## 1 Cobro/Pago (Cta externa)      311
## 2 Cobro/Pago (Cta externa)      156
## 3 Cobro/Pago (Cta externa)      248
## 4 Cobro/Pago (Cta externa)       99
## 5 Cobro/Pago (Cta externa)      123
## 6 Cobro/Pago (Cta externa)      172
## 7 Cobro/Pago (Cta externa)      140
## 8 Cobro/Pago (Cta externa)      247
## 9 Cobro/Pago (Cta externa)      183
```

Seleccionar columnas: select()

Seleccionar y ver en el visor de datos de RStudio

```
# Seleccionar y ver en el visor de datos de RStudio  
data_banco %>% select( Transaccion, Tiempo_Servicio_seg) %>% View
```

Seleccionar columnas: select()

Seleccionar todas las columnas menos Cajero

```
# Seleccionar todas las columnas menos Cajero  
data_banco %>% select( -Cajero) %>% View
```

Seleccionar columnas: select()

Seleccionar según nombre de la columna/variable.

```
# Seleccionar todas las columnas cuyo nombre contenga el texto "Tra"  
data_banco %>% select( contains("Tra")) %>% View  
# Seleccionar todas las columnas cuyo nombre inicie con "S"  
data_banco %>% select( starts_with("S")) %>% View  
# Seleccionar todas las columnas cuyo nombre finalice con "on"  
data_banco %>% select( ends_with("on")) %>% View  
# Seleccionar todas las columnas cuyo nombre contenga una "r" o un "sa"  
data_banco %>% select( matches("r?sa")) %>% View  
# Más información sobre expresiones regulares usando: ?base::regex
```

Filtrar filas por posición: slice()

Para filtrar las filas según su posición o índice se usa `slice`. Existe también `slice_sample` para seleccionar una muestra, `slice_min` y `slice_max` para seleccionar el mínimo y máximo valor de una variable respectivamente.

```
# Seleccionar las filas 3, 4 y 5 de nuestro dataframe
data_banco %>% slice( 3:5)
```

```
## # A tibble: 3 x 7
##   Sucursal Cajero ID_Transaccion Transaccion Tiempo_Servicio~ Satisfaccion
##   <dbl> <dbl> <chr> <chr> <dbl> <chr>
## 1      62  4820 2 Cobro/Pago~ 248 Regular
## 2      62  4820 2 Cobro/Pago~ 99 Regular
## 3      62  4820 2 Cobro/Pago~ 123 Muy Bueno
## # ... with 1 more variable: Monto <chr>
```

```
# Seleccionar las filas con los mayores Tiempos
data_banco %>% slice_max(Tiempo_Servicio_seg, n= 2)
```

```
## # A tibble: 2 x 7
##   Sucursal Cajero ID_Transaccion Transaccion Tiempo_Servicio~ Satisfaccion
##   <dbl> <dbl> <chr> <chr> <dbl> <chr>
## 1      85  3983 2 Cobro/Pago~ 1603. Muy Bueno
## 2      85  3678 2 Cobro/Pago~ 1337. Malo
## # ... with 1 more variable: Monto <chr>
```




Filtrar/Seleccionar filas: `filter()`

Podemos filtrar las **filas que cumplan las condiciones dadas** usando el comando `filter()`, pero para esto debemos antes entender los **operadores de relación y lógicos en R**.

Operadores de relación

```
3 == 4 # Igualdad
```

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

```
3 != 4 # Desigualdad
```

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

```
3 > 4 # Mayor que
```

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

```
3 <= 4 # Menor igual que
```

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

Operadores lógicos

```
! FALSE # No
```

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

```
TRUE & FALSE # Y
```

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

```
TRUE | FALSE # O
```

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

```
xor(TRUE,TRUE) # Ó excluyente
```

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

```
TRUE & NA # Cuidado especial con los NA
```

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

Operadores lógicos

```
xor(TRUE,TRUE) # Ó excluyente
```

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

```
TRUE & NA # Cuidado especial con los NA
```

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

Filtrar/Seleccionar filas: filter()

Filtrar las filas según las condiciones dadas en filter()

```
# Filtrar las filas correspondientes a la sucursal 62
data_banco %>% filter( Sucursal== 62 ) %>% View
# Filtrar las filas correspondientes a la sucursal 62 y hayan durado más de 120 seg
data_banco %>% filter( Sucursal== 62 & Tiempo_Servicio_seg > 120 ) %>% View
# Filtrar las filas correspondientes a la sucursal 62, hayan durado
# más de 120 segundos y la evaluación a la satisfacción sea Bueno
data_banco %>% filter( Sucursal== 62 & Tiempo_Servicio_seg > 120 &
                      Satisfaccion== "Muy Bueno") %>% View
```

Operador Pipe: %>%

Regresamos al operador Pipe %>%, recordemos que lo encontramos en el paquete magrittr y tidyverse, y su principal utilidad es que permite que el código sea más legible porque:

- Permite secuencias estructurantes de operaciones de datos de izquierda a derecha (a diferencia de dentro y fuera),
- Evitando llamadas a funciones anidadas,
- Minimiza la necesidad de variables locales y definiciones de funciones
- Facilita agregar pasos en cualquier lugar de la programación

Operador Pipe: %>%

Veamos cómo resolveríamos esto con pipe y sin pipe, *Filtrar las filas correspondientes a la sucursal 62, hayan durado más de 120 segundos y la evaluación a la satisfacción sea Bueno*

```
## Sin Pipe
View( select( filter(data_banco, Sucursal== 62 & Tiempo_Servicio_seg > 120 &
                    Satisfaccion== "Muy Bueno") ,
              Transaccion, Tiempo_Servicio_seg, Satisfaccion ) )

## Con Pipe
data_banco %>%
  filter( Sucursal== 62 & Tiempo_Servicio_seg > 120 &
          Satisfaccion== "Muy Bueno") %>%
  select( Transaccion, Tiempo_Servicio_seg, Satisfaccion ) %>%
  View
```

Operador Pipe: %>%

Hay varios tipos de Pipe, %>%, %<>% %\$%; sólo el %>% viene en el tidyverse, el resto están en el paquete magrittr. Este operador funciona así:

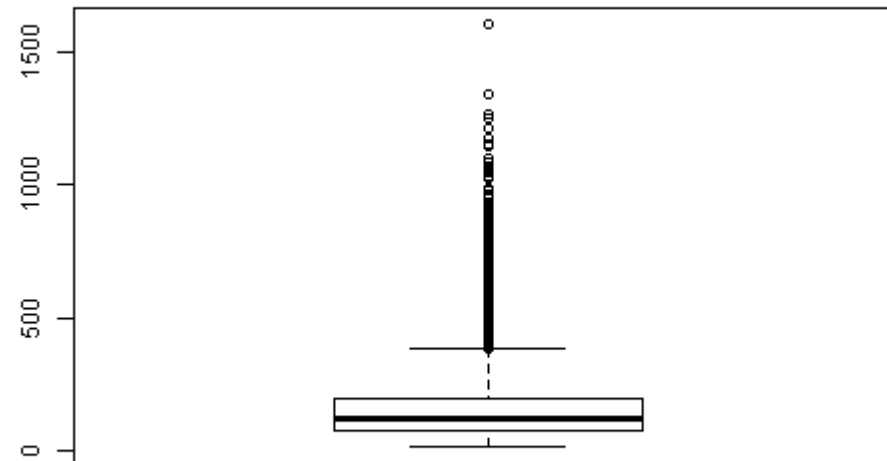
- `x %>% f` es equivalente a `f(x)`
- `x %>% f(y)` es equivalente a `f(x, y)`
- `x %>% f %>% g %>% h` es equivalente a `h(g(f(x)))`
- `x %>% f(y = nrow(.), z = ncol(.))` es equivalente a `f(x, y = nrow(x), z = ncol(x))`
- `x %>% {f(y = nrow(.), z = ncol(.))}` es equivalente a `f(y = nrow(x), z = ncol(x))`
- `x %<>% f %>% g` es equivalente a `x <- g(f(x))`
- `%$%` permite seleccionar columnas `x %$% f(col1, col2)` es equivalente a `f(x$col1, x$col2)`

Veremos algunos ejemplos para comprender mejor

Seleccionar columnas y aplicar una función

Seleccionar la columna Tiempo_Servicio_seg y obtener un boxplot

```
# Seleccionar la columna Tiempo_Servicio_seg y obtener un boxplot  
# boxplot(data_banco$Tiempo_Servicio_seg) ## Sin Pipe  
data_banco %>% select(Tiempo_Servicio_seg) %>% boxplot
```



Aplicar una función a una columna

Seleccionar la columna Tiempo_Servicio_seg y obtener los fivenumbers de Tukey

```
library(magrittr) #Porque ahora vamos a usar %$%  
# Del databanco obtener los fivenumbers de Tukey de la columna Tiempo_Servicio_seg  
data_banco %$% fivenum(Tiempo_Servicio_seg, na.rm= TRUE)
```

```
## [1] 18.13177 75.69119 122.45229 197.73046 1602.69832
```

Filtrar filas y seleccionar

```
# Con el data banco  
# Filtrar las filas correspondientes a la sucursal 85  
# calcular la correlacion entre Tiempo_Servicio_seg y Monto  
data_banco %>% # Operador pipe total  
  filter( Sucursal== 85 ) %$% # Operador pipe para seleccion de columnas  
  cor(Tiempo_Servicio_seg, as.numeric(Monto))
```

```
## [1] 0.5339392
```

Ordenar las filas: arrange()

Ordenar las filas según lo expresado en arrange()

```
# Ordenar por la satisfaccion  
data_banco %>% arrange( Satisfaccion ) %>% View  
# Ordenar cada Transaccion y dentro de cada transaccion  
# de mayor a menor por tiempo de servicio  
data_banco %>% arrange( Transaccion, desc(Tiempo_Servicio_seg) ) %>% View
```

Crear o modificar columnas/variables mutate()

Crear una nueva columna con el tiempo en minutos

```
# Crear una nueva columna con el tiempo en minutos
data_banco %>% mutate(Tiempo_Servicio_Min= Tiempo_Servicio_seg/60)
```

```
## # A tibble: 24,299 x 8
##   Sucursal Cajero ID_Transaccion Transaccion Tiempo_Servicio~ Satisfaccion
##   <dbl> <dbl> <chr> <chr> <dbl> <chr>
## 1      62    4820 2 Cobro/Pago~ 311 Muy Bueno
## 2      62    4820 2 Cobro/Pago~ 156 Malo
## 3      62    4820 2 Cobro/Pago~ 248 Regular
## 4      62    4820 2 Cobro/Pago~ 99 Regular
## 5      62    4820 2 Cobro/Pago~ 123 Muy Bueno
## 6      62    4820 2 Cobro/Pago~ 172 Bueno
## 7      62    4820 2 Cobro/Pago~ 140 Regular
## 8      62    4820 2 Cobro/Pago~ 247 Bueno
## 9      62    4820 2 Cobro/Pago~ 183 Muy Bueno
## 10     62    4820 2 Cobro/Pago~ 91 Muy Bueno
## # ... with 24,289 more rows, and 2 more variables: Monto <chr>,
## #   Tiempo_Servicio_Min <dbl>
```

Crear o modificar columnas/variables mutate()

Crear una nueva columna con el tiempo en minutos

```
# Crear una nueva columna con el tiempo en minutos
data_banco %>% mutate(Tiempo_Servicio_Min= Tiempo_Servicio_seg/60)
```

```
## # A tibble: 24,299 x 8
##   Sucursal Cajero ID_Transaccion Transaccion Tiempo_Servicio~ Satisfaccion
##   <dbl> <dbl> <chr> <chr> <dbl> <chr>
## 1      62   4820 2 Cobro/Pago~ 311 Muy Bueno
## 2      62   4820 2 Cobro/Pago~ 156 Malo
## 3      62   4820 2 Cobro/Pago~ 248 Regular
## 4      62   4820 2 Cobro/Pago~ 99 Regular
## 5      62   4820 2 Cobro/Pago~ 123 Muy Bueno
## 6      62   4820 2 Cobro/Pago~ 172 Bueno
## 7      62   4820 2 Cobro/Pago~ 140 Regular
## 8      62   4820 2 Cobro/Pago~ 247 Bueno
## 9      62   4820 2 Cobro/Pago~ 183 Muy Bueno
## 10     62   4820 2 Cobro/Pago~ 91 Muy Bueno
## # ... with 24,289 more rows, and 2 more variables: Monto <chr>,
## #   Tiempo_Servicio_Min <dbl>
```

Crear o modificar columnas/variables mutate()

Crear una nueva columna con el tiempo en minutos

```
# Crear una nueva columna con el tiempo en minutos  
data_banco %>% mutate(Tiempo_Servicio_Min= Tiempo_Servicio_seg/60)
```

Nótese que **no se asignó**, el objeto data_banco no tiene la columna Tiempo_Servicio_Min

Crear o modificar columnas/variables mutate()

Crear una nueva columna con el tiempo en minutos

```
# Crear una nueva columna con el tiempo en minutos
data_banco <- data_banco %>%
  mutate(Tiempo_Servicio_Min= Tiempo_Servicio_seg/60)
# Mostrar
data_banco
```

```
## # A tibble: 24,299 x 8
##   Sucursal Cajero ID_Transaccion Transaccion Tiempo_Servicio~ Satisfaccion
##   <dbl>   <dbl> <chr>          <chr>          <dbl> <chr>
## 1      62    4820 2              Cobro/Pago~      311 Muy Bueno
## 2      62    4820 2              Cobro/Pago~      156 Malo
## 3      62    4820 2              Cobro/Pago~      248 Regular
## 4      62    4820 2              Cobro/Pago~       99 Regular
## 5      62    4820 2              Cobro/Pago~      123 Muy Bueno
## 6      62    4820 2              Cobro/Pago~      172 Bueno
## 7      62    4820 2              Cobro/Pago~      140 Regular
## 8      62    4820 2              Cobro/Pago~      247 Bueno
## 9      62    4820 2              Cobro/Pago~      183 Muy Bueno
## 10     62    4820 2              Cobro/Pago~       91 Muy Bueno
## # ... with 24,289 more rows, and 2 more variables: Monto <chr>,
```


Nuevas columnas transmute()

Para conservar sólo las nuevas columnas se usa transmute()

```
# Crear una nueva columna con el tiempo en minutos
data_banco %>%
  transmute(Tiempo_Servicio_Min= Tiempo_Servicio_seg/60)
```

```
## # A tibble: 24,299 x 1
##   Tiempo_Servicio_Min
##   <dbl>
## 1      5.18
## 2      2.6
## 3      4.13
## 4      1.65
## 5      2.05
## 6      2.87
## 7      2.33
## 8      4.12
## 9      3.05
## 10     1.52
## # ... with 24,289 more rows
```

Entender los datos - Ejemplo

¿Está bien nuestros tipos de datos?

Si no lo están entonces debemos transformarlos, para esto aprenderemos sobre manipulación de datos.

```
# Ver la estructura del data.frame  
str(data_banco)
```

```
## tibble [24,299 x 8] (S3: tbl_df/tbl/data.frame)  
##   $ Sucursal           : num [1:24299] 62 62 62 62 62 62 62 62 62 62 ...  
##   $ Cajero             : num [1:24299] 4820 4820 4820 4820 4820 4820 4820 4820 4820 4820 ...  
##   $ ID_Transaccion     : chr [1:24299] "2" "2" "2" "2" ...  
##   $ Transaccion        : chr [1:24299] "Cobro/Pago (Cta externa)" "Cobro/Pago (Cta externa)" "Cobro/P...  
##   $ Tiempo_Servicio_seg: num [1:24299] 311 156 248 99 123 172 140 247 183 91 ...  
##   $ Satisfaccion       : chr [1:24299] "Muy Bueno" "Malo" "Regular" "Regular" ...  
##   $ Monto              : chr [1:24299] "2889,3" "1670,69" "3172,49" "1764.92" ...  
##   $ Tiempo_Servicio_Min: num [1:24299] 5.18 2.6 4.13 1.65 2.05 ...
```

Ejemplo - Manipulación de datos

Lo primero que necesitamos es corregir los tipos de datos, nótese que

- **Monto** tiene una mezcla de "," y "."
- **Sucursal** y **Cajero** deberían ser de tipo character
- **Satisfaccion** debe ser factor ordenado

```
data_banco <- data_banco %>%  
  mutate( Monto= str_replace(Monto, pattern = ",", replacement = ".") ) %>%  
  mutate(Sucursal= as.character(Sucursal),  
         Cajero = as.character(Cajero),  
         Satisfaccion = parse_factor(Satisfaccion,  
                                     levels= c('Muy Malo', 'Malo', 'Regular',  
                                               'Bueno', 'Muy Bueno'), ordered = T),  
         Monto= parse_number(Monto, locale = locale(decimal_mark = ".")))
```

Ejemplo - Explorar los datos

Con las columnas corregidas, podemos empezar a explorar nuestros datos; para ello podemos empezar seleccionando columnas y filtrar filas para dar vistazos a los valores que toma cada variable.

Sin embargo, es imposible poder descubrir las estructuras subyacentes de los datos de esa manera; necesitamos resumir la complejidad de los cientos, miles o millones de observaciones en unos pocos valores; aquí entran en acción las **medidas estadísticas descriptivas** que veremos en el siguiente capítulo.

FIN

Curso: Manejo de datos y reportería con R

Néstor Montaña