



#

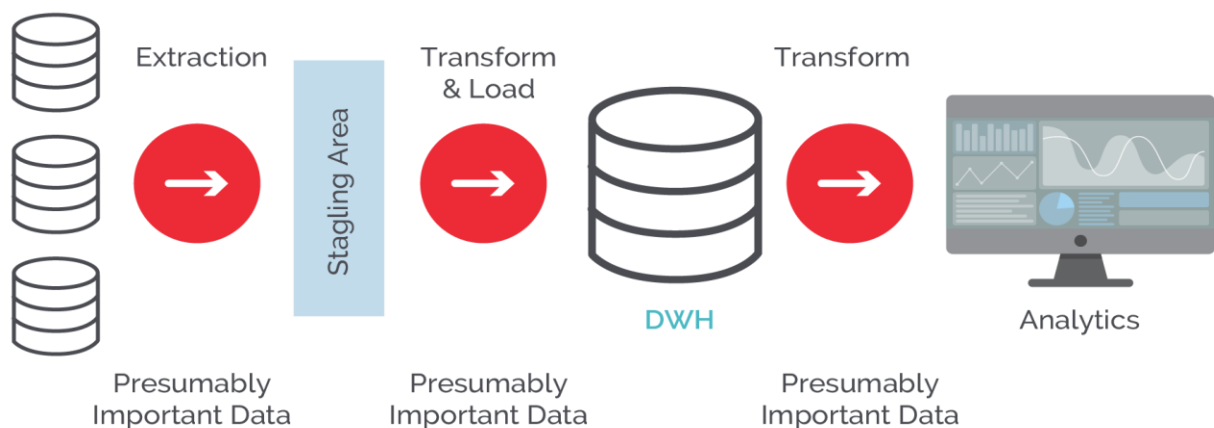
Gestión de Datos en R

Como sabemos, la información puede ser representada de muchas maneras. Tablas distintas pueden contener la misma información, cada una ordenada de una forma diferente. En esta oportunidad veremos un conjunto de herramientas que nos permitirán gestionar con facilidad nuestros datos, ayudando a disponer de la información tal como la necesitamos para nuestros análisis.

Cuando trabajamos con datos es importante tener en claro los pasos necesarios para llevar adelante un análisis. Estos pasos se repiten una y otra vez, y su identificación nos ayudará a ordenar nuestros análisis.

En un **proceso de Data Science** los datos de entrada generalmente, no están preparados para ser analizados o para ser entrada o input de algoritmos. Por esta razón, existe una serie de actividades que ayudan a “pulir” los datos de entrada, y prepararlos para que sirvan para el proceso de Data Science. A este conjunto de actividades lo denominamos preprocesamiento de datos o **proceso ETL (extracción, transformación y carga de datos)**. Se trata de una tecnología que tiene la función de integración de datos, para ofrecer una visión mejorada de los mismos.

En la Ciencia de Datos, ETL es el proceso donde se ponen a disposición los datos extraídos de múltiples fuentes, se limpian, es decir, los valida, normaliza, realiza determinadas transformaciones y los guarda en un almacén de datos para su posterior análisis.



Fuente: <https://rpubs.com/JairoAyala/ET>



Entonces, al proceso ETL se puede aplicar para:

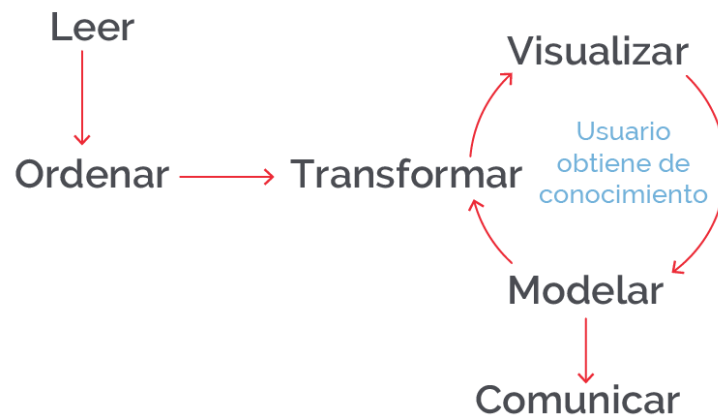
- Migración de datos de una aplicación a otra.
- Replicación de datos para copias de seguridad o análisis de redundancia.
- Procesos operativos.
- Construcción de almacenes de datos.

Cuando trabajamos con datos hablamos de 3 niveles de calidad de los datos:

- **Datos brutos** (Raw Data): pueden carecer de encabezados, contener tipos de datos erróneos (por ejemplo, números almacenados como cadenas), etiquetas de categoría errónea, codificación de caracteres desconocidos o inesperados, etc. Esto será lo primero a saldar para que podamos abrir nuestra hoja de datos.
- **Datos técnicamente correctos**: en este estado los datos pueden ser leídos en un dataframe, con nombres, tipos y etiquetas correctos, sin mayores problemas. Sin embargo, aún pueden contener errores como los que ocurren al momento de la carga o en la conversión de formatos en el almacenamiento. Por ejemplo, una variable de edad puede ser reportada como negativa o una persona menor de edad puede estar registrada para poseer una licencia de conducir. Tales incoherencias dependen obviamente del tema al que los datos pertenecen, y deben ser subsanados antes de avanzar en inferencias estadísticas válidas.
- **Datos consistentes**: los datos están listos para aplicar métodos estadísticos. Son los datos que la mayoría de las teorías estadísticas utilizan como punto de partida.

Los métodos de limpieza de datos, como la imputación de los valores perdidos, influirán en los resultados estadísticos, por lo que deben tenerse en cuenta en los siguientes análisis o interpretación de los mismos. Una vez que los resultados estadísticos se han producido pueden almacenarse para su reutilización y, por último, los resultados pueden ser formateados para incluir en los informes estadísticos o publicaciones.

En este sentido, la **Ciencia de Datos** nos provee elementos de distintas disciplinas para facilitar la interpretación, análisis y modelado de nuestros datos, culminando con la comunicación de la información útil. Veamos el esquema que se encuentra a continuación y pensemos cómo es el proceso de la gestión de datos.



Esquema de trabajo de la ciencia de datos

Como se puede ver, el análisis de datos lleva una serie de pasos que se repite: **Leer, Ordenar, Transformar, Visualizar, Modelar, Comunicar**.

La **lectura** de los datos requiere de **importar** nuestros datos a R.

El paso siguiente es **ordenarlos**, lo cual significa que **cada columna represente una variable y cada fila una observación**. Esto nos permitirá contar con una estructura consistente y enfocar nuestra atención en qué preguntas queremos hacernos.

Una vez que los datos estén ordenados, comienza la **transformación**. Esta etapa incluirá un **análisis exploratorio** de nuestros datos, permitiendo crear nuevas variables que sean funciones de variables existentes y calcular un conjunto de estadísticas resumen.

Tras esto, una buena **visualización** nos mostrará cosas que no esperábamos, planteará nuevas preguntas sobre los datos, indicará que estamos haciendo una pregunta incorrecta o que necesitamos recopilar datos diferentes.

También, recurriremos a los **modelos** como herramientas fundamentales complementarias para la visualización.

El último paso de la ciencia de datos es la **comunicación**, una parte absolutamente crítica de cualquier proyecto de análisis de datos. No importa qué tan bien sus modelos y visualización lo

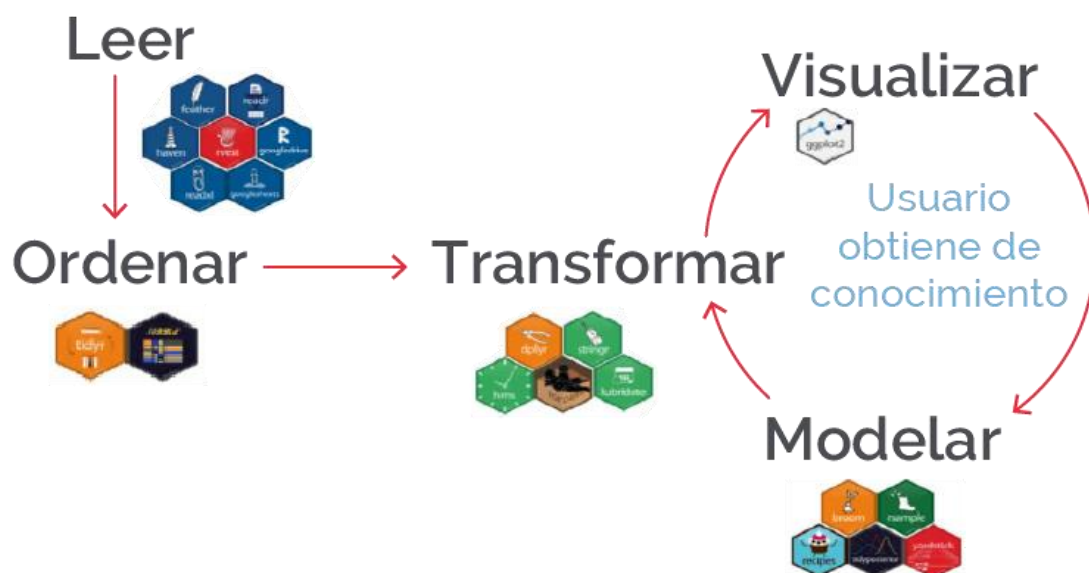


hayan llevado a comprender los datos, a menos que también pueda comunicar sus resultados a otros.

1

Introducción al Ecosistema Tidyverse

La Tidyverse es el nombre de un conjunto de paquetes diseñados para funcionar juntos desarrollados por Hadley Wickham para manipulación y estructuración de datos.



Los principios básicos en los que se basa son:

- Reutilizar las estructuras de datos
- Resolver problemas complejos combinando varias piezas sencillas
- Utilizar programación funcional

Las librerías incluidas en el paquete Tidyverse tienen como objetivo cubrir todas las fases del análisis de datos dentro de R: importar datos, ponerlos en formato ordenado (tidy), buscar relaciones entre ellos (mediante su transformación, visualización y creación de modelos) y



comunicar los resultados. También, nos ayudarán a trabajar con fechas, cadenas de caracteres o factores siguiendo los mismos principios.

1.1 / ¿Cómo instalamos Tidyverse?

Para instalar el paquete tidyverse escribimos en la consola de RStudio:

```
> install.packages("tidyverse")
```

El paquete tidyverse base actual se puede descargar del repositorio oficial CRAN.

Para poder usar las funciones, objetos y archivos de ayuda de un paquete es necesario que los cargues con la función library()

```
> library(tidyverse)

-- Attaching packages ----- tidyverse 1.3.0 --
v ggplot2 3.3.2 v dplyr 1.0.0
v tibble 3.0.1 v stringr 1.4.0
v tidyr 1.1.0 v forcats 0.5.0
v purrr 0.3.4
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag() masks stats::lag()
```

Como vemos es posible que nos devuelva que existen conflictos con funciones que tienen el mismo nombre, pero están en otros paquetes ya instalados. En este caso, es recomendable definir a que paquete pertenece la función con la forma:

```
> nombre_paquete::nombre_funcion
```



Pero es un problema que cuando surja nos daremos cuenta... ¡No nos preocupemos por ahora!
Para conocer los paquetes incluidos en esta versión escribimos:

```
> tidyverse_packages()

##[1] "broom" "cli" "crayon" "dbplyr" "dplyr" "forcats"
##[7] "ggplot2" "haven" "hms" "httr" "jsonlite" "lubridate"
##[13] "magrittr" "modelr" "pillar" "purrr" "readr" "readxl"
##[19] "reprex" "rlang" "rstudioapi" "rvest" "stringr" "tibble"

##[25] "tidyr" "xml2" "tidyverse"
```

A lo largo del curso conoceremos cada uno de estos paquetes pilares de tidyverse donde aprenderemos la lógica básica de sus funciones más relevantes.

Existen varios elementos comunes compartidos por los paquetes del universo tidyverse que describiremos a continuación:

Los **tibbles** (tbl) son similares a data.frames pero presentan ciertas mejoras que facilitan su manejo.

La clase de datos de esta estructura tabular es tbl_df y su estructura cuenta con la visualización del tamaño de la tabla de datos (observaciones x variables), el tipo de datos de cada variable (int, chr, ...) y muestra las primeras 10 filas de la estructura (cantidad que se puede modificar mediante options).

```
> require(tibble) # activamos el paquete

> x <- tibble(letras = letters)
> x
```



```
# A tibble: 26 x 1
  letras
  <chr>
1 a
2 b
3 c
4 d
5 e
6 f
7 g
8 h
9 i
10 j
# ... with 16 more rows
```

La función **tribble()** es útil para tablas pequeñas. Esta función utiliza el diseño fila por fila haciéndolo más fácil la carga de datos manual.

Es importante que prestemos atención a la entrada de datos en el código: los encabezados de las columnas están definidos por el símbolo de fórmula (es decir, comienzan con ~), y las entradas de las filas están separadas por comas.

```
> tribble(
+   ~Sexo,~Edad,~Peso,
+   #-----|--|----
+   "Mujer", 20, 47.6,
+   "Varón", 34, 72.5
+ )

# A tibble: 2 x 3
  Sexo    Edad  Peso
  <chr> <dbl> <dbl>
1 Mujer     20  47.6
2 Varón     34  72.5
```



La función **as_tibble()** sirve para forzar los objetos `data.frame` a `tbl_df`.

Comencemos leyendo nuestros datos.

2 Importación de datos

2.1 / Trabajando con Hojas de Cálculo

El Importar datos es, por lo general, el primer paso que se debe dar para realizar análisis de datos con R.

Importar datos puede ser un paso problemático si no se tienen en cuenta una serie de cuestiones previamente.

R trabaja con archivos planos de texto, del tipo `.csv`, `.txt` o `.dat`, pero existen librerías específicas para abrir y guardar otros tipos de formatos como `.xls` o `.dbl`.

2.2 / Mejores prácticas

Es recomendable que antes de empezar nos aseguremos que los datos están bien preparados para ser importados. A continuación, les presentamos una breve lista de recomendaciones para evitar problemas con la lectura de los archivos de Excel y hojas de cálculo en R.



- La primera fila de la hoja de cálculo se suele reservar para el encabezamiento. En esta fila se encuentran los nombres de las columnas.
- La primera columna se suele utilizar para identificar la unidad de muestreo. Si trabajamos con series temporales, las fechas suelen ocupar la primer columna.
- En general, resulta conveniente evitar nombres, valores o campos con espacios en blanco. Si desea concatenar palabras, hágalo insertando un punto (.) o un guión (_)
- Se prefieren los nombres cortos a los largos. Tener esto en consideración, en particular, cuando se seleccionan los nombres de las columnas.
- Intenten evitar el uso de nombres que contengan símbolos como ?, \$, %, ^, &, *, (,), -, #, ?, , , <, >, /, |, \, [], {, y }
- Eliminen cualquier comentario que haya hecho en su archivo de Excel para evitar que se agreguen columnas adicionales o NA a su archivo
- Asegúrense que las referencias a datos nulos o faltantes haya sido homogénea a lo largo del documento (Ej. NA, d/f, df, -, 9999). En caso de haber utilizado más de una, tenerlas presentes porque las necesitaremos!

2.3 / Importación de Hojas de cálculo

Desde un archivo de texto delimitado por comas (.csv)

Para abrir un archivo .csv podemos utilizar 3 funciones muy similares: **`read.table()`**, **`read.csv()`** y **`read.csv2()`**.

La diferencia de **`read.table()`** con **`read.csv()`** y **`read.csv2()`** es q para el primero se establece que el separador es la coma, mientras que para el último es el punto y coma. Es útil para archivos creados en Excel y guardados con formato csv que por la configuración regional del sistema operativo utiliza la coma como decimal (en lugar del punto de la notación anglosajona) y el punto y coma como separador.

Como estas solo se diferencian en los parámetros que vienen por default en sus argumentos, explicaremos la función genérica **`read.table()`** que resulta la más versátil de las 3. Para más información, no duden de consultar la ayuda de R.



`read.table()` utiliza una serie de parámetros que debemos conocer para que nuestra tabla se abra correctamente. Estos son:

<code>header</code>	TRUE/FALSE Sirve para indicar si la tabla comienza con los nombres de las columnas
<code>sep</code>	El <i>character</i> separador que se utilizó en la tabla. Este suele ser: coma, punto y coma, un espacio.
<code>dec</code>	El <i>character</i> que se utilizó como decimal. Este suele ser el punto o la coma
<code>nrows</code>	Por defecto es -1, porque asume que la cabecera ocupa solo una fila. En caso que tengamos una tabla cuya cabecera sea mayor, debemos corregir este parámetro con números negativos.
<code>na.strings</code>	Cómo se indican los datos faltantes en la tabla, por default es "NA"
<code>fill</code>	TRUE/FALSE. Para indicar si se deben incorporar las filas incompletas. Por defecto es TRUE.
<code>row.names</code>	Puede ser un vector que dé los nombres de las filas, o un número que dé la columna de la tabla que contiene los nombres de las filas, o un string de caracteres que dé el nombre de la columna de la tabla que contiene los nombres de las filas.

En los próximos ejercicios vamos a utilizar la hoja de datos "*mydata.csv*" que subimos a la plataforma.

El primer argumento es el nombre del archivo, siempre se escribe encerrado entre comillas (" "). Especificando algunos de los parámetros mencionados, nuestra línea de ejecución quedaría así:

```
> mi_csv <- read.table( "mydata.csv", header = TRUE,  
                        sep = ",", dec = ".")  
> head(mi_csv)
```



Los parámetros que no nos interesa modificar respecto a los parámetros por defecto, no los incorporamos.

La función ***read.table()*** también lee otros formatos como .dat

Desde un archivo de Excel (.xls, .xlsx)

Para abrir los archivos de Excel es necesario incorporar la librería **readxl**.

```
> install.packages("readxl")  
> library(readxl)
```

Esta librería tiene 3 funciones homólogas a las que veíamos para .csv: ***read_excel()***, ***read_xls()*** y ***read_xlxs()***.

La principal particularidad de estos ficheros es que ya no tenemos una hoja de cálculos, sino un libro de hojas de cálculo.

Como vimos, la clase data.frame es bidimensional, por lo que deberemos abrir de a una las pestañas.



Para indicar una pestaña distinta a la primera, tenemos el parámetro **sheet** (pestaña), donde deberemos indicar la pestaña que deseamos visualizar en R.

A continuación, les dejamos los parámetros de mayor interés:

sheet	Pestañas. En caso de querer leer otra pestaña que no sea la primera, debemos especificarlo
skip	En caso que tengamos una tabla cuya cabecera sea mayor a 1, debemos corregir este parámetro con números positivos.
col_names	TRUE para usar la primera fila como nombre de las columnas
na	Cómo se indican los datos faltantes en la tabla, por default es "" (vacío)

Nuestra línea de comando quedaría:

```
> mi_excel <- read_excel("mydata.xlsx", col_names = TRUE)
> head(mi_excel)
```




Resulta conveniente que antes de abrir una tabla en R veamos su estructura con un lector de texto plano (del tipo Bloc de Notas). Nos interesa observar:

- Qué carácter se usa como separador?
- Qué carácter se usa para indicar los decimales?
- Cómo se indican los datos faltantes?
- Cuántas filas tiene el encabezado?

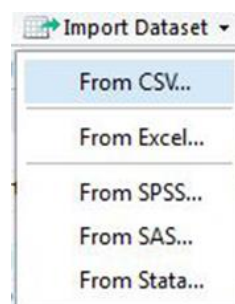
2.4 / Lectura de archivos de datos usando el menú de RStudio

Otra opción es realizar la lectura a través de uno de los menús de RStudio.

El botón que debemos pulsar para iniciar este proceso se encuentra en el panel de entorno

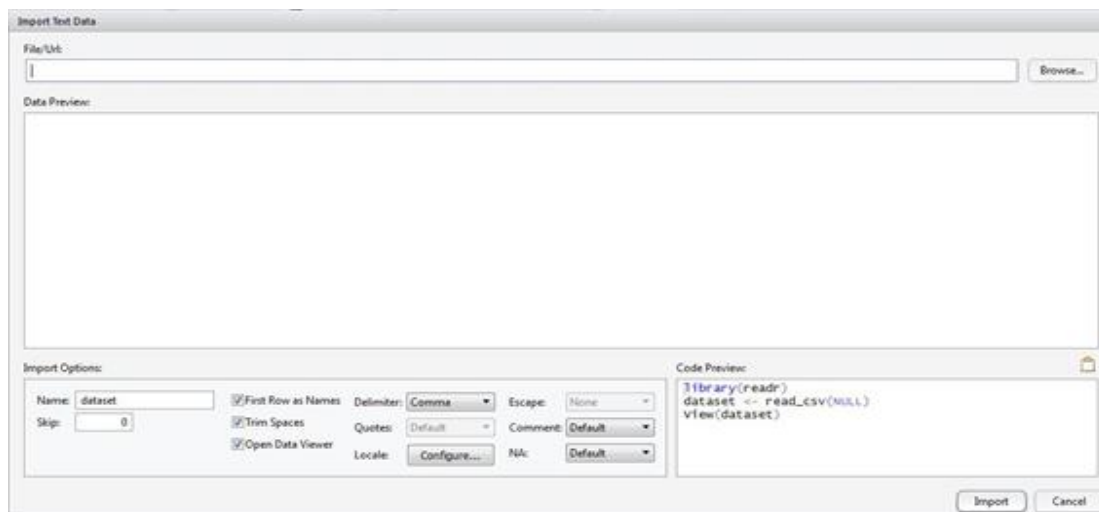
(*Enviroment*) y tiene la forma  Import Dataset ▾

Al pulsarlo nos despliega un listado de formatos de importación:





El primero de ellos es el formato CSV y si hacemos click con el mouse nos abrirá la siguiente ventana:



Dentro de la ventana **Import Text Data** podemos definir distintas opciones para la lectura de archivos de datos de texto plano (ASCII). Está compuesta por un casillero superior donde vamos a buscar (Browse...) el archivo que queremos leer. Un área en el medio de la pantalla donde se previsualizan los datos del archivo. Una zona inferior para configurar las opciones de la importación.

Mencionaremos algunas:

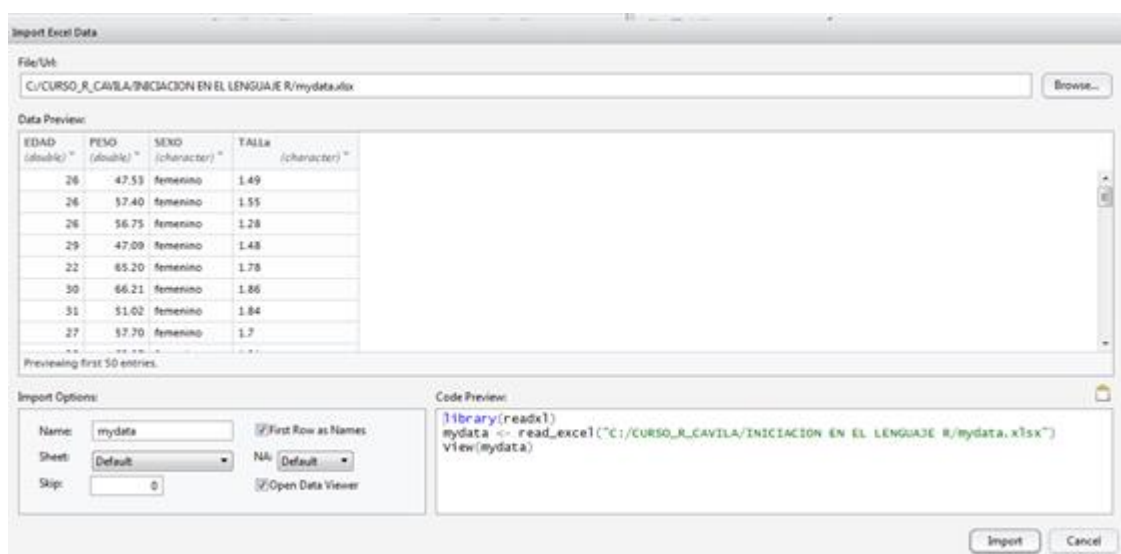
Name	nombre del dataframe (por defecto es dataset)
Skip	líneas a saltar para comenzar a leer el archivo (por defecto es 0)
First Row as Names	es igual al argumento header, es decir, define si la primer línea del archivo contiene los nombres de las variables.
Trim Spaces	elimina o no los espacios encontrados entre dato y separador
Open Data Viewer	establece si al término de la lectura abre la tabla en el visualizador de RStudio



Delimiter	para seleccionar cual es el caracter separador del archivo. Entre ellos se encuentra coma (Comma), punto y coma (Semicolon), tabulación (Tab) y espacio en blanco (Whitespace).
Locale	donde podemos configurar un grupo de formatos (fecha, decimal, etc)

Y finalmente, un rectángulo inferior derecho para previsualizar el código generado en R.

Mostraremos a continuación como queda si es configurado para leer el archivo **mydata.csv** que habíamos importado desde la consola.



En el rectángulo **Code Preview** podemos observar que en la primera línea RStudio activa un paquete llamado **readr** por lo que no utiliza el paquete base de R para hacer la importación.

2.5 / Formateo de dataframes

Una vez leída la tabla de datos, R almacena los datos en la estructura de un dataframe. Seguro nos interese conocer cuántas variables y observaciones tiene nuestro set de datos, con qué tipo de datos trabajamos y si tiene una cabecera con el nombres de las variables correctas.



Para explorar inicialmente el dataframe usamos algunas de las funciones vistas como:

str()	Devuelve la estructura del dataframe
ncol()	Devuelve el número de columnas (variables) del dataframe
nrow()	Devuelve el número de filas (observaciones) del dataframe
dim()	Devuelve la dimensión del dataframe (filas por columnas)

Podemos encontrarnos con situaciones donde la cabecera de la tabla de datos no exista o que deseamos cambiar los nombres existentes.

La función que nos permite manipular la cabecera del dataframe es **names()**.

Con ella podemos hacer:

<code>names(dataframe)</code>	Mostrar los nombres de las variables
<code>names(dataframe) <- c("var1", "var2", ...)</code>	Crear nombres de variables en la cabecera
<code>names(dataframe)[4] <- "var4"</code>	Modificar nombres de variables puntuales

Por ejemplo, vamos a aplicar la función **names()** al dataframe **mi_csv**:

```
> mi_csv <- read.table( "mydata.csv", header = TRUE,  
                        sep = ",", dec = ".")  
  
> names(mi_csv)  
[1] "EDAD" "PESO" "SEXO" "TALLa"
```

Cuando las variables no tienen nombres, por defecto vemos nombres del tipo V1, V2,



Si queremos cambiar el nombre de una variable, usamos un índice para modificar sólo aquella que es de nuestro interés:

```
> names(mi_csv)[4] <- "TALLA"
> names(mi_csv)
[1] "EDAD" "PESO" "SEXO" "TALLA"
```

Otra de las tareas de formateo del dataframe es hacer un recorte o eliminar variables que deseamos descartar. Para eliminar variables (columnas) específicas, podemos utilizar el sistema de índices y el objeto nulo (NULL) de la siguiente manera:

```
> mi_csv[,2] <- NULL
> names(mi_csv)
[1] "EDAD" "SEXO" "TALLA"
```

De esta forma, eliminamos la segunda columna (utilizamos índice [, 2]) y comprobamos que efectivamente la variable “PESO” no existe más.

Si lo que necesitamos es hacer un recorte del dataframe, para quedarnos con un subconjunto de variables menor al original, se puede aplicar una selección por índice:

```
> datos <- mi_csv[,2:3]
> names(datos)
[1] "SEXO" "TALLA"
```

También podemos reordenar las variables, siempre usando el sistema de índices, potente y flexible, propio de R:

```
> mi_csv <- mi_csv[c(2,3,1)]
> names(mi_csv)
[1] "SEXO" "TALLA" "EDAD"
```




2.6 / Escritura de archivos

Una vez realizado el tratamiento específico de los datos (limpieza, validación, creación de nuevas variables, etc.) necesitamos poder exportar la información en formatos adecuados que nos asegure volver a leerlos en el futuro por R o por otros softwares.

R base solo permite exportar las tablas de datos a formato de texto plano (como .txt, .csv o .dat). Sin embargo, a través de los paquetes específicos se pueden guardar en otros formatos diversos. La función utilizada por R base es **`write.table()`**, que genera un archivo con los datos almacenados en un objeto, típicamente un dataframe, aunque puede ser de cualquier otro (vector, matriz, etc) Veámoslo con un ejemplo:

En nuestra sesión contamos con el dataframe **`mi_csv`** que importamos del archivo **`mydata.csv`**.

Vamos a probar el funcionamiento de la función **`write.table()`** para exportar el dataframe en un archivo al que denominaremos **`mi_csv.txt`**

```
> write.table(mi_csv, file = "mi_csv.txt", sep = ",",  
              row.names = FALSE)
```

Analicemos esta línea detenidamente.

Tras la apertura del paréntesis va el nombre del dataframe con los datos a exportar (en este caso la variable **`mi_csv`**). A continuación, en el parámetro **`file`** escribimos entre comillas el nombre que queremos darle al fichero de exportación (en el ejemplo **`"mi_csv.txt"`**). El parámetro **`sep`** sirve para definir el caracter delimitador de columnas en el archivo a crear. En este ejemplo, utilizamos como argumento la coma. Finalmente, el parámetro opcional **`row.names`** está en **`FALSE`** para que no incorpore una variable con el nombre de las filas.

Si queremos confirmar el formato del archivo exportado podemos abrirlo con cualquier editor de texto (notepad, etc).



Resumen

Los Data Frames:

- Se usan para trabajar con datos tabulados
- Es un tipo de lista en la que todos los elementos tienen igual longitud
- Las columnas se trabajan como vectores.
- A diferencia de las matrices puede tener elems de diferentes clases
- Podemos abrir .csv, .dat o .txt con: **`read.table()`, `read.csv()` o `read_csv2()`**
- Podemos abrir .xlsx o xls con la librería readxl: **`read_excel()`, `read_xls()` o `read_xlsx()`**
- La función names() nos muestra los nombres de las columna. También, nos permite editarlos.
- Para guardar un .csv, .dat o .txt con: **`write.table()`, `write.csv()` o `write_csv2()`**

3

Datos Ausentes

Es común que cuando pedimos responder una encuesta o formulario, puede que alguna pregunta quede sin responder. Entonces, nos preguntamos: ¿descartamos la encuesta porque le falta una respuesta, o tratamos de solucionar ese inconveniente para poder utilizar la información provista por el resto de las herramientas?

Cuando recolectamos datos de manera automatizada o semi-automatizada, puede ser que los datos estén mal cargados, o no se les dio importancia a la hora de cargarlos. Por ejemplo, si usamos un GPS para rastrear los recorridos de los colectivos. ¿Qué hacemos si faltan datos? ¿Descartamos el día o tratamos de estimar los datos faltantes?

Puede ocurrir que se nos escape una tecla e ingresemos un número mal, por ejemplo en una cuenta corriente de un cliente, y el sistema no lo chequea. Pero no nos damos cuenta del error, sino que seguimos operando en el sistema hasta que queremos medir estadísticas y preprocesar



los datos. Recién ahí nos damos cuenta del error. ¿Qué hacemos? Sabemos que el dato está mal, pero en general no podemos simplemente borrar el cliente, porque tiene muchos datos relacionados. ¿Y entonces...?

Los valores faltantes o perdidos (NA) complejizan el análisis estadístico de las bases de datos. Dependerá del operador si omite los elementos de un conjunto de datos que contienen valores perdidos o si lo imputa (sustituye el valor), pero su falta es algo que debe tratarse antes de cualquier análisis.

En la práctica algunos de los softwares de cálculo numérico comúnmente utilizado puede confundir un valor perdido con un valor o categoría por defecto. Por ejemplo, en Microsoft Excel, el resultado de sumar el contenido de una celda que contiene el número 1 con una celda vacía resulta igual a 1. Este comportamiento es definitivamente indeseado ya que Excel imputa silenciosamente un cero donde debería haber dicho que tiene un valor que es “incapaz de calcular”.

Otro error comúnmente encontrado es confundir un valor NA en los datos categóricos con la categoría “desconocido”. Si el desconocido o el típico “No Sabe/No Contesta” es realmente una categoría, debe agregarse como un nivel de factor para que pueda analizarse apropiadamente.

Consideremos como ejemplo una variable categórica que representa el lugar de nacimiento. Aquí, la categoría “desconocida” significa que no tenemos conocimiento sobre dónde nace una persona. En contraste, NA indica que no tenemos información para determinar si el lugar de nacimiento es conocido o no.

El comportamiento de la funcionalidad central de R es completamente coherente con la idea de que el analista debe decidir qué hacer con los datos que faltan.

La opción común de “dejar afuera los registros con datos faltantes” es soportada por muchas funciones básicas del lenguaje a través del parámetro ***na.rm***

Para ver cómo trabajar, continuemos trabajando con base de datos “mydata.csv”:



```
> datos <- read.table( "mydata.csv", header = TRUE,  
                      sep = ",", dec = ".")
```

Previo a comenzar, corregimos el nombre de la variable, tal como vimos anteriormente:

```
> names(datos)[4] <- "TALLA" #corregimos el nombre
```

La variable TALLA tiene un dato del tipo NA en la fila 16:

```
> which(is.na(datos$TALLA) == TRUE)  
[1] 16
```

Por tanto, cuando queramos realizar la media con la función **mean()**, R nos dirá que no está disponible:

```
> mean(datos$TALLA) # calculo la media  
[1] NA
```

Para saldar esto, podemos utilizar el parámetro **na.rm = TRUE** que excluye el valor NA:

```
> mean(datos$TALLA, na.rm = TRUE)  
[1] 1.698776
```

Una función útil es la función **complete.cases()** que nos devuelve un vector con las filas que se encuentren completas. La fila que no se encuentra completa devuelve un FALSE.

```
> complete.cases(datos$TALLA)  
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  
[11] TRUE TRUE TRUE TRUE TRUE FALSE TRUE TRUE TRUE TRUE
```



```
[21] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[31] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[41] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Si queremos la tabla completa, podemos utilizar ese vector como índice, bajo la forma:

```
> datos[complete.cases(datos$TALLA),]
```

La función **na.omit()** devuelve el mismo resultado:

```
> na.omit(datos)
```

4

Lectura sin formato predeterminado

Todas las formas de importación o lectura de datos en R traen incorporadas transformaciones de tipos de datos de forma predeterminada que se basan en suposiciones que el lenguaje realiza por nosotros. Esta situación, que a veces puede ser de ayuda, en la mayoría de las ocasiones puede acarrear inconvenientes.

Estos problemas se dan cuando estamos tratando con tablas de datos en bruto, donde desconocemos el nivel de calidad de los datos y sospechamos que tendremos que realizar un proceso de depuración.

El caso más típico es el de la lectura de variables categóricas y su coerción a tipo de dato Factor (que es el que corresponde) pero basándose en categorías automáticas extraídas de todas las diferentes categorías encontradas en la fuente de datos.



Por ejemplo para el caso de la variable sexo, si los datos crudos tuviesen baja calidad podemos encontrarnos para la categoría Femenino con los siguientes valores posibles: Femenino, femenino, FEMENINO, etc.

Si dejamos que durante la importación el lenguaje automáticamente construya factores, cada una de las diferentes formas de escribir la categoría Femenino será un nivel del factor sexo. Esto generará niveles erróneos por lo que tendremos que reconstruir el factor.

Para evitar estos problemas tenemos que importar los archivos de datos tal como se encuentran utilizando los argumentos opcionales adecuados que las funciones de lectura traen incorporados.

La familia de funciones **read.table()** tiene la opción **as.is = TRUE** para indicar que haga la lectura de los datos “como están”. Entonces las variables categóricas serán incorporadas como caracteres en lugar de factores. La conversión a factor se hará de forma manual luego de procesar las categorías de las variables cualitativas.

4.1 / Conversión de tipo de datos

Cuando decimos conversión de tipos de datos nos referimos a la coerción de las clases de los elementos a otras clases nuevas.

Las funciones que utilizaremos son de la familia **as.** y van a vincularse a los tipos de datos básicos del lenguaje con dos salvedades.

Las funciones más utilizadas para gestionar conversiones de datos en dataframes son:



<code>as.numeric</code>	Convierte a tipo numérico
<code>as.integer</code>	Convierte a tipo entero
<code>as.character</code>	Convierte a tipo carácter
<code>as.logical</code>	Convierte a tipo lógico o booleano

Además existen dos coerciones útiles para el trabajo previo al análisis de variables categóricas o cualitativas.

<code>as.factor</code>	Convierte a tipo factor
<code>as.ordered</code>	Convierte a tipo factor ordenado

Conversión de carácter a factor (automático)

Continuemos con el ejemplo de la variable categórica SEXO de tipo character codificada con H (hombre) y M (mujer). Estamos frente a un tipo carácter.

```
> class(datos$SEXO)
[1] "character"
```

Utilizamos la función de coerción **as.factor()**

```
> datos$SEXO <- as.factor(datos$SEXO)

> class(datos$SEXO) # Verificamos el tipo convertido

[1] "factor"
```

Vemos los niveles del factor (como la coerción fue automática se tomaron los valores de las categorías originales para definir los niveles)



```
> levels(datos$SEXO)
[1] "H" "M"
```

Conversión de carácter a factor (etiquetas personalizadas)

Variable categórica SEXO de tipo character codificada con H (hombre) y M (mujer)

Estamos frente a un tipo carácter

```
> class(datos$SEXO)
[1] "character"
```

Utilizamos la función **factor()** para poder incorporar argumentos a medida (definimos las etiquetas Hombre y Mujer – el orden siempre respeta el ordenamiento alfabético ascendente).

```
> datos$SEXO <- factor(datos$SEXO, labels = c("Hombre", "Mujer"))
> class(datos$SEXO) # Verificamos el tipo convertido
[1] "factor"
> levels(datos$SEXO)
[1] "Hombre" "Mujer"
```

5

Paquete dplyr

El **paquete dplyr** contiene una colección de funciones para realizar operaciones comunes de manipulación de datos como: filtrar por fila, seleccionar columnas específicas, reordenar filas, añadir nuevas filas y agregar datos. Como gran ventaja, nos permite realizar: *split-apply-combine*.

Esto es, básicamente, calcular estadísticos (y otros) por grupos.

- *Split*: cortar por grupo
- *Apply*: aplicar la función
- *Combine*: combinar los grupos para dotarles de una estructura.



Si bien esta tarea también se puede realizar con la función **aggregate()** de la sintaxis básica de R, veremos la facilidad que nos brinda el paquete *dplyr*.

La mayor ventaja de *dplyr* es que su sintaxis es: más sencilla, más consistente, está enfocada en datasets en lugar de a vectores e integra como conector entre sus funciones al operador `%>%` (pipe) logrando una única tubería (“pipeline”).

De esta forma, proporciona una “gramática” (verbos) para la manipulación y operaciones con dataframes y/o tibbles.

Las funciones del paquete responden a las siguientes acciones (verbos):

- **select()**: devuelve un conjunto de columnas (variables)
- **rename()**: renombra variables en una conjunto de datos
- **filter()**: devuelve un conjunto de filas según una o varias condiciones lógicas
- **arrange()**: reordena filas de un conjunto de datos
- **mutate()**: añade nuevas variables/columnas o transforma variables existentes
- **summarise()** / **summarize()**: genera resúmenes estadísticos de diferentes variables en el conjunto de datos.
- **group_by()**: agrupa un conjunto de filas seleccionado, en un conjunto de filas de resumen de acuerdo con los valores de una o más columnas o expresiones.

Todas estas funciones tienen como particularidad que su primer argumento es el dataframe al que se le realizará la operación, mientras que los subsiguientes argumentos describen cómo realizar tal operación.

Para ello, podemos referirnos a las columnas sin utilizar el operador `$`. Es decir, sólo podemos utilizar el nombre de la columna (variable de estudio). Como prerequisite, los datos deben estar bien organizados: debe existir una observación por columna y cada columna debe representar una variable, medida o característica de esa observación.

Recuerda que, si ya instalaste el paquete Tidyverse, puedes activarlo o también instalarlo solo y activarlo mediante:



```
> require (tidyverse)
> require (dplyr)
```

Comencemos a ver cómo funciona cada una de las funciones mencionadas aplicándolas al dataset “*iris*”, el cual contiene las mediciones en centímetros de las variables longitud (Length) y ancho (Width) de los pétalos (Petal) y sépalos (Sepal) de 50 flores de cada una de las 3 especies (Species) del género *Iris*: *Iris setosa*, *Iris versicolor* e *Iris virginica*

```
> data(iris) # cargar el dataset en el workspace
> names(iris) # nombre de las columnas
[1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width"
"Species"
```

```
> iris %>% select(Sepal.Length, Sepal.Width)
```

Función *select()*

Esta función principalmente **selecciona las variables que especificamos** devolviendo un conjunto de datos “recortado”. Por ejemplo, si queremos seleccionar las columnas “Sepal.Length” y “Sepal.Width” de *iris* podemos escribir:

Como también,

```
> select(iris, Sepal.Length, Sepal.Width)
```

Si queremos todas las variables menos “Species”, podemos escribir:

```
> iris %>% select(-Species)
```



	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1	5.1	3.5	1.4	0.2
2	4.9	3.0	1.4	0.2
3	4.7	3.2	1.3	0.2
4	4.6	3.1	1.5	0.2
5	5.0	3.6	1.4	0.2
6	5.4	3.9	1.7	0.4
7	4.6	3.4	1.4	0.3

También, podemos hacerlo indicando las columnas con números:

```
> iris %>% select(1,3) # selecciona columna 1 y 3
```

	Sepal.Length	Petal.Length
1	5.1	1.4
2	4.9	1.4
3	4.7	1.3
4	4.6	1.5
5	5.0	1.4
6	5.4	1.7
7	4.6	1.4
8	5.0	1.5
9	4.4	1.4
10	4.9	1.5

Esto también lo podemos usar para reordenar un dataframe. Por ejemplo, si queremos pasar "Specie" a la primera columna, podemos hacer:

```
> iris %>% select("Species",everything())
```



	Species	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1	setosa	5.1	3.5	1.4	0.2
2	setosa	4.9	3.0	1.4	0.2
3	setosa	4.7	3.2	1.3	0.2
4	setosa	4.6	3.1	1.5	0.2
5	setosa	5.0	3.6	1.4	0.2
6	setosa	5.4	3.9	1.7	0.4
7	setosa	4.6	3.4	1.4	0.3
8	setosa	5.0	3.4	1.5	0.2
9	setosa	4.4	2.9	1.4	0.2
10	setosa	4.9	3.1	1.5	0.1

Función *rename()*

Esta función nos permite **renombrar variables** mientras que mantiene las demás no mencionadas. Por ejemplo, podemos cambiar el nombre de algunas variables haciendo:

```
> iris %>% rename("Especie"= "Species", "Long  
Sepalo"="Sepal.Length", "Ancho Sepalo"= "Sepal.Width")
```

Función *filter()*

Así como la función ***select()*** es utilizada para **seleccionar columnas**, la función ***filter()*** nos permite **filtrar filas** del conjunto de datos, produciendo un subconjunto.

Es similar a la función R base ***subset()*** y su estructura básica es:

- El primer argumento es el nombre del dataframe o tibble.
- El segundo y siguientes argumentos son las expresiones que filtran ese conjunto de datos.

```
> iris %>% filter(Species == "versicolor")
```



	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	7.0	3.2	4.7	1.4	versicolor
2	6.4	3.2	4.5	1.5	versicolor
3	6.9	3.1	4.9	1.5	versicolor
4	5.5	2.3	4.0	1.3	versicolor
5	6.5	2.8	4.6	1.5	versicolor
6	5.7	2.8	4.5	1.3	versicolor
7	6.3	3.3	4.7	1.6	versicolor
8	4.9	2.4	3.3	1.0	versicolor
9	6.6	2.9	4.6	1.3	versicolor
10	5.2	2.7	3.9	1.4	versicolor
...					

Esta función utiliza los operadores propios del lenguaje R que ya hemos visto.

Veamos cómo funciona con algunos ejemplos:

```
# filtrar sólo especie setosa

> filter(iris, Species == 'setosa')

# filtrar especie setosa o virginica

> filter(iris, Species == 'setosa' | Species == 'virginica')

# especie setosa con longitud de sépalo menor a 5 mm

> filter(iris, Species == 'setosa', Sepal.Length < 5)
```

Función *arrange()*

Se utiliza para **ordenar las filas de un conjunto de datos** de acuerdo con una o varias columnas (variables). Por defecto, el ordenamiento es ascendente.



Para probar cómo funciona, ordenemos *Iris* por la variable *longitud del pétalo (Petal.Length)*:

```
> iris %>% arrange(Petal.Length) # ascendente por defecto
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	4.6	3.6	1.0	0.2	setosa
2	4.3	3.0	1.1	0.1	setosa
3	5.8	4.0	1.2	0.2	setosa
4	5.0	3.2	1.2	0.2	setosa
5	4.7	3.2	1.3	0.2	setosa
6	5.4	3.9	1.3	0.4	setosa
7	5.5	3.5	1.3	0.2	setosa
8	4.4	3.0	1.3	0.2	setosa
9	5.0	3.5	1.3	0.3	setosa
10	4.5	2.3	1.3	0.3	setosa
.....					

Para hacerlo de forma descendente, podemos escribir:

```
> iris %>% arrange(desc(Petal.Length)) # descendente
```

También, nos permite combinar los ordenamientos:

```
# longitud de pétalo ascendente y ancho descendente  
  
> iris %>% arrange(Petal.Length, desc(Petal.Width) )
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	4.6	3.6	1.0	0.2	setosa
2	4.3	3.0	1.1	0.1	setosa
3	5.8	4.0	1.2	0.2	setosa
4	5.0	3.2	1.2	0.2	setosa



5	5.4	3.9	1.3	0.4	setosa
6	5.0	3.5	1.3	0.3	setosa
7	4.5	2.3	1.3	0.3	setosa
8	4.7	3.2	1.3	0.2	setosa
9	5.5	3.5	1.3	0.2	setosa
10	4.4	3.0	1.3	0.2	setosa
...					

Función *mutate()*

A menudo, tendremos la necesidad de **crear nuevas variables que se calculan a partir de variables existentes**. La función *mutate()* nos permite computar fácilmente las transformaciones de nuestras variables.

Como ejemplo, vamos crear una nueva variable con la forma de los pétalos (*Petal.Shape*) como la relación entre el ancho y el largo, y seleccione solo las nuevas variables y *especies*.

```
> iris %>%  
+   mutate(Petal.Shape = Petal.Width / Petal.Length,  
+          Sepal.Shape = Sepal.Width / Sepal.Length) %>%  
+   select(Species, Petal.Shape, Sepal.Shape)
```

	Species	Petal.Shape	Sepal.Shape
1	setosa	0.14285714	0.6862745
2	setosa	0.14285714	0.6122449
3	setosa	0.15384615	0.6808511
4	setosa	0.13333333	0.6739130
5	setosa	0.14285714	0.7200000
6	setosa	0.23529412	0.7222222
7	setosa	0.21428571	0.7391304
8	setosa	0.13333333	0.6800000



```
9      setosa  0.14285714  0.6590909
10     setosa  0.06666667  0.6326531
...
```

Observemos que la función realiza el cálculo e incorpora una nueva variable por cada observación con el resultado. De esta forma, se pueden construir múltiples variables en la misma expresión, solamente separadas por comas.

Como dijimos al comienzo del texto, se pueden abordar muchas tareas de análisis de datos utilizando el paradigma de *split-apply-combine* (dividir-aplicar-combinar): dividir los datos en grupos, aplicar un análisis a cada grupo y luego combinar los resultados.

A continuación, veremos 2 funciones que hacen esta tarea aun mucho más fácil.

Función *summarise()* y *group_by()*

La función ***summarise()*** colapsa un conjunto de datos en una sola fila. Funciona de forma análoga a la función ***mutate()***, excepto que en lugar de añadir nuevas columnas crea un nuevo tibble de una sola observación.

Esta función combina bien junto a ***group_by()***, quien toma como argumentos los nombres de columna que contienen las variables categóricas para las que desea calcular las estadísticas de resumen, cambiando la unidad de análisis del conjunto de datos completo a grupos específicos.

Así cuando usamos ***summarise()*** o ***summarize()*** los cálculos se aplicarán automáticamente “por grupo”, produciendo una salida con tantas filas como grupos existan.

Veamos cómo se calcularía la media de la longitud de pétalos por especie:

```
> iris %>%
+   group_by(Species) %>%
+   summarise(Mean.Petal.Length = mean(Petal.Length))
```




En la salida vemos en listadas las especies y el valor de las medias:

```
`summarise()` ungrouping output (override with `.groups`  
argument)  
# A tibble: 3 x 2  
  Species Mean.Petal.Length  
  <fct>          <dbl>  
1 setosa          1.46  
2 versicolor      4.26  
3 virginica        5.55
```

Veamos lo sencilla que resulta la escritura cuando queremos realizar varios procesos. Podemos agrupar por múltiples columnas y crear nuevas variables.

```
> iris %>%  
+   mutate(Petal.Long = Petal.Length > 5) %>%  
+   group_by(Species, Petal.Long) %>%  
+   summarise(Mean.Petal.Length = mean(Petal.Length),  
+             n.Petals = length(Petal.Length),  
+             sd.Petal.Length = sd(Petal.Length),  
+             SE.Petal.Length = sd(Petal.Length) /  
+             sqrt(length(Petal.Length)))
```



```
`summarise()` regrouping output by 'Species' (override with  
`.groups` argument)  
# A tibble: 5 x 6  
# Groups:   Species [3]  
  Species Petal.Long Mean.Petal.Length n.Petals sd.Petal.Length  
SE.Petal.Length  
  <fct>    <lgl>      <dbl>      <int>      <dbl>      <dbl>  
1 setosa  FALSE      1.46        50      0.174      0.0246  
2 versicolor FALSE      4.24        49      0.459      0.0655  
3 versicolor TRUE       5.1         1      NA        NA  
4 virginica FALSE      4.87         9      0.158      0.0527  
5 virginica TRUE       5.70        41      0.489      0.0764
```

Podemos observar que aparece un valor de NA en las variables sd y SE, porque el número de pétalos largos es solo uno y, por lo tanto, no es posible calcular la desviación estándar.

Gestión de factores

Un **factor** es una clase para objetos (variables) que contienen datos categóricos. Su estructura está compuesta por dos vectores: en uno almacena índices enteros que se usan para especificar una clasificación (*levels*, niveles), y el segundo es un vector de caracteres que contiene las categorías (*labels*, etiquetas), a los que hace referencia el primer vector.

El **paquete forcats** es parte del ecosistema Tidyverse pensado para trabajar con factores.

Para conocer sus potencialidades primero debemos activarlo

```
> require(forcats)
```

Vamos a trabajar con un conjunto de datos llamado **Datos_salud.xlsx**



```
> require(readxl)

> Datos_salud <- read_excel("Datos_salud.xlsx", sheet = 1, col_names=T)
```

Vemos la estructura de los datos:

```
> str(Datos_salud)
```

```
tibble [19 x 9] (S3: tbl_df/tbl/data.frame)
 $ Enfermedad      : chr [1:19] "Si" "Si" "No" "Si" ...
 $ Sexo            : chr [1:19] "Varon" "Mujer" "Mujer" "Mujer"
 ...
 $ Civil           : chr [1:19] "Soltero" "Viudo" "Casado"
 "Soltero".
 $ Esalud          : chr [1:19] "Mala" "Muy mala" "Buena" "Mala"
 ...
 $ Ciudad          : chr [1:19] "La Plata" "La Plata" "La Plata" ...
 $ Comorbilidades: chr [1:19] "EPOC" "Gastritis"
 "aterosclerosis" ...
 $ ACTIFIS         : num [1:19] 1 1 1 2 1 1 2 2 1 2 ...
 $ VECES           : num [1:19] 1 4 4 0 5 5 0 0 3 0 ...
 $ TABACO          : num [1:19] 0 0 0 1 0 1 0 1 11 0 ...
```

Como podemos observar nuestro objeto tiene 9 variables de tipo carácter y 19 observaciones.

Dado que el archivo fue importado desde un archivo xlsx (Excel) mediante una función del **paquete readxl**, se generó un tibble y los tibble no transforman automáticamente en factor al tipo character.

A lo largo de todo el documento haremos la transformación manual de cada variable.



Comencemos transformando en factor la variable “*Enfermedad*” usando la función base de R, ***factor()***

```
> Datos_salud$Enfermedad <- factor(Datos_salud$Enfermedad)

> levels(Datos_salud$Enfermedad) # vemos los niveles del factor
```

¿Cuáles fueron los niveles obtenidos?

as_factor() y ***fct_recode()***

La función del paquete **forcats** para realizar la misma tarea se llama ***as_factor()***. Realicemos el mismo procedimiento con esta función:

```
> Datos_salud$Sexo <- as_factor(Datos_salud$Sexo)

> levels(Datos_salud$Sexo)
[1] "Varon"      "Mujer"      "Masculino"  "Femenino"
```

Aquí se nos presenta un problema muy común cuando trabajamos con datos reales cargados por diferentes usuarios o cuando unimos bases de diverso origen. Las categorías se encuentran etiquetadas de manera diferente aunque conceptualmente se refieran a lo mismo (ejemplo: “Femenino” - “Mujer”)

Para corregir este inconveniente el paquete **forcats** nos ofrece una función que recodifica los niveles. Se llama ***fct_recode()***



```
> Datos_salud$Sexo <- fct_recode(Datos_salud$Sexo,  
                                Varon="Masculino",  
                                Mujer="Femenino")  
  
> levels(Datos_salud$Sexo)  
  
[1] "Varon" "Mujer"
```

En los argumentos le indicamos que “Masculino” es igual a “Varón” y “Femenino” igual a “Mujer”.

Esto provoca que en todos los casos donde aparezca “Masculino” sea reemplazado por “Varon” y cuando aparezcan “Femenino” se cambie por “Mujer”. Luego, verificamos que los niveles sean los dos que necesitamos.

Paso siguiente, podemos pedir un listado de frecuencia de los niveles de la variable Sexo. Para ellos usamos la función **fct_count()**.

```
> fct_count(Datos_salud$Sexo)
```

```
# A tibble: 2 x 2  
  f      n  
  <fct> <int>  
1 Varon    10  
2 Mujer     9
```

fct_explicit_na()

Continuamos con la siguiente variable, “Civil”.

Primero hacemos la transformación y le pedimos sus niveles:



```
> Datos_salud$Civil <- as_factor(Datos_salud$Civil)

> levels(Datos_salud$Civil)
[1] "Soltero"    "Viudo"      "Casado"     "Divorciado"
```

Tras esto, queremos conocer la frecuencia de cada uno:

```
> fct_count(Datos_salud$Civil)
```

```
# A tibble: 5 x 2
  f          n
  <fct>    <int>
1 Soltero      7
2 Viudo        3
3 Casado       5
4 Divorciado   3
5 NA           1
```

Como vemos aparecen los 4 niveles más un valor faltante (NA).

Supongamos que deseamos mostrar dentro de una tabla de frecuencia la cantidad de valores perdidos o desconocidos que tenemos de la variable Estado Civil. Debemos etiquetar ese NA para poder visualizarlo. Para ellos el paquete nos ofrece la función **fct_explicit_na()**. ¡Veamos cómo se usa!

```
> Datos_salud$Civil <-
  fct_explicit_na(Datos_salud$Civil, na_level = "Desconocido")
> levels(Datos_salud$Civil)
[1] "Soltero"      "Viudo"        "Casado"       "Divorciado"
"Desconocido"
```



fct_relevel() y **fct_rev()**

Continuemos con la variable estado de salud (*Esalud*).

Como se trata de una variable categórica ordinal vamos a generar el factor mediante la función R base **ordered()**

```
> Datos_salud$Esalud <- ordered(Datos_salud$Esalud)

> levels(Datos_salud$Esalud)

[1] "Buena"      "Mala"       "Muy buena"  "Muy mala"   "Regular"
```

Observamos que el orden de los niveles no sigue el orden “real”. Para modificarlo usamos la función **fct_relevel()**

```
> Datos_salud$Esalud <- fct_relevel(Datos_salud$Esalud,
                                     "Muy buena", "Buena",
                                     "Regular", "Mala", "Muy mala")

> levels(Datos_salud$Esalud)
```

Ahora los niveles tienen un orden lógico que comienza en “Muy buena” salud y termina en “Muy mala”.

Observemos qué pasa cuando llamamos a la variable *Esalud*

```
> Datos_salud$Esalud

[1] Mala      Muy mala   Buena     Mala      Buena     Buena     Regular    Muy
buena

 [9] Buena     Regular    Buena     Mala      Buena     Muy buena Buena
Mala

[17] Buena     Muy mala   Regular
```



```
Levels: Muy buena < Buena < Regular < Mala < Muy mala
```

Veamos qué pasó.

El orden es el que construimos, pero entre cada uno de ellos hay un símbolo < que indica la relación: el nivel de la izquierda es menor al ubicado a la derecha. En nuestro caso esta situación es inversa (Muy buena no es peor que buena sino mejor).

¿Podemos revertir este esquema de niveles sin reescribir todo? Sí, mediante la función **fct_rev()**.

```
> Datos_salud$Esalud <- fct_rev(Datos_salud$Esalud)
```

fct_other()

Sigamos con la variable *Ciudad* pasándola a factor:

```
> Datos_salud$Ciudad <- as_factor(Datos_salud$Ciudad)
> fct_count(Datos_salud$Ciudad)
```

```
# A tibble: 4 x 2
  f          n
  <fct>    <int>
1 La Plata    16
2 Berisso     1
3 Ensenada    1
4 City Bell   1
```

Cuando estamos frente a situaciones como esta, donde hay varias categorías con poca frecuencia, es mejor agruparlas en un “otras/os”. Eso lo podemos con la función **fct_other()**. En **keep** especificamos las variables que deseamos que se conserven y en **other_level** definimos cómo queremos que se llamen las otras.



```
> Datos_salud$Ciudad <- fct_other(Datos_salud$Ciudad,  
                                keep = "La Plata",  
                                other_level = "Otras")  
  
> levels(Datos_salud$Ciudad)  
[1] "La Plata" "Otras"
```

fct_collapse()

Para finalizar convirtamos a factor la última variable: *Comorbilidades*

```
> Datos_salud$Comorbilidades<- as_factor  
                                (Datos_salud$Comorbilidades)  
  
> levels(Datos_salud$Comorbilidades)  
[1] "EPOC"          "Gastritis"      "aterosclerosis" "TBC"  
  
[5] "Neumonía"      "Hipertensión"  "Hepatitis"
```

Con la función **fct_collapse()** podemos agrupar los niveles a grupos que serían las nuevas etiquetas del nivel.

Por ejemplo, vamos a crear una nueva variable llamada "Comor_agrupada" con los niveles Respiratoria, Digestiva y Circulatoria.

```
> Datos_salud$Comor_agrupadas <-  
fct_collapse(Datos_salud$Comorbilidades,  
             Respiratoria = c("EPOC", "TBC",  
                             "Neumonía"),  
             Digestiva = c("Hepatitis", "Gastritis"),  
             Circulatorio = c("aterosclerosis",  
                              "Hipertensión"))  
  
> levels(Datos_salud$Comor_agrupadas)  
[1] "Respiratoria" "Digestiva"    "Circulatorio"
```



6

Otras funciones de forcats

Otras funciones importantes del paquete *forcats* son:

fct_drop(): elimina los niveles que no se utilizan

fct_expand(): incorpora niveles a la lista de niveles de un factor

fct_c(): concatena factores combinando niveles

7

Más funciones

Aprovechando los datos que tenemos en esta base de datos, vamos a aplicar algunas funciones que no son del paquete *forcats* pero pertenecen al universo de Tidyverse y son de mucha utilidad cuando trabajamos con factores.

Observemos la variable **VECES**, la misma contabiliza las veces por semana que la persona realiza actividad física. Si queremos ver los valores que presenta dicha variable podemos usar la función ***unique()***:

```
> unique(Datos_salud$VECES)
```

Muchas veces, en nuestro trabajo diario, nos vamos a ver obligados a generar otro dataframe seleccionando un subconjunto de datos a partir de uno original. Esto lo podemos hacer con la función ***subset()***.

Por ejemplo, si queremos generar un subconjunto al que llamaremos *salud2* con las observaciones de aquellas personas que hacen actividad física 1 a 7 veces, podemos escribir:



```
> salud2 <- subset(Datos_salud, VECES %in% 1:7)
```

Veamos qué pasa con la variable TABACO.

Ésta se encuentra codificada de la siguiente manera:

- 0 = Nunca fumó
- 1 = Fuma actualmente
- 3 = Exfumador

Podemos verificar si todos los valores se encuentran dentro de ese rango o si hay problemas de carga en alguna observación:

```
> unique(Datos_salud$TABACO)
[1] 0 1 11 2
```

El 11 es un error de carga ¿cómo lo modificarías? Contanos ...



Autor: Myrian Aguilar. Esta obra está bajo una [Licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional](https://creativecommons.org/licenses/by-nc-sa/4.0/). Mundos E.