

Introducción a R

21 de agosto de 2023

INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO (ITAM)

Seminario de Investigación Económica

Instructor: Horacio Larreguy

Asistente: Eduardo Zago

Autor: Manuel Quintero

1. Introducción a R y RStudio

De acuerdo con el sitio oficial de R: *The R Foundation* (2021). R es un lenguaje de programación de **software libre** para la computación y gráficos estadísticos. El lenguaje R incluye una amplia variedad de técnicas y pruebas estadísticas para modelos tanto lineales como no lineales.

Fortalezas de R:

- Facilidad en el manejo de datos de bases pequeñas y medianas (< 2 GB).
- Conjunto de operadores para el calculo de matrices.
- Variedad de paquetes bien documentados para el análisis estadístico.

Debilidades de R:

- Lee todo los datos a la memoria RAM desde un inicio (no los fragmenta).
- Límite aproximado de 2 a 4 GB de almacenamiento o un índice vectorial de 2 mil millones.
- Todos los objetos de R se almacenan en la memoria, hay que eliminarlos constantemente.

RStudio es un entorno de desarrollo integrado (IDE) tanto para R como para Python que cuenta con un editor, una consola, ambiente global de trabajo, visualización de gráficas en una interfaz gráfica de usuario (GUI). Además, se puede utilizar para trabajar con Markdown (RMarkdown), HTML, \LaTeX , entre otros lenguajes de programación.

Instalación: Las versiones más recientes de R y RStudio se pueden descargar desde los siguientes enlaces:

1. R: <https://cran.r-project.org/bin/windows/base/>.
2. RStudio: <https://www.rstudio.com/products/rstudio/>.

1.1. Básico de RStudio

El libro del *R Development Core Team* (2000) escrito por los desarrolladores contiene una introducción detallada al lenguaje de programación R.

R, al ser un *software* libre, gran cantidad de desarrolladores y científicos crean sus paquetes para hacer que sus investigaciones sean reproducibles; por lo general, estos paquetes contienen funciones para realizar una tarea en específica, muchos de estos paquetes son aceptados por la comunidad científica para su uso en trabajos de investigación. Estos paquetes son de gran ayuda para obtener resultados rápidamente, sin tener que repetir el trabajo ya realizado por otros. Por esto, para empezar a trabajar en R es necesario saber como instalar y cargar paquetes, así como conocer otros comandos básicos.

Limpiar el *Global Environment* o *workspace*

```
rm(list = ls()) # Limpia todo lo almacenado en el workspace
rm(list = setdiff(ls(), "x")) # Elimina todo excepto la variable x

.rs.restartR() # Reiniciar la sesión: limpia el workspace y paquetes
```

Comando para instalar un paquete o actualizarlo

```
install.packages("nombre")
# Podemos utilizar un paquete sin instalarlo de la siguiente forma,
nombreDelPaquete::funcion # Donde función es una operación dentro del paquete
nombreDelPaquete
```

Comando para desinstalar un paquete

```
remove.packages("nombre")
```

Comando para cargar un paquete y poder utilizarlo

```
library(nombre)
```

Comando para cambiar directorio de trabajo ¹

¹Véase el [Apéndice A](#) para un ejemplo de código para limpiar el directorio, instalar y cargar múlti-

```
setwd("C:/Users/nombre/Desktop/Tutorial") # establece directorio
getwd('') # string que regresa el directorio actual
```

Comando para obtener ayuda de una función específica

```
help(mean)
?mean
```

Vectores:

```
# Vectores
palabras <- c("carro", "barco", "avión")
x <- c(1,2,3,4)
y <- c(2,4,6,8)

# Aritmética vectorial
sqrt(x) # Raiz cuadrada
x + y # Suma
2*x*y # Producto
x/y # Division entrada a entrada

length(x) # Longitud del vector
sum(x) # Suma las entradas del vector
```

Sucesiones

```
# Generar sucesiones
x <- c(1:10) # Ascendente del 1 al 10, enteros
y <- c(10:1) # Descendente del 10 al 1, enteros
z <- seq(0, 10, by=.25) # Del 0 al 10, incrementos en 0.25
w <- rep(c(1,2),2) # Dos vectores c(1,2) concatenados
```

Operadores lógicos

```
x > y # TRUE si x es mayor que y
x >= y # TRUE si x es mayor o igual a y
x < y # TRUE si x es menor que y
x <= y # TRUE si x es menor o igual a y
x == y # TRUE si x es igual a y
x != y # TRUE si x diferente de y
is.na(x) # TRUE si x es NA o NaN
!is.na(x) # TRUE si x no es NA y no es NaN
x == 1 & y == 2 # Intersección
```

ples paquetes al mismo tiempo y establecer directorio de trabajo donde se encuentra el archivo.

```
x == 1 | y == 2 # Unión
```

2. Datos en R

2.1. Lectura de datos

Importación e exportación de archivos .txt ([read.table\(\)](#))

```
read.table('name.txt', header = F, sep = "|") # Importar
write.table(datos, file = '', sep = " ", row.names = F, col.names = T) # Exportar
```

Importación e exportación de archivos .xlsx o .xls (paquetes [xlsx](#) o [readxl](#))

```
library(xlsx)
read_excel('path', sheet = NULL, col_names = T, ...) # Determina automáticamente si es
              xls o xlsx
read_xlsx('path.xlsx', sheet = NULL, col_names = T, ...)
read_xls('path.xls', sheet = NULL, col_names = T, ...)

write.xlsx(data, file = '', sheetName = "name") # Exportar
```

Importación e exportación de archivos .csv ([read.csv\(\)](#))

```
read.csv('file.csv', header = T, sep = ",", encoding = 'UTF-8', ...) # Importar
write.csv(data, file = '' , row.names = F, col.names = T, fileEncoding = 'UTF-8') #
Exportar
```

Importación e exportación de archivos .dta (paquetes [haven](#) o [readstata13](#))

```
library(haven)
read_dta('file.dta', encoding = '', skip = '')
write_dta(data, path = '')

# readstata13 es mejor para nombres de variables mayores a 32 caracteres
library(readstata13)
readstata13(data, )
save.dta13(data, file = '')
```

Importación e exportación de archivos .shp o .geojson (paquete [sf](#))

```
st_read(dsn, layer, ...) # Importar
st_write(obj, dsn, layer, ...) # Exportar
```

Importación e exportación de archivos .parquet o .parquet.gz (datos grandes, comprimidos.)

```
library(arrow)
read_parquet("/path", ...) # Importar
write_parquet(obj, "/path", ...) # Exportar
```

2.2. Manejo de datos

La paquetería clásica para el manejo de datos es `dplyr` y haciendo uso del operador *Pipe*: `%>%`². El paquete `tidyverse` es el paquete madre de `dplyr`.

Generamos 2 bases de datos aleatorias

```
# Generamos números aleatorios de una distribución exponencial
datos_x <- cbind.data.frame(floor(matrix(rexp(25, rate=.1), ncol=5)), c(1:5))
names(datos_x) <- c(paste0("X", c(1:5)), "id")

# Generamos números aleatorios de una distribución normal
datos_y <- cbind.data.frame(matrix(rnorm(50), ncol=5), c(1:10))
names(datos_y) <- c(paste0("Y", c(1:5)), "id")
```

Instalamos el paquete

```
install.packages("tidyverse")
library(dplyr)
```

Distintas funciones de `dplyr` para el manejo de datos

```
# Comando para seleccionar columnas
select(datos_x, -4)

# Comando para crear nueva columna
mutate(datos_x, X8 = X1 - X2 + X3)

# Comando para eliminar renglones
slice(datos_x, -n()) # Elimina el ultimo renglón

# Comando para fusionar 2 columnas con valores NA
mutate(datos_x, NUEVA = coalesce(X6,X7))

# Comando para filtrar por ciertos valores
filter(datos_x, X6 == 1)

# Comando para renombrar columnas
rename(datos_x, EDAD = X2)
```

²Nuevas versiones de R incorporaron un nuevo pipe operator, de base R: `|>`

Utilizando el operador `%>%`.

```
# Hacer todo junto
nueva_base_x <- datos_x %>% dplyr::select(c(-4)) %>% mutate(X8 = X1 - X2 + X3) %>%
  slice(-n()) %>%
  mutate(NUEVA = coalesce(X6,X7)) %>% filter(X6 == 1) %>% rename(EDAD = X2)
```

Y de igual forma nos podría interesar aplicar cierta función a distintos tipos de variables. Una forma sería repetir el `mutate()` varias veces, sin embargo, si el número de variables es grande, esto se vuelve ineficiente e incluso inviable³. Una forma de hacerlo es usar las funciones `across()`, `where()`, `everything()`, etc.

```
# Función para sustituir 2 por 0
dummy_gen <- function(x) ifelse(x == 2, 0, x)

# Aplicarlo a ciertas variables usando sus nombres:
df %>% mutate(across(c(var1:var8, var12), dummy_gen))

# Aplicado a cierto tipo de variables, por ejemplo, numéricas:
df %>% mutate(across(where(is.numeric), dummy_gen))

# Aplicado a todas las variables en el df:
df %>% mutate(across(everything(), dummy_gen))

# Haciendo operaciones sobre la misma variable no incluidas en una función:
df %>% mutate(across(everything(), ~ifelse(.x == 2, 0, .x)))
```

³Antes se usaban las funciones `mutate_at()`, `mutate_all()`, etc.

Colapsar datos desagregados

En el análisis de datos, hay ocasiones donde nos interesa resumir ciertas variables por grupo. Por ejemplo, supongamos que tenemos observaciones de delitos para cada estado en el tiempo, digamos, para cada mes. Podemos resumir/agregar estas variables de varias formas, las más comunes son la suma (`sum()`) y la media (`mean()`). También, podemos agruparlas por estado o por años, usando la función `group_by()`

```
# Agrupar y agregar una variable de interés usando media:
df %>% group_by(estado) %>% summarise(var1, mean, na.rm = T) %>% ungroup()

# Agregando distintas variables usando across() y generando una variable n que cuente
  el número de observaciones:
df %>% group_by(estado) %>%
  summarise(across(c(var1:var5, starts_with("y")), mean, na.rm=T),
    n = n())

# Agregando distintos tipos de variables de distintas formas:
df %>% group_by(estado) %>%
  summarise(across(where(is.numeric), mean, na.rm=T)),
  across(where(is.factor), sum, na.rm = T),
  n = n())
```

Pregunta: Si tenemos 32 estados y observaciones por dos años mensuales de delitos (entonces, $n = 12 \times 2 \times 32 = 768$) y agrupamos por año y por estado, con cuántas observaciones nos quedaríamos?

Reshape

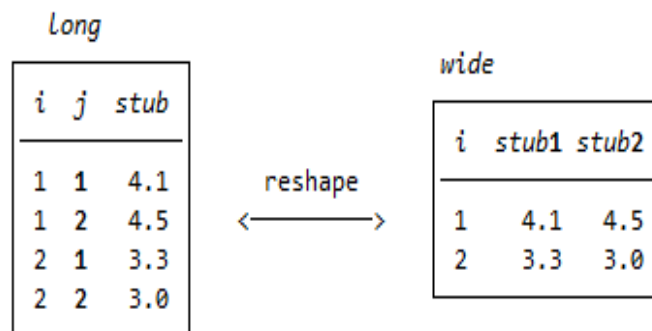
Una practica común al trabajar con bases de datos es la transformación o reordenamiento de la misma. Las operaciones más comunes son permutar renglones y columnas, resumir y transponer los datos. En R, se puede hacer uso del paquete [reshape2](#) para realizar este tipo de operaciones. Puede encontrar una explicación más detallada de las siguientes funciones en el tutorial de Datacamp: Smith ([2020](#)).

- `cbind()` y `cbind.data.frame()` son funciones que sirven para combinar vectores, matrices y bases de datos por columnas.
- `rbind()` y `rbind.data.frame()` son funciones que sirven para combinar vectores, matrices y bases de datos por renglones.
- `t()` es la función para obtener la transpuesta de un vector, una matriz o una base de datos.
- `melt()` transforma la base de formato wide a formato long.
- `dcast()` transforma la base de formato long a wide.

De estas funciones, la más interesante y probablemente las más complicadas de entender y usar son `melt()` y `cast()`⁴. Para analizar datos, hay ocasiones donde necesitas tus datos en formato long, y otras veces en formato wide. Para los fines de este curso, casi siempre necesitaremos nuestra base en formato long. Veamos la siguiente ilustración del help de Stata para ver la diferencia:

⁴Estas funciones hacen lo mismo que el `reshape wide-long` de Stata y el `pivot_wider-longer` de **tidyr**

Figura 1: Ilustración Reshape



Y podemos escribir código para este caso especial:

```
# De formato wide a long
df %>% melt(id.vars = c("i")) #ID variables son las variables que identifican
  renglones individuales de datos.

# Controlando el nombre del nuevo identificador j y del valor:
df_long <- df %>% melt(id.vars = c("i"),
  variable.name = "j",
  value.name = "stub") # en este caso la variable j tendría valores: {stub1,
  stub2}, no 1 y 2

# Ahora pasamos de long a wide:
df_wide <- df_long %>% dcast(i ~ j, value.var = "stub")
```

Fusionar bases de datos⁵

- `inner_join()`: retiene solo los renglones de x que tienen pareja en y y todas las columnas de x e y .
- `left_join()`: retiene todas las columnas de x e y y los renglones de x que no tienen pareja en y , tendrán valores NA en las nuevas columnas. Si hay múltiples parejas, todas las combinaciones se regresan.
- `right_join()`: retiene todas las columnas de x e y y los renglones de y que no tienen pareja en x , tendrán valores NA en las nuevas columnas. Si hay múltiples parejas, todas las combinaciones se regresan.
- `full_join()`: retiene todos los valores de x e y con NA donde no hay pareja.

```
inner_join(datos_x, datos_y, by = "id")
left_join(datos_x, datos_y, by = "id")
right_join(datos_x, datos_y, by = "id")
full_join(datos_x, datos_y, by = c("idx" = "idy")) # Si la variable del merge
           tiene distintos nombres
```

⁵También son útiles las funciones `se미join()`, `antijoin()` `nestjoin()`. Para ver una descripción más detallada de las siguientes funciones, véase **tidy**.

3. Análisis Gráfico y Descriptivo

Análisis Gráfico Una vez que se tiene la base de datos limpia, lo primero que se tiene que hacer antes de realizar el análisis econométrico es visualizar tus datos gráficamente y descriptivamente. Para graficar el paquete se llama **ggplot2** que es parte del **tidyverse**. Cada gráfica tiene que ser adecuada al contexto, gusto y necesidad del proyecto, por lo que la mejor opción para utilizar ggplot2 es buscar las funciones en Google.

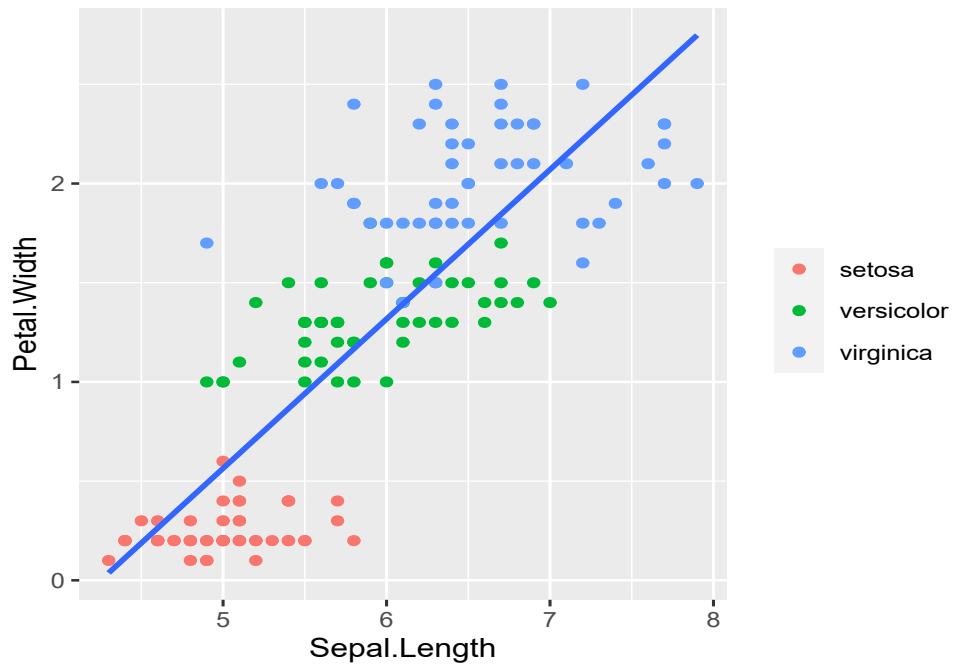
En las siguientes secciones trabajaremos con la base de datos **iris** que cuenta con 150 observaciones y 5 variables de una planta conocida como iris.

```
data(iris) # Cargamos la base de datos iris de R
names(iris) # Vemos el nombre de las variables en la base de datos
[1] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width" "Species"
```

Algunos ejemplos de gráficas y de funciones de ggplot2 para modificarlas:

```
# Scatter Plot para ver la relación entre dos variables:
g1 <- ggplot(iris, aes(Sepal.Length, Petal.Width)) +
  geom_point(aes(Sepal.Length, Petal.Width, color=factor(Species))) + # type of
  graph
  theme(legend.title = element_blank()) +
  geom_smooth(method='lm', se = FALSE) # Mejor aproximación lineal a los datos
ggsave(g1, filename = 'scatter_iris.pdf', device = cairo_pdf,
  dpi = 300, width = 12, height = 10, units = 'cm') # Salvamos en pdf
```

Figura 2: Scatter Plot



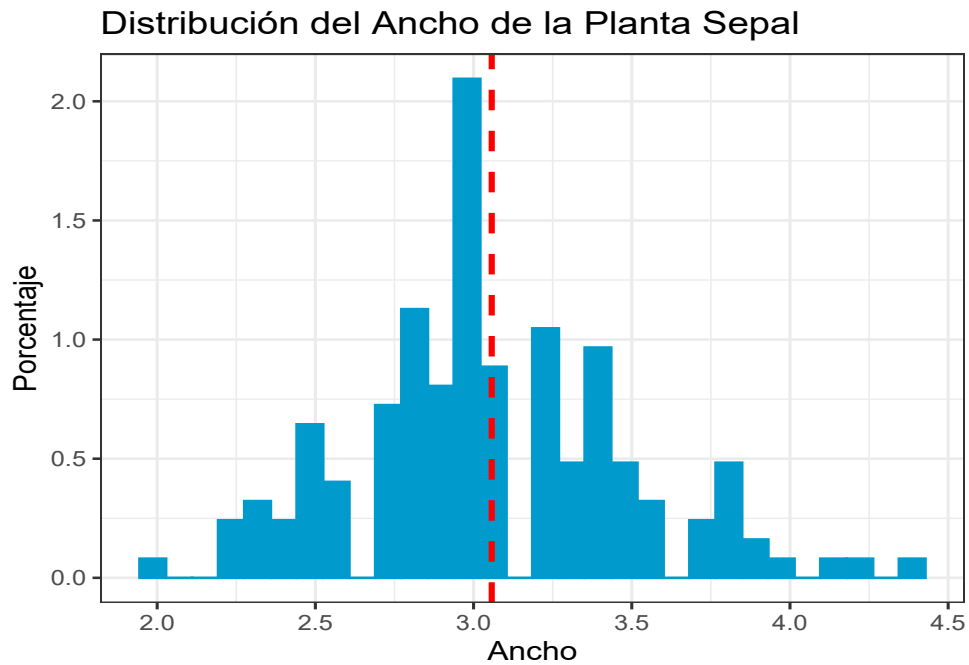
De igual forma es muy común que nos interese ver como se distribuye cierta variable, para eso utilizamos el histograma.

```
# Sacamos la media:
media <- mean(iris$Sepal.Width)

# Gráficamos:
g2 <- ggplot(iris, aes(Sepal.Width)) +
  geom_histogram(alpha = 1, aes(y = ..density..),
    position = 'identity', color="deepskyblue3",
    fill="deepskyblue3") +
  geom_vline(aes(xintercept = media), color="red",
    linetype="dashed", size=1) + # Agregamos una línea en la media.
  ggtitle("Distribución del Ancho de la Planta Sepal") +
  labs(x = "Ancho", y = "Porcentaje") + theme_bw()

ggsave(g2, filename = 'hist_iris.pdf', device = cairo_pdf,
  dpi = 300, width = 12, height = 10, units = 'cm') # Salvamos en pdf
```

Figura 3: Histograma



Como se mencionó, cada gráfica depende del contexto y de los tipos de datos. Por lo tanto, si te trabas, recomendamos visitar páginas como [sthda](#), [DataCamp](#), o buscar en Google para buscar soluciones.

Estadística Descriptiva

Es de mucha utilidad ver la distribución de nuestras variables. En muchas ocasiones nos pueden hacer más fácil la interpretación de nuestros resultados, así como la magnitud de los mismos (ej. un aumento de 2 desviaciones estandar). Con la librería **stargazer** podemos realizar tablas descriptivas fácilmente.

```
library(stargazer)

# Estadística descriptiva de toda la base:
iris %>% select(-Species) %>%
  stargazer(title="Estadísticas Descriptivas",
            summary.stat = c("n", "mean", "sd", "max", "min"),
            out="est_desc.tex")
```

Tabla 1: Estadísticas Descriptivas

Statistic	N	Mean	St. Dev.	Max	Min
Sepal.Length	150	5.843	0.828	7.900	4.300
Sepal.Width	150	3.057	0.436	4.400	2.000
Petal.Length	150	3.758	1.765	6.900	1.000
Petal.Width	150	1.199	0.762	2.500	0.100

Si quisieramos sacarlas de los tres grupos, para cada variable, podríamos usar las técnicas de agregamiento de datos que vimos la sección pasada.

```
library(kableExtra)
iris %>% group_by(Species) %>% summarise(n = n(),
  media = mean(Sepal.Length),
  de = round(sd(Sepal.Length), 3),
  min = min(Sepal.Length),
  max = max(Sepal.Length),
  mediana = median(Sepal.Length)) %>% kbl(col.names = c("Especie", "N", "Media",
  "DE", "Min.", "Max.", "Q50"),
  format = "latex",
  booktabs = T,
  caption = "Estadísticas Descriptivas Por Especie: Length Sepal")
```

Tabla 2: Estadísticas Descriptivas Por Especie: Length Sepal

Especie	N	Media	DE	Min.	Max.	Q50
setosa	50	5.006	0.352	4.3	5.8	5.0
versicolor	50	5.936	0.516	4.9	7.0	5.9
virginica	50	6.588	0.636	4.9	7.9	6.5

4. Regresión lineal

La función clásica para estimar modelos de regresión lineal en R por el método de mínimos cuadrados ordinarios es `lm()`, la cual viene del paquete **lme**.

```
lm(formula, data, exactDof = FALSE, subset, na.action, contrasts = NULL, weights = NULL, ...)
```

El argumento `formula` es una expresión de la forma $Y \sim X$, donde Y es la variable dependiente y X la matriz de variables explicativas. El segundo argumento, `data` especifica cual es la base de datos que contiene las variables que aparecen en `formula`. Conforme avance el curso iremos revisando como agregar cosas mucho más interesantes como efectos fijos, estimación por Variables Instrumentales, entre otras.

Corremos una regresión lineal simple tomando como variable dependiente el largo del sépallo de la planta y como variable explicativa el ancho del sépallo de la planta. Creamos el modelo usando el comando `lm()` y obtenemos el resumen de los resultados con la función `summary()`.

```
# Creamos un modelo de regresión lineal simple
reg_simple <- felm(Sepal.Length ~ Sepal.Width, data = iris)
# Imprimimos los resultados
summary(reg_simple)
```

Call:

```
felm(formula = Sepal.Length ~ Sepal.Width, data = iris)
```

Residuals:

Min	1Q	Median	3Q	Max
-1.5561	-0.6333	-0.1120	0.5579	2.2226

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	6.5262	0.4789	13.63	<2e-16 ***
Sepal.Width	-0.2234	0.1551	-1.44	0.152

Signif.	codes:	0	***	0.001	**	0.01	*	0.05	.	0.1	1
---------	--------	---	-----	-------	----	------	---	------	---	-----	---

Residual standard error: 0.8251 on 148 degrees of freedom

Multiple R-squared(full model): 0.01382 Adjusted R-squared: 0.007159

Multiple R-squared(proj model): 0.01382 Adjusted R-squared: 0.007159

F-statistic(full model): 2.074 on 1 and 148 DF, p-value: 0.1519

F-statistic(proj model): 2.074 on 1 and 148 DF, p-value: 0.1519

Una regresión lineal múltiple se corre de manera análoga, escribiendo el argumento formula de la siguiente manera:

$$Y \sim X_1 + X_2 + \dots + X_n,$$

Por ejemplo, explicamos el largo del sépalo por la anchura tanto del sépalo como del pétalo.

```
# Regresión lineal múltiple
reg_multiple1 <- felm(Sepal.Length ~ Sepal.Width + Petal.Width, data = iris)
reg_multiple2 <- felm(Petal.Length ~ Sepal.Width + Petal.Width + Sepal.Length, data =
  iris)
```

Supongamos que queremos introducir efectos fijos por el tipo de especie de la planta:

```
# La mejor alternativa a estos modelos es hacer uso de felm del paquete lfe.  
require(lfe)  
regresion_ef_felm <- felm(Sepal.Length ~ Sepal.Width | Species, data = iris)
```

5. De R a L^AT_EX

Cuando queremos reportar los resultados de una manera elegante y formal tenemos que exportar los resultados de la función `summary()` en una mejor presentación. En R, tenemos dos maneras de trabajar con L^AT_EX. La primera es trabajar directamente en **R Markdown**, que nos permite escribir reportes que contengan código R y compilen un documento en formato .pdf, .html o .doc interpretando correctamente comandos HTML o L^AT_EX. Usualmente, solo queremos generar el código en TeX de ciertos resultados para importarlos a un compilador de L^AT_EX. Los paquetes más utilizados para exportar resultados econométricos a tablas en código TeX son [xtable](#) y [stargazer](#). El paquete es una gran herramienta para exportar nuestros resultados ya que también soporta los objetos de la función `felm()`.

5.1. Paquete stargazer

El **stargazer** fue creado por Marek Hlavac de la universidad de Harvard y es uno de los más desarrollados para exportar resultados estadísticos de R a L^AT_EX con la función [stargazer\(\)](#). Esta función tiene una gran variedad de opciones para personalizar las tablas tanto en formato HTML como en formato TeX. La documentación del paquete se puede consultar [aquí](#).

```
# Tabla sencilla para una regresión lineal simple
stargazer(reg_simple) # Por default el resultado es código "latex"
```

Además, con **stargazer** podemos presentar múltiples regresiones en una sola tabla. Si las especificaciones comparten variables explicativas, éstas se alinean automáticamente en renglones.

También podemos especificar que estadísticos incluir, como el estadístico F, R, R ajustado, el número de observaciones entre otros, con el atributo *omit.stat*. Un atributo muy útil es *add.lines* que lo utilizamos para agregar renglones con información adicional en la parte inferior de la tabla.

```

# Dos regresiones multivariadas en una sola tabla
tabla2 <- stargazer(reg_multiple1, reg_multiple2,
  header = FALSE,
  font.size = "scriptsize",
  dep.var.labels.include = FALSE,
  table.placement = "H",
  column.labels = c("Largo del Sépalo", "Largo del pétalo"),
  covariate.labels = c("Ancho del Sépalo", "Ancho del pétalo", "Largo del pé
    talo", "Constante"),
  omit.stat = c("f", "ser", "adj.rsq"),
  title = "Ejemplo de múltiples regresiones multivariadas en una sola tabla",
  type = "latex")

# Un problema con stargazer es la forma forma de agregar las notas, este que se
  presenta aquí es una forma fácil.
note.latex <- "\\multicolumn{3}{l} {\\parbox[t]{6cm}{ \\textit{Notas:}
De esta manera podemos modificar las notas. * denota  $p < 0.1$ , ** denota  $p < 0.05$ , y **
  * denota  $p < 0.01$ .}} \\\\"
tabla2[grepl("Note", tabla2)] <- note.latex
# Imprimimos el código de LaTeX guardado en el objeto tabla2
cat(tabla2)

```

Tabla 3: Ejemplo de múltiples regresiones multivariadas en una sola tabla

	<i>Dependent variable:</i>	
	Largo del Sépalo	Largo del pétalo
	(1)	(2)
Ancho del Sépalo	0.399*** (0.091)	-0.646*** (0.068)
Ancho del pétalo	0.972*** (0.052)	1.447*** (0.068)
Largo del pétalo		0.729*** (0.058)
Constante	3.457*** (0.309)	-0.263 (0.297)
Observations	150	150
R ²	0.707	0.968

Notas: De esta manera podemos modificar las notas. * denota $p < 0.1$, ** denota $p < 0.05$, y *** denota $p < 0.01$.

6. Apéndice A

```
# Instalamos los paquete necesarios
list.of.packages <- c("stargazer", "xtable", "AER", "biostat3", "car", "ggplot2", "did",
  ", "rddtools")

# Verificamos aquellos paquetes que no han sido instalados
new.packages <- list.of.packages[!(list.of.packages %in% installed.packages()[,"
  Package"])]

# Los instalamos y cargamos todos a la vez
if(length(new.packages)) install.packages(new.packages) # verificamos que exista al
  menos un paquete en new.packages y lo instalamos
lapply(list.of.packages, library, character.only = TRUE) # Función en R que aplica la
  función library a todos los elementos en list.of.packages

# Limpiamos el workspace
rm(list = ls())

# Establecemos el directorio de trabajo al directorio de este documento con el paquete
  rstudioapi
rstudioapi::getActiveDocumentContext
setwd(dirname(rstudioapi::getActiveDocumentContext()$path))
```