

Introducción a R para biología en la Facultad de Ciencias UNAM

Leonardo Tomás Martínez Trueba - leonardotrueba@ciencias.unam.mx

Contents

1 Sobre este manual	3
1.1 Cómo usar este manual	3
1.2 Fuentes adicionales	3
1.3 Agradecimientos	3
2 Antes de empezar	4
2.1 Sobre el texto y el código	4
2.2 Librerías necesarias	4
2.3 Sobre los datos de este manual	5
2.4 Instalación de R	5
2.5 Uso de RStudio	5
3 Conceptos básicos de R	7
3.1 Introducción de información a R	7
3.2 Tipos de datos	10
3.2.1 Datos numéricos	10
3.2.2 Operadores	11
3.2.3 Datos de carácter	12
3.2.4 Datos lógicos	12
3.2.5 Datos ausentes	13
3.3 Introducción a los objetos en R	13
3.4 Funciones	14
3.5 Objetos	15
3.5.1 Vectores	15
3.5.1.1 Selección de datos de un vector	19
3.5.1.2 Funciones para crear vectores	21
3.5.1.3 Factores	24
3.5.2 Operaciones con vectores	25
3.5.3 Matrices	26
3.5.3.1 acceder y editar los valores de una matriz	27
3.5.4 Data frames	29
3.5.4.1 Construcción de data frames en R	30
3.5.4.2 Acceder y manipular datos	33
3.5.5 Tibble	35
3.5.6 Objetos con tres dimensiones: Listas, arreglos y más	37
3.6 Introducción de datos	39
3.6.1 Formato del archivo	39
3.6.2 Cargar datos a R	39
3.6.2.1 Ruta del archivo	39
3.6.2.2 Funciones para introducir datos	42
4 El paquete Tidyverse	45

4.1	dplyr	46
4.1.1	Select	46
4.1.2	Arrange	51
4.1.3	Filter	57
4.1.4	Mutate	62
4.1.5	group_by() y summarize()	64
4.1.6	Exportar data frames y tibbles	66
4.2	tidyr: datos largos y anchos	67
5	Gráficas con ggplot2	72
5.1	Primeros pasos	72
5.2	Estéticas	75
5.3	Ejes	83
5.4	Leyenda	89
5.5	Colores	93
5.5.1	Paletas	103
5.6	Temas	108
5.7	Facetting	108
5.8	Exportar gráficas	117
5.9	Gráfica de cajas y bigotes	118
5.9.1	Gráfica de violín	134
5.10	Gráficas de barras	138
5.10.0.1	Barras de error	142
5.10.1	Gráficas de barras agrupadas	148
5.11	Gráficas de dispersión	161
5.12	Gráfica de líneas	179
6	Mapas	187
6.1	Obtención de datos y creación de capas	187
6.2	Datos para un mapa de distribución de especies	193
6.3	Mapa de distribución para más de una especie	200
6.4	Modificación de la leyenda	210
6.5	Mapa de tipos de vegetación	217
7	Pruebas de hipótesis en R	232
7.1	Prueba de normalidad	232
7.2	Prueba de homocedasticidad	235
7.3	Comparación entre dos grupos	237
7.3.1	Prueba de t para grupos independientes	237
7.3.2	Prueba de Wilcoxon para grupos independientes	246
7.3.3	Prueba de t para muestras pareadas	248
7.3.4	Prueba de Wilcoxon para muestras pareadas	251
7.3.5	Prueba de t para una muestra	251
7.3.6	Prueba de Wilcoxon para una muestra	253
7.4	Comparación entre tres o más grupos	255
7.4.1	ANOVA	256
7.4.2	Prueba de kruskal Wallis	265
7.5	Prueba de xi cuadrada	270
8	Prácticas biología	272
8.1	Cuantificación de BSA mediante curva patrón	272

1 Sobre este manual

Muchas veces durante la carrera me vi en la situación de tener que procesar datos, hacer una gráfica o una prueba de hipótesis sin tener una idea clara sobre cómo debía de hacerlo. Sin duda estas inquietudes serán compartidas por muchxs otrxs ya sea para una materia, el servicio social o la tesis. El principal objetivo del manual es proveer a quien sea que lo necesite las herramientas necesarias para solucionar los problemas que se presenten en su formación como biólogox. A lo largo de mi trayectoria en la carrera pude familiarizarme con R de tal modo que constantemente buscaba hacer uso de él para lo que fuera y quisiera compartir ese conocimiento con la esperanza de que sea de utilidad para otras personas, deseando que se puedan evitar cualquier coraje o inconveniente que luego nos aquejan con todo lo relativo a estadística y también al usar R.

Los métodos aquí presentados solo son una posible solución dentro de las múltiples que hay y los temas están sesgados a mi manera de resolver los problemas. Esta no es de ninguna manera una guía completamente correcta ni definitiva y R no será la herramienta más conveniente para todo mundo.

1.1 Cómo usar este manual

Los temas comienzan desde lo más esencial sobre R y van incrementando en complejidad. Sin embargo, no por eso deben leerse de principio a fin, mi recomendación sería usarlo como una guía cuando se presente un problema que se desee solucionar. Seguramente será necesario consultar diferentes partes del manual, ya que los temas más avanzados hacen uso de los métodos explicados en secciones anteriores, pero al mismo tiempo he intentado que cada sección sea suficiente clara por sí misma, por lo cual se encontrará información repetida en algunas ocasiones.

En resumen el manual se compone de las siguientes secciones:

- Antes de empezar: algunos requisitos para usar R y este manual.
- Conceptos básicos de R: cómo usar objetos y funciones.
- El paquete tidyverse: cómo manipular, transformar y arreglar los datos en R
- Gráficas con ggplot: gráficas de barras, cajas y bigotes, dispersión y líneas.
- Mapas: dónde obtener datos y cómo usarlos para hacer mapas de distribución y de tipos de vegetación de México y sus estados.
- Pruebas de hipótesis: supuestos, funciones necesarias y gráficas.
- Prácticas de biología.

He de aceptar que mi conocimiento es limitado y cualquier retroalimentación para mejorar este manual será infinitamente apreciada. Así mismo, es claro que no se encuentran aquí todas las soluciones para todas las materias ni áreas de estudio dentro de la biología, algo que puede ser remediado poco a poco con el conocimiento de aquellxs que deseen colaborar en este proyecto.

1.2 Fuentes adicionales

Estos materiales de libre acceso han sido fundamentales en mi aprendizaje sobre R. No dudo que también podrían ser de utilidad par muchxs más:

- R for Data Science
- ggplot2:elegant graphics for data analysis
- The R Graph Gallery

1.3 Agradecimientos

Este trabajo existe de principio a fin gracias al apoyo inigualable de aquellas personas que a través de su tiempo y cariño me dieron la motivación necesaria para sentarme a escribir. A Renata porque eres parte

de mí sin importar el tiempo ni la distancia. A Kari por darme la certeza de que podía comenzar con esto y más. A Estefy por toda tu paciencia durante la carrera y tu cariño más allá de la biología. A Dany por escuchar una y otra vez mis debrayes sobre estadística y llenar mi vida de color. A Guadalupe por endulzar mi vida y acompañarme a cada momento en esta y otras travesías.

Sin más que decir, agradezco el tiempo y atención de quien sea que se encuentre en algún momento leyendo este manual.

- Leo

2 Antes de empezar

2.1 Sobre el texto y el código

El código se muestra en cuadros separados del texto y en ocasiones dentro del texto se muestran pequeños fragmentos con el formato del código para claramente señalar objetos, funciones, argumentos o elementos dentro de R y RStudio. Dichos fragmentos tienen el siguiente **formato**. Las funciones van acompañadas de dos paréntesis al final **funcion()**.

El código presentado aquí sigue algunas reglas de formato para que sea más fácil de leer:

- Los nombres de los objetos con más de una palabra son separados por un guión bajo (`_`), en lugar de colocar todas las palabras juntas, usar mayúsculas o puntos.
- Se utiliza un espacio antes y después de los operadores como `=`, `+`, `-`, `*`, `/`, `~` y `<-`. Cuando se usan comas únicamente se coloca un espacio después de la coma.
- La longitud de cada línea no excede los 80 caracteres. Se puede colocar una regla vertical para visualizar este límite en RStudio dentro del menú Tools>Global Options>Code>Display>Show Margin. Este límite es únicamente visual y es posible excederlo. Esta cantidad de caracteres es común debido a que no excede el espacio horizontal disponible en la mayor parte de los monitores. Cuando se desea iniciar una nueva línea en R, se realiza después de una coma o de algún operador para claramente indicar que los contenidos continúan en la línea inferior.

2.2 Librerías necesarias

Estas son las librerías empleadas para las diferentes secciones del manual. Deben instaladas con la función `install.packages("paquete")`.

- `tidyverse` agrupa a una serie de paquetes que utilizan la misma sintaxis y que han sido desarrollados para su uso conjunto. Principalmente se usarán `readr`(leer y escribir datos), `dplyr` (transformar data frames) y `ggplot2` (crear gráficas).
- `palatteer` agrupa paletas continuas y discretas para su uso en gráficas creadas con `ggplot2`.
- `sf` permite introducir y usar datos espaciales para la creación de mapas.
- `multcompView` provee herramientas para agrupar comparaciones realizadas por pruebas post hoc de acuerdo con los valores de p.
- `dunn.test` contiene una sola función del mismo nombre para hacer una prueba post hoc no paramétrica.

Para utilizar las funciones de cada paquete es necesario primero cargarlas con `library()` como se muestra a continuación:

```
library(tidyverse)
## -- Attaching packages -----
## v ggplot2 3.3.6      v purrr   0.3.5
## v tibble   3.1.8      v dplyr    1.0.10
## v tidyverse 1.3.2     v stringr  1.4.1
```

```

## v readr  2.1.3      vforcats 0.5.2
## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()

library(paletteer)
library(sf)

## Linking to GEOS 3.10.2, GDAL 3.4.1, PROJ 8.2.1; sf_use_s2() is TRUE

library(multcompView)
library(dunn.test)

```

A lo largo del manual las librerías también serán llamadas paquetes o paqueterías, pero los tres términos se refieren a lo mismo.

2.3 Sobre los datos de este manual

Las diferentes secciones hacen uso de datos (usualmente data frames) que provienen de los paquetes básicos de R, bases de datos o que fueron inventados para su uso en este manual sin ningún fundamento biológico. Mencione esto solo para aclarar que los siguientes archivos proporcionados junto con el manual no reflejan ningún fenómeno biológico y solo son ejemplos convenientes para su uso en los diferentes temas:

- chicharos.csv
- granos_maiz.csv
- granos_trigo.csv
- insectos.csv
- insectos_machos_hembras_sitio.csv
- lagartijas.csv
- pesos_conejos.csv
- peso_seco.csv
- pesos_seriola.csv
- semillas.csv

El resto de los datos se encuentran en R o su procedencia es indicada en las secciones en las que son utilizadas.

2.4 Instalación de R

Se puede obtener en el sitio oficial de R. Primero se debe seleccionar uno de los dos repositorios ubicados en México. Después se selecciona el sistema operativo deseado y se elige la opción de instalar R por primera vez. Una vez instalado, R puede utilizarse desde la terminal o símbolo del sistema, o también a través de la interfaz gráfica (RGui) que se descarga junto con R.

2.5 Uso de RStudio

Debido a que es un entorno de desarrollo integrado (IDE), RStudio permite trabajar más fácilmente con R. Se puede obtener en el sitio de RStudio, en la ventana de productos. La versión gratis y de código abierto es la adecuada para los propósitos de este manual.

Al abrir RStudio se muestra únicamente la consola y los paneles laterales

Para abrir un nuevo script se utiliza la pestaña en la esquina superior izquierda que abre un menú en donde se muestran los posibles documentos para crear. Alternativamente, se puede usar el atajo:Ctrl+Shift+N.

Con el script abierto se tiene la disposición que usualmente se usará para trabajar con este programa. Los paneles de la izquierda facilitan la comunicación entre el script y la consola, mientras que los paneles derechos permiten visualizar otros aspectos de la sesión de trabajo como el ambiente, historial, archivos, gráficas, paquetes, la ventana de ayuda, etc.

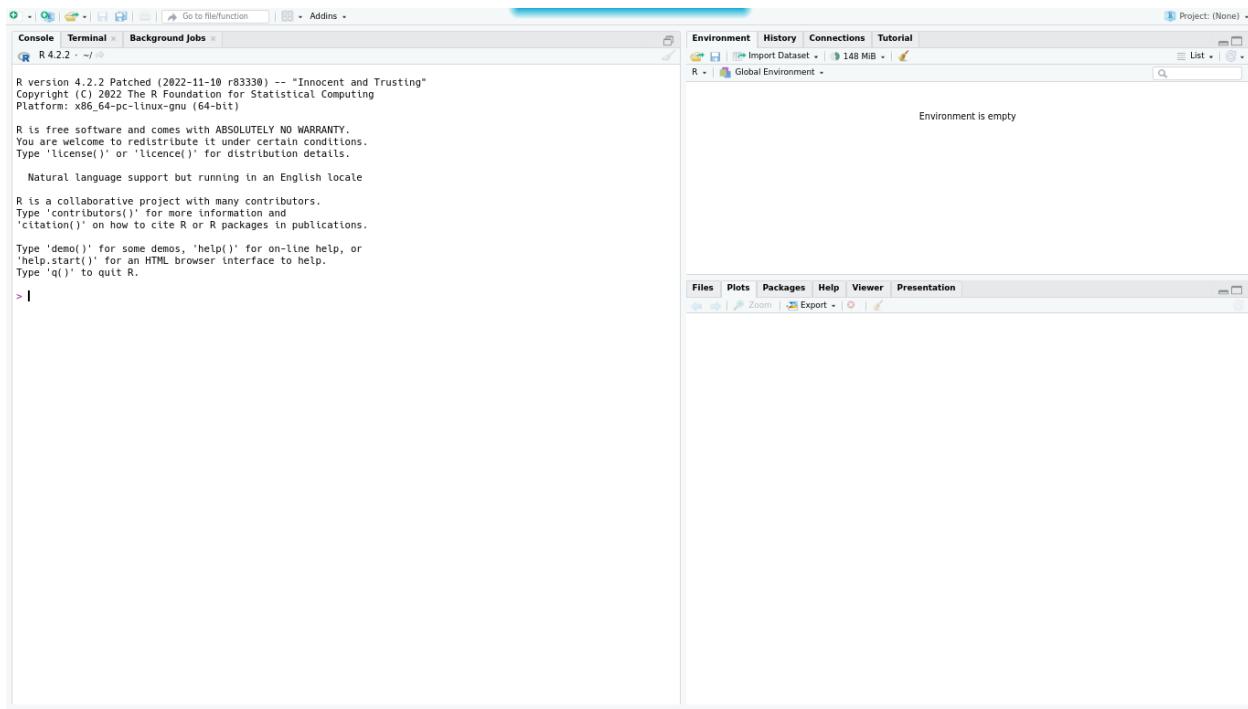


Figure 1: Vista inicial de RStudio

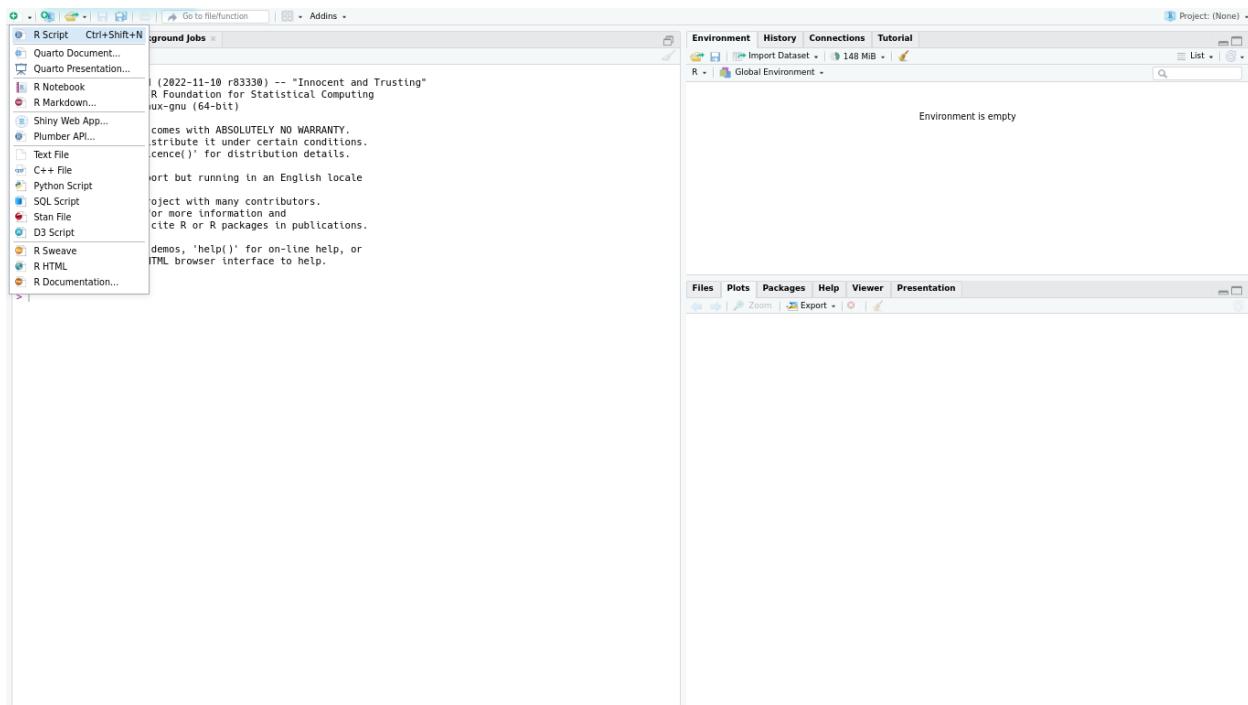


Figure 2: Abrir nuevo script

Si es necesario, el tamaño de cada panel puede modificarse arrastrando sus bordes y cada panel puede minimizarse.

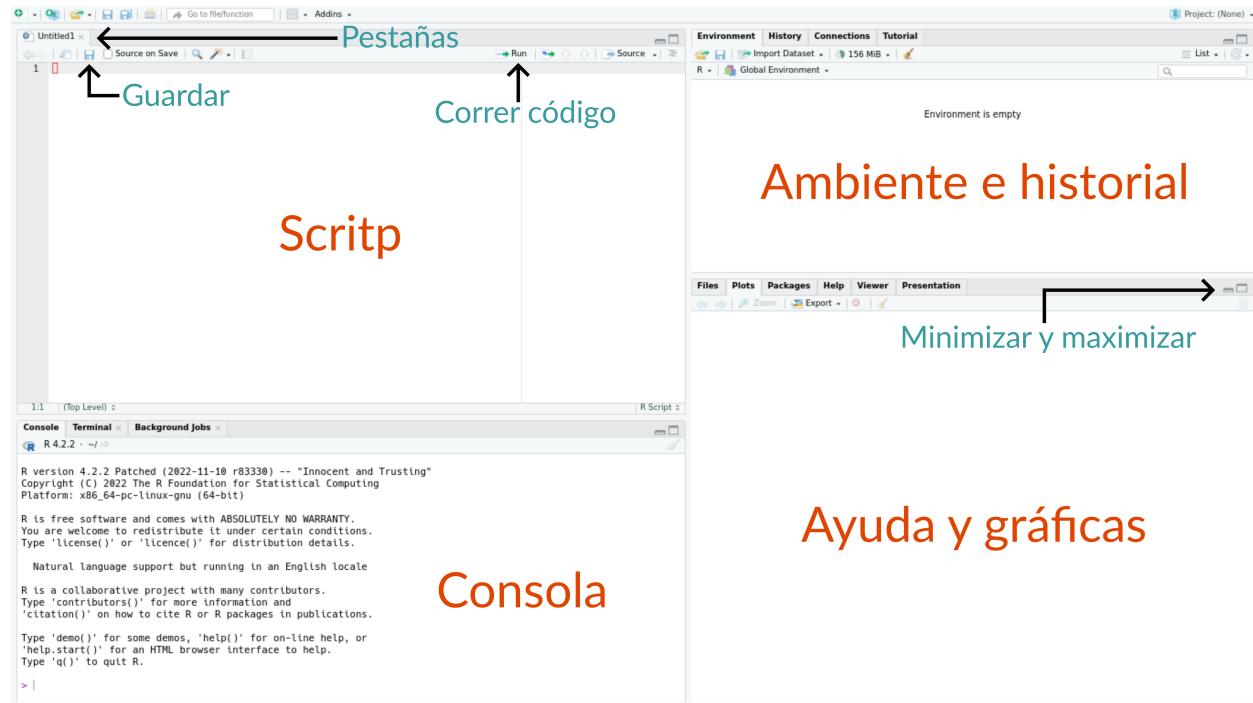


Figure 3: Principales funciones de RStudio

A lo largo del manual se describirán algunos aspectos de la programación en R de acuerdo con su funcionamiento en RStudio, pero su uso no es esencial para conseguir los resultados aquí presentados y cualquier otro editor de texto es suficiente.

3 Conceptos básicos de R

3.1 Introducción de información a R

El flujo de trabajo en R consiste en mandar instrucciones a la consola, en donde son interpretadas por el lenguaje. Estas instrucciones pueden escribirse directamente en la consola y al presionar Enter son evaluadas por R, pero la manera más conveniente de trabajar es utilizando scripts.

Usualmente las instrucciones son introducidas mediante funciones, pero para comenzar con algo mas sencillo se pueden realizar operaciones aritméticas:

```
10 * 2
## [1] 20
```

Si se escribe un número sin ninguna operación la consola solo lo repite, reconociendo que se trata de un número:

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

Si se escriben letras o palabras, serán interpretadas como nombres de objetos guardados en R o nombres de funciones. A menos de que se haya escrito un nombre que corresponda a un objeto o una función, se obtendrá un error:

```

datos
## Error in eval(expr, envir, enclos): object 'datos' not found
A
## Error in eval(expr, envir, enclos): object 'A' not found

```

Cuando se escribe el nombre de un objeto se imprime en la consola, en este caso un data frame con el crecimiento de chícharos de acuerdo con la adición de nitrógeno, fósforo y/o potasio en un diseño experimental fraccionado:

```

npk
##      block N P K yield
## 1      1 0 1 1  49.5
## 2      1 1 1 0  62.8
## 3      1 0 0 0  46.8
## 4      1 1 0 1  57.0
## 5      2 1 0 0  59.8
## 6      2 1 1 1  58.5
## 7      2 0 0 1  55.5
## 8      2 0 1 0  56.0
## 9      3 0 1 0  62.8
## 10     3 1 1 1  55.8
## 11     3 1 0 0  69.5
## 12     3 0 0 1  55.0
## 13     4 1 0 0  62.0
## 14     4 1 1 1  48.8
## 15     4 0 0 1  45.5
## 16     4 0 1 0  44.2
## 17     5 1 1 0  52.0
## 18     5 0 0 0  51.5
## 19     5 1 0 1  49.8
## 20     5 0 1 1  48.8
## 21     6 1 0 1  57.2
## 22     6 1 1 0  59.0
## 23     6 0 1 1  53.2
## 24     6 0 0 0  56.0

```

Si no se trata de un objeto, el error obtenido puede ser evitado escribiendo las letras o palabras entre comillas, y únicamente se reconocen como datos de carácter:

```

"datos"
## [1] "datos"
"A"
## [1] "A"

```

Si se escribe el nombre de una función, es necesario siempre añadir los paréntesis para indicar que se pretende usar esa función, ya que dentro se colocan las instrucciones o argumentos necesarios. Cuando no se escriben los paréntesis se imprime una descripción de la función pero no se utiliza:

```

plot
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x563dd2b2a910>
## <environment: namespace:base>

```

Es por ello que se necesitan escribir los paréntesis aún si se trata de una función sin argumentos, como es el caso de `ls()` y `getwd()`:

```
getwd
## function ()
## .Internal(getwd())
## <bytecode: 0x563dd21c16f8>
## <environment: namespace:base>

getwd()
## [1] "/home/leot/Documents/programacion/R/introduccion_R_biology"
```

Todos los comandos anteriores pueden ser escritos directamente en la consola, pero cuando se manejan funciones más largas que abarcan múltiples líneas, el uso de la consola presenta limitaciones. Al usar RStudio es más fácil escribir y editar los comandos a través de scripts y luego mandarlos a la consola presionando **Ctrl + Enter** o con el botón **Run** en la esquina superior derecha de la ventana del script. Usualmente se envían unas cuantas líneas a medida que se progresá en la tarea que se desea completar, pero también se puede evaluar un script completo con el botón **Source** a la derecha de **Run**.

Otra ventaja de usar RStudio es que los scripts cuentan con sintaxis de colores para identificar fácilmente distintos elementos. De igual manera facilita su organización al acomodar las líneas automáticamente de manera jerárquica de acuerdo con el uso de paréntesis:

```
funcion_1(argumento_1 = TRUE,
           argumento_2 = funcion(arg_1 = c(elemento_1, elemento_2,
                                           elemento_3, elemento_4,
                                           elemento_5, elemento_6),
                                 arg_2),
           argumento_3)
```

En el ejemplo anterior, el desplazamiento horizontal de cada línea está determinado por los paréntesis, ya que al abrir un nuevo par dentro de otro par, las nuevas líneas comienzan más hacia derecha cuando se presiona **Enter**, de modo que quedan alineadas con los contenidos del mismo paréntesis en el que se encuentran anidadas. La primera línea de la función establece el valor del primer argumento y después de la coma comienza una nueva línea que consiste del segundo argumento en donde se requiere una segunda función con sus propios argumentos. Esta segunda función establece el segundo nivel en la jerarquía, y sus argumentos (`arg_1` y `arg_2`) son fácilmente identificables porque tienen la misma posición horizontal a pesar de que se encuentran en diferentes líneas. `arg_1` necesita especificar un vector, por lo cual se abre un nuevo par de paréntesis y con ello un nuevo nivel en la jerarquía. Al listar los elementos y comenzar nuevas líneas, estos quedan alineados más hacia la derecha. Una vez que se cierran los paréntesis del vector y de la segunda función, la nueva línea regresa a la alineación correspondiente a los argumentos de la función inicial.

Como se puede ver, el uso de paréntesis puede ser algo conflictivo si es que se pierde en cuenta su posición y número. Cuando se utilizan menos, no se cierran adecuadamente todos los componentes del comando y eso ocasiona un error.

```
mean(c(1, 2, 3, 4)
## Error: <text>:2:0: unexpected end of input
## 1: mean(c(1, 2, 3, 4)
##          ^
```

Del mismo modo, si se utilizan más de los necesarios, también ocurre un error:

```
mean(c(1, 2, 3, 4)))
## Error: <text>:1:20: unexpected ')'
## 1: mean(c(1, 2, 3, 4)))
##          ^
```

Los mensajes obtenidos no indican que se trata de un paréntesis ausente o extra y se pueden obtener errores similares por ejemplo si falta una coma o un signo de igual:

```
mean(c(1, 2, 3, 4) na.rm = T)  
## Error: <text>:1:20: unexpected symbol  
## 1: mean(c(1, 2, 3, 4) na.rm  
##          ^  
  
mean(c(1, 2, 3, 4), na.rm T)  
## Error: <text>:1:27: unexpected symbol  
## 1: mean(c(1, 2, 3, 4), na.rm T  
##          ^
```

En caso del paquete `tidyverse`, estos errores son más claros cuando hay paréntesis de más:

```
library(tidyverse)  
filter(.data = iris, Species == "setosa")  
  
## Error: <text>:2:42: unexpected ')'  
## 1: library(tidyverse)  
## 2: filter(.data = iris, Species == "setosa")  
##          ^
```

Pero el mensaje de error no es tan claro cuando los paréntesis no se cierran correctamente:

```
filter(.data = iris, Species == "setosa"  
  
## Error: <text>:3:0: unexpected end of input  
## 1:  
## 2: filter(.data = iris, Species == "setosa"  
##          ^
```

Usualmente los errores se solucionan revisando el código y colocando los pequeños detalles ausentes pero que son esenciales para que funcione. En el caso de los paréntesis, RStudio resalta el paréntesis que abre y cierra al colocar el puntero sobre cualquiera de los dos para fácilmente identificarlos. Una ayuda adicional son los paréntesis arcoíris, los cuales colorean cada par de paréntesis en un color ligeramente distinto. Esta opción se encuentra en el menú de Herramientas>Opciones Globales>Código>Display.

3.2 Tipos de datos

Para introducir información y describirla es necesario usar distintos tipos de datos. En R estos son numéricos, de carácter y lógicos. Usando esta clasificación se pueden emplear datos numéricos continuos, discontinuos, categóricos ordenados y no ordenados.

3.2.1 Datos numéricos

Sencillamente, son números. Se dividen en dos, los números enteros (`integers`) que no tienen posiciones decimales, y los números continuos (`numeric` o `double`). En ocasiones, es necesario tener en cuenta esta distinción debido a que algunas funciones aceptan únicamente datos numéricos enteros o continuos. Para verificar la identidad de un tipo de dato se usa la función `class()`:

```
class(10)  
## [1] "numeric"  
  
class(45.24)  
## [1] "numeric"
```

De igual manera se pueden utilizar `is.integer()` e `is.numeric` las cuales dan como resultado un valor falso o verdadero.

```
is.integer(1.5)
## [1] FALSE
is.numeric(1.5)
## [1] TRUE
```

Por defecto, todos los valores introducidos son interpretados como números continuos, aún si no tienen posiciones decimales. Para que R reconozca un valor como un número entero debe ser establecido como tal con la función `as.integer()`:

```
class(1)
## [1] "numeric"
class(as.integer(1))
## [1] "integer"
```

Aquellas funciones que requieran el uso de números enteros suelen realizar la conversión automáticamente, pero en caso de que la conversión tenga que realizarse manualmente se puede emplear el método descrito. Otra manera de indicar que se tratan de números enteros es colocando una L al final:

```
class(67L)
## [1] "integer"
```

3.2.2 Operadores

Con los datos numéricos pueden realizarse operaciones aritméticas:

```
3 + 5
## [1] 8
34 * 12
## [1] 408
```

Los símbolos de suma (+), resta (-), división (/) y multiplicación (*) son llamados operadores en R. Existen otros operadores, como el de potencia (^) y una variedad de operadores lógicos para establecer comparaciones entre dos elementos:

- & (AND)
- | (OR)
- ! (NOT)
- == (igual a)
- != (no es igual a)
- > (mayor que)
- >= (mayor o igual que)
- < (menor que)
- <= (menor o igual que)

3.2.3 Datos de carácter

Son aquellos que usualmente utilizamos para introducir información en forma de palabras o letras. Para que R los reconozca adecuadamente es necesario escribirlos entre comillas ("").

```
class("Leo")
## [1] "character"
class(Leo)
## Error in eval(expr, envir, enclos): object 'Leo' not found
```

La función anterior indica un error debido a que no se utilizaron comillas, por lo que R interpreta la cadena de caracteres (string) como el nombre de un objeto en el ambiente global, y al no encontrarlo presenta el error.

Los datos de carácter también pueden estar conformados por números, solamente deben de ser introducidos con comillas para que no sean interpretados como datos numéricos:

```
class("3")
## [1] "character"
```

Los datos de carácter pueden estar conformados por múltiples palabras:

```
class("Esto es una oración.")
## [1] "character"
```

3.2.4 Datos lógicos

Son aquellos que solo pueden tomar dos valores, verdadero (TRUE) o falso (FALSE) y también pueden indicarse solamente con las letras T y F. Este tipo de datos usualmente se utiliza para establecer argumentos dentro de una función, realizar comparaciones mediante los operadores lógicos y para interpretar resultados o mensajes de error.

```
class(F)
## [1] "logical"
class(TRUE)
## [1] "logical"
```

Es necesario escribir el valor con letras mayúsculas, de lo contrario no será reconocido adecuadamente:

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

Las comparaciones realizadas con operadores lógicos dan como resultado un valor falso o verdadero, al igual que todas las funciones utilizadas para corroborar el tipo de dato u objeto:

```
7 > 4
## [1] TRUE
is.vector(1)
## [1] TRUE
```

3.2.5 Datos ausentes

Se representan con NA (not available), son un tipo de dato lógico que indica la ausencia de un valor.

```
class(NA)  
## [1] "logical"
```

3.3 Introducción a los objetos en R

La programación con R involucra el uso de diferentes tipos de objetos, los cuales organizan los datos de diferente manera para su uso. Cada uno será explicado con mayor detalle en secciones posteriores, pero es útil contar con definiciones preliminares para comprender otros aspectos de la programación. Los objetos son vectores, matrices, data frames y listas.

Los vectores agrupan un conjunto de datos, siempre y cuando todos correspondan al mismo tipo (numérico, de carácter o lógico). Pueden tener cero (vector vacío), uno o más elementos los cuales están separados por comas.

```
c(2, 4, 6, 8, 10)  
## [1] 2 4 6 8 10  
c("Adrián", "Fernando", "David")  
## [1] "Adrián"    "Fernando"  "David"  
c(TRUE, FALSE, FALSE, TRUE)  
## [1] TRUE FALSE FALSE TRUE
```

Las matrices son vectores organizados en dos dimensiones, por lo cual solo agrupan datos de la misma categoría, pero pueden ser organizadas de diferentes maneras de acuerdo con el número de filas y columnas. Para los temas de este manual las matrices no son de gran relevancia.

```
matrix(data = 1:10, nrow = 2)  
##      [,1] [,2] [,3] [,4] [,5]  
## [1,]     1     3     5     7     9  
## [2,]     2     4     6     8    10  
  
matrix(data = 1:10, ncol = 2)  
##      [,1] [,2]  
## [1,]     1     6  
## [2,]     2     7  
## [3,]     3     8  
## [4,]     4     9  
## [5,]     5    10
```

Los data frames. Son estructuras rectangulares, es decir que cada una de sus filas y columnas debe tener la misma longitud. En R, cada columna representa un vector diferente, por lo que cada una agrupa a un solo tipo de datos. En conjunto, la agrupación de vectores en un data frame permite representar el valor de diferentes variables (columnas) para cada observación (filas):

```
head(iris)  
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 1          5.1       3.5        1.4       0.2  setosa  
## 2          4.9       3.0        1.4       0.2  setosa  
## 3          4.7       3.2        1.3       0.2  setosa  
## 4          4.6       3.1        1.5       0.2  setosa  
## 5          5.0       3.6        1.4       0.2  setosa
```

```
## 6      5.4      3.9      1.7      0.4  setosa
```

El data frame `iris` está incluido con los paquetes básicos de R, las primeras cuatro columnas indican la longitud y ancho de sépalos y tépalos. Estas cuatro variables son vectores numéricos, mientras que la última columna indica la especie a la que pertenecen las observaciones mediante un vector de carácter. De esta manera, las filas agrupan a todas las observaciones, es decir, cada una de las flores medidas y las columnas muestran los valores de cada variable que le corresponden a cada flor.

Las listas y arreglos son objetos que agrupan a cualquier tipo de objeto, incluidas otras listas. `iris3` es un arreglo que cuenta con los mismos datos de `iris`, pero separados en tres data frames, uno para cada especie:

```
head(iris3)

## , , Setosa
##
##      Sepal L. Sepal W. Petal L. Petal W.
## [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
##
## , , Versicolor
##
##      Sepal L. Sepal W. Petal L. Petal W.
## [1,]    7.0    3.2    4.7    1.4
## [2,]    6.4    3.2    4.5    1.5
## [3,]    6.9    3.1    4.9    1.5
## [4,]    5.5    2.3    4.0    1.3
## [5,]    6.5    2.8    4.6    1.5
## [6,]    5.7    2.8    4.5    1.3
##
## , , Virginica
##
##      Sepal L. Sepal W. Petal L. Petal W.
## [1,]    6.3    3.3    6.0    2.5
## [2,]    5.8    2.7    5.1    1.9
## [3,]    7.1    3.0    5.9    2.1
## [4,]    6.3    2.9    5.6    1.8
## [5,]    6.5    3.0    5.8    2.2
## [6,]    7.6    3.0    6.6    2.1
```

3.4 Funciones

Son instrucciones que le damos a R para introducir, crear, manipular y transformar datos. Siempre siguen la estructura `nombre_de_la_funcion(argumento_1 = VALOR, argumento_2 = VALOR)`. Dentro de los paréntesis se escriben los argumentos, los cuales funcionan como parámetros o instrucciones adicionales para la función y son indicadas con el signo `=`. Usualmente el primer argumento indica el objeto sobre el cual se va a aplicar la función.

```
mean(x = 1:5, na.rm = T)
## [1] 3
```

En este caso, la función para obtener la media aritmética `mean()` es aplicada sobre la secuencia de números del 1 al 5 (la cual es un vector). El segundo argumento `na.rm` indica que se deben ignorar los datos ausentes (`NA`), aunque en este caso no hay ninguno. El nombre de los argumentos y a qué se refieren puede cambiar

en distintas funciones. En este caso `x` representa el objeto sobre el cual se aplica la función, pero es una denominación arbitraria y en otras funciones puede cambiar a `X`, `.data`, `data`, etc. Esta información, junto con el resto de los argumentos, puede ser consultada en la documentación de cada función, la cual puede accederse con la función `help(nombre_de_la_función)`, o bien `?nombre_de_la_función`.

```
?mean
```

Los argumentos pueden ser indicados sin escribir sus nombres. Esto se debe a que cada uno de los valores indicados son interpretados de acuerdo con el orden en el que los argumentos son presentados en la documentación. Es por ello que la expresión `mean(1:5, 0, T)` da el mismo resultado que al indicar `x` y `na.rm`.

```
mean(1:5, 0, T)
```

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

Sin embargo, si se omite el valor 0 que corresponde al argumento `trim`, se obtiene un error:

```
mean(1:5, T)
```

```
## Error in mean.default(1:5, T): 'trim' must be numeric of length one
```

Para corregir el error es necesario indicar que el valor `T` corresponde a `na.rm`. Mientras no se utilice el mismo orden establecido en la documentación deben indicarse el nombre de cada uno de los argumentos. Afortunadamente, algunas funciones y familias de funciones comparte una estructura similar en cuanto al orden de los argumentos. En diferentes textos y en línea, el código usualmente se encuentra sin los nombres de los argumentos ya que es una manera más eficiente de programar. Sin embargo, para facilitar el aprendizaje de los argumentos de cada función y poder entender nuestro propio código tiempo después de haberlo escrito es recomendable escribir los argumentos, al menos en etapas iniciales de nuestro uso en R.

Además del orden de los argumentos debe considerarse que las funciones tienen valores por defecto, los cuales serán aplicados si es que no se indica algún otro valor de manera explícita. En el caso de la función `mean()` los argumentos `trim` y `na.rm` tienen los valores 0 y `FALSE`, respectivamente. Al usar la función indicando únicamente `x` se están usando los valores por defecto de los otros dos argumentos.

```
mean(x = 1:5)
```

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

Después de mandar una función a la consola se obtiene un resultado, el cual puede ser cualquiera de los objetos usados en R. El resultado de `mean()` corresponde a un vector que contiene un solo elemento que corresponde al promedio. Los objetos obtenidos pueden ser utilizados para operaciones adicionales, por lo cual es útil guardar el resultado de las funciones como objetos para su posterior uso.

```
media_1 <- mean(x = 1:5)
```

3.5 Objetos

Como se mencionó anteriormente, los principales objetos en R son los vectores, matrices, data frames y listas. En esta sección se dará una descripción más detallada sobre cada uno de ellos así como su creación y manipulación en R.

3.5.1 Vectores

Son objetos que agrupan a un conjunto de elementos que pertenecen al mismo tipo de datos. Es por ello que existen vectores numéricos, de carácter o lógicos. Para crear un vector usualmente se utiliza la función `c()` (combinar o concatenar), cuyos argumentos representan los elementos individuales que forman el vector resultante y son separados por comas. Por ejemplo, un vector numérico con cinco números:

```
c(1, 2, 3, 5, 7)
```

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

En este ejemplo, el comando únicamente imprime el vector en la consola. Si se desea emplear el vector en funciones más complejas, puede resultar problemático escribirlo de nuevo una o más veces. Lo más conveniente es guardarla como un objeto identificable a través de un nombre, el cual se escribe al inicio de la línea, seguido del operador de asignación (`<-`) (menor que y signo de menos) y después la función que da origen al vector:

```
nombre <- funcion()
```

De esta manera, el vector `c(1, 2, 3, 5, 7)` puede ser guardado en el ambiente global y accedido a través del nombre asignado

```
num_1 <- c(1, 2, 3, 5, 7)
```

La única operación realizada aquí es la asignación, por lo que el vector no se imprime en la consola. El vector debe aparecer junto con su nombre en la pestaña **Environment** en el panel de la esquina superior derecha, bajo el apartado **Values**. De igual manera se indica el tipo de vector (numérico) y los primeros valores que lo conforman. En la medida en que la consola no muestre un error, la asignación se completó adecuadamente.

Los objetos pueden ser consultados en la pestaña **Environment** o pueden mostrarse en la consola mediante la función `ls()`, la cual no tiene ningún argumento, pero es necesario aún así escribir los paréntesis.

```
ls()
```

```
## [1] "media_1" "num_1"
```

Si se desea borrar alguno de los objetos se escribe dentro de la función `rm()` (remove):

```
remove(num_1)
```

```
ls()
```

```
## [1] "media_1"
```

Es importante recordar que la asignación de los nombres es arbitraria, cualquier combinación de caracteres puede emplearse mientras que no comiencen con un número o que tengan espacios. Los siguientes dos vectores no son válidos:

```
1vector <- c(1, 10, 30)
## Error: <text>:1:2: unexpected symbol
## 1: 1vector
##           ^
vector 1 <- c(1, 10, 30)
## Error: <text>:1:8: unexpected numeric constant
## 1: vector 1
##           ^
```

Los vectores pueden contener números siempre que no se encuentren en la primera posición y para separar las palabras que conforman el nombre del objeto se puede usar guión bajo `_` o punto `:`:

```
vector1 <- c(1, 3, 5, 7)
vector1
## [1] 1 3 5 7
vector_1 <- c(2, 4, 6, 8)
vector_1
## [1] 2 4 6 8
```

Para acceder a los datos de cada vector se debe usar el nombre de caracteres tal cual como fue inicialmente especificado, tomando también en cuenta la posición de letras mayúsculas o minúsculas, debido a que la misma letra en mayúscula representa un carácter diferente a la misma letra en minúscula, y por lo tanto un nombre diferente:

```

vector_1
## [1] 2 4 6 8
Vector_1
## Error in eval(expr, envir, enclos): object 'Vector_1' not found

```

Debido a que R acepta cualquier combinación de caracteres como el nombre de un objeto (incluso letras individuales), es importante nombrarlos de manera que sea fácilmente recordarlos, así como sus contenidos. Sin embargo, deben evitarse aquellas palabras que representan nombres de funciones o valores predeterminados en R:

- if
- else
- repeat
- while
- function
- in
- next
- break
- TRUE
- T
- FALSE
- F
- NULL
- Inf
- NaN
- NA
- c

Finalmente, es importante mencionar que la asignación puede realizarse también a través de `->`(signo de menos y signo de mayor que) y de `=`. El uso de `->` únicamente implica que la asignación ocurre en la dirección opuesta a `<-`:

```

objeto_1 <- c(10, 20, 30)
objeto_1
## [1] 10 20 30
c(15, 30, 45) -> objeto_2
objeto_2
## [1] 15 30 45

```

El uso de `=` no es recomendable debido a que no indica la dirección en la que ocurre la asignación, además de que su uso es más relevante para establecer los argumentos en una función, y es recomendable limitar su uso para este propósito. Si es que se desea comparar dos datos se utiliza el operador lógico de igualdad `==`.

R cuenta con vectores que se cargan por defecto en cada sesión y pueden ser accedidos mediante su nombre a pesar de que no se encuentren en el ambiente global. Los objetos LETTERS y letters son vectores de carácter en los que cada elemento corresponde a cada una de las letras del alfabeto.

```
LETTERS
```

```
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"  
## [20] "T" "U" "V" "W" "X" "Y" "Z"  
  
letters  
  
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"  
## [20] "t" "u" "v" "w" "x" "y" "z"
```

A diferencia de los vectores numéricos, los vectores de carácter pueden agrupar letras y números, así como cualquier otro carácter. Cada uno de sus elementos está separado por comillas y para construirlos se utiliza también la función `c()`:

```
c("A", "B", "C")  
  
## [1] "A" "B" "C"
```

Los vectores de carácter pueden agrupar elementos que contengan más de un carácter, y no todos los caracteres tienen que ser de la misma longitud. La única consideración a tomar en cuenta es que todo lo que se encuentre entre un par de comillas va a formar parte de un solo elemento, y cada elemento debe estar separado por una coma.

```
c("A", "apis", "picea", "Arabidopsis")  
  
## [1] "A"           "apis"         "picea"        "Arabidopsis"
```

Los vectores de carácter también pueden contener números, ya sea solos o junto con letras y otros números:

```
c("1", "1568", "1h", "10_m", "salvia")  
  
## [1] "1"          "1568"        "1h"          "10_m"        "salvia"
```

Un elemento puede contener cualquier carácter, incluidas comas, ya que mientras se encuentren dentro de las comillas son consideradas como elementos de uno de los valores del vector:

```
c("Incluso, esto podría ser una oración entera", "FALSE", "T",  
  "c() no cuenta como una función al estar entre comillas", "+", "10-5")  
  
## [1] "Incluso, esto podría ser una oración entera"  
## [2] "FALSE"  
## [3] "T"  
## [4] "c() no cuenta como una función al estar entre comillas"  
## [5] "+"  
## [6] "10-5"
```

En caso de que un vector sea construido como elementos por fuera de comillas, R lo interpretará como un objeto que se encuentra guardado en el ambiente global y mostrará un error si es que no existe un objeto con tal nombre:

```
c("1", "cuatro", datos)  
  
## Error in eval(expr, envir, enclos): object 'datos' not found
```

Sin embargo, un vector si puede ser construido a partir de vectores preexistentes:

```
c(LETTERS, letters)  
  
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"  
## [20] "T" "U" "V" "W" "X" "Y" "Z" "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"  
## [39] "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

Si se mezclan números que no están entre comillas, se realiza automáticamente una conversión para que todos los elementos sean de carácter:

```
c(1, 10, "4")
## [1] "1"  "10" "4"
```

Lo mismo ocurre cuando se mezclan números sin comillas con letras que si lo están:

```
c("letras", 4)
## [1] "letras" "4"
```

De esta manera el vector siempre cuenta con datos del mismo tipo.

3.5.1.1 Selección de datos de un vector Cuando los vectores se imprimen en la consola, cada línea que es impresa comienza con un número entre corchetes [<#)]. Esta sintaxis indica la posición que corresponde al elemento del inicio de cada línea, de modo que al imprimirse el vector, este comienza con [1]. Esto indica que los elementos de los vectores están ordenados y siempre serán impresos de acuerdo con ese orden, aquellos vectores creados con c() siguen el orden en el cual fueron escritos. Usando esta sintaxis se pueden acceder elementos individuales de acuerdo con su posición dentro del vector. Primero se coloca el nombre del vector al que se va a acceder y luego entre corchetes la posición deseada:

```
LETTERS[10]
```

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

Como resultado se obtiene un vector con los elementos seleccionados, en este caso solamente uno. Esta operación no ocasiona ninguna modificación al vector original, por lo que no se está extrayendo el dato del vector original, solo se crea uno nuevo con los datos indicados. Debido a esto, la letra J aún forma parte del vector LETTERS:

```
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

Para consultar más de un elemento se tiene que usar un vector numérico dentro de los corchetes, el contiene las posiciones que se van a consultar:

```
letters[c(1, 3, 6)]
## [1] "a" "c" "f"
letters[c(5, 3, 10, 2)]
## [1] "e" "c" "j" "b"
```

Los valores tienen que estar especificados dentro de un vector ya que una como por fuera de la función c() indicaría una posición en una columna, las cuales no existen en los vectores:

```
letters[1, 3]
## Error in letters[1, 3]: incorrect number of dimensions
```

El vector introducido también puede ser a través de un nombre previamente asignado:

```
posiciones <- c(1, 3, 5, 7, 9, 11, 13, 15)
letters[posiciones]
## [1] "a" "c" "e" "g" "i" "k" "m" "o"
```

Usando los corchetes se puede crear un subconjunto de los valores deseados creando un objeto separado. Por ejemplo, para separar las vocales del alfabeto:

```
vocales <- LETTERS[c(1, 5, 9, 15, 21)]
vocales
## [1] "A" "E" "I" "O" "U"
```

Los corchetes permiten no solamente ver sino también modificar los valores dentro de un vector. Primero se indica la posición del vector que se desea modificar y se indica el nuevo valor con =:

```
vocales[1] = "W"  
vocales  
## [1] "W" "E" "I" "O" "U"
```

Al imprimir el objeto se aprecia que el primer elemento ha cambiado, por lo que al realizar esta operación el vector original si es modificado. En caso de que se deseé recuperar el vector original se puede emplear el mismo procedimiento, o se puede correr el comando que inicialmente dio origen al vector.

A través de este método se pueden cambiar múltiples elementos usando un vector numérico y usando un vector con los nuevos elementos en el orden en que se desean introducir de acuerdo con las posiciones previamente especificadas:

```
vocales[c(1, 3, 5)] = c("Z", "X", "Y")  
vocales  
## [1] "Z" "E" "X" "O" "Y"
```

Usando los corchetes también se pueden agregar nuevos valores, siempre y cuando se coloquen al final del vector:

```
vocales[c(6, 7)] = c("G", "J")  
vocales  
## [1] "Z" "E" "X" "O" "Y" "G" "J"
```

Esta manera de interactuar con los vectores no es esencial para los temas desarrollados en el manual. Si bien algún uso podría surgir en su momento, el principal objetivo de explicar estos métodos consiste en proveer suficiente información sobre la sintaxis que podría aparecer en los resultados de los comandos, mensajes de error y también en scripts que no sean los propios.

Un ultimo detalle acerca de los vectores es que cada uno de sus elementos puede tener un nombre, los cuales se asignan con la función `names()` de la siguiente manera:

```
alturas <- c(1.56, 1.78, 1.60, 1.85)  
alturas  
## [1] 1.56 1.78 1.60 1.85  
  
names(alturas) <- c("Arturo", "José", "Brenda", "Jocelyn")  
alturas  
## Arturo     José    Brenda Jocelyn  
##   1.56     1.78     1.60     1.85
```

Al crear el vector, sus valores son identificados únicamente a través de su posición, pero una vez que se asignan los nombres estos aparecen por encima de cada valor. Es importante tener en cuenta que los nombres se asignarán de acuerdo con su posición en el vector. Si se usan menos nombres que elementos en el vector inicial, los datos que quedan sin nombre aparecen como `NA` (not available).

```
pesos <- c(58, 82, 64, 49)  
pesos  
## [1] 58 82 64 49  
  
names(pesos) <- c("José", "Victor")  
pesos  
##     José Victor  <NA>  <NA>  
##      58       82      64      49
```

La función `names()` también puede usarse para acceder únicamente a los nombres:

```
names(alturas)  
## [1] "Arturo"  "José"     "Brenda"   "Jocelyn"  
  
names(pesos)  
## [1] "José"    "Victor"   NA         NA
```

Los vectores con nombres no son muy comunes y su apariencia es similar a la de una tabla. En caso de que exista confusión al respecto, su identidad puede corroborarse con `class()`:

```
class(alturas)  
## [1] "numeric"
```

3.5.1.2 Funciones para crear vectores Los vectores usados hasta el momento han sido creados manualmente, indicando cada uno de sus elementos, pero también es posible usar funciones para crearlos.

Empezando con los vectores numéricos, cuando se desea usar una secuencia de números enteros consecutivos se utiliza la sintaxis `primer_número:último_número` la cual crea un vector con todos los dígitos comprendidos entre ambos números. Este método no requiere el uso de `c()`:

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

De esta manera se pueden crear fácilmente vectores de gran tamaño:

```
250:350  
## [1] 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267  
## [19] 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285  
## [37] 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303  
## [55] 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321  
## [73] 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339  
## [91] 340 341 342 343 344 345 346 347 348 349 350
```

El uso de esta sintaxis dentro de `c()` permite crear vectores que mezclan números enteros consecutivos junto con cualquier otro valor numérico:

```
c(1.3, 2, 4:20, 60:70)  
## [1] 1.3 2.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0 13.0 14.0 15.0 16.0  
## [16] 17.0 18.0 19.0 20.0 60.0 61.0 62.0 63.0 64.0 65.0 66.0 67.0 68.0 69.0 70.0
```

Para crear una secuencia de números no consecutivos pero con una separación constante entre cada dígito se utiliza la función `seq()`, indicando los argumentos del número inicial `from`, número final `to` y la separación entre cada elemento de la secuencia `by`.

```
seq(from = 12, to = 36, by = 2)  
## [1] 12 14 16 18 20 22 24 26 28 30 32 34 36
```

La separación puede ser también a través de posiciones decimales:

```
seq(from = 10, to = 11, by = .1)  
## [1] 10.0 10.1 10.2 10.3 10.4 10.5 10.6 10.7 10.8 10.9 11.0
```

Con vectores de gran tamaño, su visualización puede ser problemática debido a que R imprime hasta 1000 posiciones de un vector. Para limitar la cantidad de elementos mostrados se utiliza la función `head()`, cuyo argumento es el vector que se desea visualizar o la función que da origen al vector:

```
head(seq(from = 0, to = 50, by = .1))
```

```
## [1] 0.0 0.1 0.2 0.3 0.4 0.5
```

Al tener funciones anidadas dentro de otras pueden ocurrir errores debido al número de paréntesis, por lo cual se puede guardar el vector como un objeto y luego usar su nombre como argumento dentro de la función:

```
secuencia <- seq(from = 0, to = 50, by = .1)
head(secuencia)
```

```
## [1] 0.0 0.1 0.2 0.3 0.4 0.5
```

Por defecto, `head()` muestra únicamente las primeras 6 posiciones del vector especificado. Para mostrar una cantidad diferente de elementos se debe modificar el argumento `n`:

```
head(x = secuencia, n = 10)
## [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
head(x = secuencia, n = 3)
## [1] 0.0 0.1 0.2
```

Los vectores de carácter pueden ser creados con la función `paste()`, la cual permite construir un vector de acuerdo con un patrón establecido. Por ejemplo, una serie de tratamientos pueden ser identificados por número: tratamiento_1, tratamiento_2, tratamiento_3, etc. Construir un vector con 10 o más tratamientos podría resultar un poco engorroso, el uso de `paste()` permite realizarlo más fácilmente y sin la posibilidad de cometer un error al escribir cada elemento de manera individual. Como primer argumento se escribe el término o términos que identificarán el patrón, mientras que el segundo es un vector que contiene los caracteres que serán combinados con el primer argumento para crear nombres diferentes. Finalmente, se indica el carácter que se usará para separar ambas partes del término con el argumento `sep`:

```
paste("tratamiento", 1:10, sep = "_")
## [1] "tratamiento_1"  "tratamiento_2"  "tratamiento_3"  "tratamiento_4"
## [5] "tratamiento_5"  "tratamiento_6"  "tratamiento_7"  "tratamiento_8"
## [9] "tratamiento_9"  "tratamiento_10"
```

Los primeros dos argumentos carecen de nombre, pero deben ir al inicio de la función. En la documentación únicamente están especificados con `...`, que representa una forma de denotar argumentos que son identificados a través del nombre del valor que toman. Lo importante a tener en cuenta sobre esta función es que el argumento de mayor tamaño es el que determina la longitud del vector resultante. De igual manera es importante mencionar que los datos del vector numérico son convertidos a datos de carácter al correr la función.

Los nombres pueden también ser delimitados con letras o con palabras y el separador puede ser cualquier carácter, incluido un espacio ():

```
paste("tratamiento", LETTERS[1:10], sep = " ")
## [1] "tratamiento A" "tratamiento B" "tratamiento C" "tratamiento D"
## [5] "tratamiento E" "tratamiento F" "tratamiento G" "tratamiento H"
## [9] "tratamiento I" "tratamiento J"
```

Este método podría servir para crear una lista de especies que pertenecen al mismo género:

```
especies_bursera <- c("bipinnata", "glabrifolia", "copallifera",
                      "citronella", "cuneata", "palmeri",
                      "penicillata", "excelsa", "graveolens",
                      "laxiflora", "tomentosa", "linanoe",
                      "microphylla", "vejar-vazquezii",
                      "ariensis", "fagaroides", "simaruba")
```

```

bursera <- paste("Bursera", especies_bursera, sep = "-")
bursera

## [1] "Bursera-bipinnata"      "Bursera-glabrifolia"
## [3] "Bursera-copallifera"    "Bursera-citronella"
## [5] "Bursera-cuneata"        "Bursera-palmeri"
## [7] "Bursera-penicillata"    "Bursera-excelsa"
## [9] "Bursera-graveolens"     "Bursera-laxiflora"
## [11] "Bursera-tomentosa"      "Bursera-linanoe"
## [13] "Bursera-microphylla"   "Bursera-vejar-vazquezii"
## [15] "Bursera-ariensis"       "Bursera-fagaroides"
## [17] "Bursera-simaruba"

```

Para crear un vector que incluya especies de más de un genero se puede usar el mismo método para distintos géneros y combinar ambos vectores:

```

especies_protium <- c("aidanianum", "macrocarpum", "macrophyllum",
                      "morii", "nervosum", "opacum")

protium <- paste("Protium", especies_protium, sep = "-")
protium

## [1] "Protium-aidanianum"    "Protium-macrocarpum"  "Protium-macrophyllum"
## [4] "Protium-morii"          "Protium-nervosum"    "Protium-opacum"

c(bursera, protium)

## [1] "Bursera-bipinnata"      "Bursera-glabrifolia"
## [3] "Bursera-copallifera"    "Bursera-citronella"
## [5] "Bursera-cuneata"        "Bursera-palmeri"
## [7] "Bursera-penicillata"    "Bursera-excelsa"
## [9] "Bursera-graveolens"     "Bursera-laxiflora"
## [11] "Bursera-tomentosa"      "Bursera-linanoe"
## [13] "Bursera-microphylla"   "Bursera-vejar-vazquezii"
## [15] "Bursera-ariensis"       "Bursera-fagaroides"
## [17] "Bursera-simaruba"       "Protium-aidanianum"
## [19] "Protium-macrocarpum"    "Protium-macrophyllum"
## [21] "Protium-morii"          "Protium-nervosum"
## [23] "Protium-opacum"

```

Otra función que permite crear vectores de carácter es `rep()`, la cual únicamente repite un elemento o un vector el número de veces indicado. Sus argumentos son `x` que indica el elemento a repetir, `times` e `each`.

```

rep(x = "isla_1", times = 10)

## [1] "isla_1" "isla_1" "isla_1" "isla_1" "isla_1" "isla_1" "isla_1" "isla_1"
## [9] "isla_1" "isla_1"

rep(x = c("times_1", "times_2"), times = 10)

## [1] "times_1" "times_2" "times_1" "times_2" "times_1" "times_2" "times_1"
## [8] "times_2" "times_1" "times_2" "times_1" "times_2" "times_1" "times_2"
## [15] "times_1" "times_2" "times_1" "times_2" "times_1" "times_2"

```

Cuando se utilizan vectores con más de un elemento, `times` indica el número de veces que se repite el vector completamente, mientras que `each` establece cuántas veces se repetirá cada elemento dentro del vector antes de que comiencen las repeticiones del siguiente elemento:

```
rep(x = c("isla_1", "isla_2", "isla_3"), each = 4)
```

```
## [1] "isla_1" "isla_1" "isla_1" "isla_1" "isla_2" "isla_2" "isla_2"
## [9] "isla_3" "isla_3" "isla_3" "isla_3"
```

Cuando se especifican ambos argumentos, primero se realizarán las repeticiones de cada elemento y luego este patrón será repetido el número de veces indicado por `times`. De este modo, la longitud total del vector resultante será el múltiplo de los valores indicados por `times`, `each` y la longitud del vector inicial:

```
islas <- rep(x = c("isla_1", "isla_2", "isla_3"), times = 2, each = 4)
islas

## [1] "isla_1" "isla_1" "isla_1" "isla_2" "isla_2" "isla_2"
## [9] "isla_3" "isla_3" "isla_3" "isla_3" "isla_1" "isla_1" "isla_1"
## [17] "isla_2" "isla_2" "isla_2" "isla_2" "isla_3" "isla_3" "isla_3"

length(islas)

## [1] 24
```

Los patrones que se pueden crear con `rep()` son útiles cuando se desea establecer el orden de datos de acuerdo con una cierta clasificación. Por ejemplo, con un experimento en el que los datos numéricos obtenidos para cada tratamiento corresponden a cuatro réplicas:

```
rep(x = paste("tratamiento", 1:5, sep = "_"),
     each = 4)

## [1] "tratamiento_1" "tratamiento_1" "tratamiento_1" "tratamiento_1"
## [5] "tratamiento_2" "tratamiento_2" "tratamiento_2" "tratamiento_2"
## [9] "tratamiento_3" "tratamiento_3" "tratamiento_3" "tratamiento_3"
## [13] "tratamiento_4" "tratamiento_4" "tratamiento_4" "tratamiento_4"
## [17] "tratamiento_5" "tratamiento_5" "tratamiento_5" "tratamiento_5"
```

Posteriormente, cada elemento del vector es utilizado en la creación de un data frame para correctamente identificar cada valor numérico.

3.5.1.3 Factores Son vectores de carácter que describen un conjunto de categorías únicas que describen los datos. Por ejemplo, en `iris`, la variable `Species` corresponde a un factor. Un vector con factores puede construirse indicando solamente las categorías, es decir, los valores únicos que se van a repetir múltiples veces de acuerdo con lo que sea necesario para describir cada observación. Para indicar esta estructura se emplea la función `factor()`, en donde se indican las categorías mediante el argumento `x` y si se tratan de categorías ordenadas o no:

```
factores_1 <- factor(x = c("sitio_1", "sitio_2", "sitio_3", "sitio_4"),
                      ordered = F)
factores_1

## [1] sitio_1 sitio_2 sitio_3 sitio_4
## Levels: sitio_1 sitio_2 sitio_3 sitio_4

class(factores_1)

## [1] "factor"
```

Debido a que no existe ninguna clasificación jerárquica en los factores (sitios) establecidos, el argumento `ordered` es falso. Si se tuviera un conjunto de datos en el que se estudian plantas y estas se dividen en plántula, adulta y senescente si se trataría de una organización ordenada, debido a que los nombres indican categorías en orden creciente.

```
factor(x = c("plantula", "adulta", "senescente"),
       ordered = T)

## [1] plantula    adulta      senescente
## Levels: adulta < plantula < senescente
```

En este caso se observa que los niveles (categorías ordenadas) están en el orden incorrecto, debido a que R ha interpretado que la palabra `adulta` es una categoría superior a `plantula`. Para corregir el error se tiene que indicar el orden en el argumento `levels`, asegurando que sean escritos en orden creciente:

```
factor(x = c("plantula", "adulta", "senescente"),
       levels = c("plantula", "adulta", "senescente"), ordered = T)
```

```
## [1] plantula    adulta      senescente
## Levels: plantula < adulta < senescente
```

Al usar `factor()`, el argumento `x` puede contener un vector con las categorías, aún si se encuentran repetidas. Sin embargo, esto no es adecuado para `levels`, el cual interpreta el orden en el que se escriben como el orden ascendente.

```
sitios <- rep(paste("sitio", LETTERS[1:3], sep = "_"), times = 3, each = 4)
sitios

## [1] "sitio_A" "sitio_A" "sitio_A" "sitio_B" "sitio_B" "sitio_B"
## [8] "sitio_B" "sitio_C" "sitio_C" "sitio_C" "sitio_A" "sitio_A"
## [15] "sitio_A" "sitio_A" "sitio_B" "sitio_B" "sitio_B" "sitio_C"
## [22] "sitio_C" "sitio_C" "sitio_C" "sitio_A" "sitio_A" "sitio_A"
## [29] "sitio_B" "sitio_B" "sitio_B" "sitio_C" "sitio_C" "sitio_C"
## [36] "sitio_C"

factor(x = sitios, ordered = F)

## [1] sitio_A sitio_A sitio_A sitio_B sitio_B sitio_B sitio_C
## [10] sitio_C sitio_C sitio_A sitio_A sitio_A sitio_B sitio_B
## [19] sitio_B sitio_B sitio_C sitio_C sitio_C sitio_A sitio_A sitio_A
## [28] sitio_A sitio_B sitio_B sitio_B sitio_C sitio_C sitio_C sitio_C
## Levels: sitio_A sitio_B sitio_C
```

Si se desean conocer los valores únicos de un vector (aún si no está especificado como un factor) se utiliza la función `unique()`:

```
unique(sitios)

## [1] "sitio_A" "sitio_B" "sitio_C"
```

De esa manera se pueden conocer rápidamente las categorías que podrían indicarse como niveles, aunque tal vez tenga que modificarse el orden mostrado por `unique()`. Esta función también es útil cuando se tienen que conocer las categorías de un vector de gran tamaño, en donde no pueden ser distinguidas fácilmente.

3.5.2 Operaciones con vectores

Como se ha visto en ejemplos anteriores, los vectores pueden ser utilizados como argumentos en otras funciones:

```
alturas <- c(1.55, 1.6, 1.73, 1.65)
mean(x = alturas)

## [1] 1.6325
```

Esta no es la única manera en la que los vectores pueden ser utilizados, también se pueden realizar operaciones aritméticas con ellos. Si se deseara duplicar el valor de cada uno de los elementos contenidos en el vector `alturas` se podría realizar de esta manera:

```
alturas * 2

## [1] 3.10 3.20 3.46 3.30
```

En esta expresión se aplica una de las propiedades de R llamada reciclaje, en lugar de usar un vector de longitud equivalente cuyos elementos son el número 2 para indicar que cada elemento `alturas` debe ser

multiplicado por dos, se utiliza solamente un dígito: 2. De esta manera R interpreta que ese valor debe ser reciclado para cada uno de los elementos del otro vector empleado. Utilizando esta propiedad es posible hacer más cortas las operaciones en lugar de crear un vector de igual longitud:

```
alturas * c(2, 2, 2, 2)  
## [1] 3.10 3.20 3.46 3.30
```

En caso de que se desee multiplicar cada elemento por un valor diferente si es necesario especificar cada uno de ellos y asegurarse que su posición corresponda con el valor que se desea multiplicar:

```
alturas * c(1, 5, 10, 3)  
## [1] 1.55 8.00 17.30 4.95
```

En caso de que se utilice un vector con más de un elemento pero cuya longitud es menor a la del vector que se está multiplicando, el vector de menor tamaño también será reciclado hasta cubrir la longitud del vector más largo:

```
alturas * c(5, 10)  
## [1] 7.75 16.00 8.65 16.50
```

Después de agotar la longitud del vector que contiene 5 y 10, la multiplicación regresa al primer elemento para continuar con la operación. Este es el mismo comportamiento, por ejemplo, para un vector con tres elementos:

```
alturas * c(10, 5, 3)  
## Warning in alturas * c(10, 5, 3): longer object length is not a multiple of  
## shorter object length  
## [1] 15.50 8.00 5.19 16.50
```

Para este caso R muestra una advertencia, pero aún así lleva a cabo la multiplicación. La propiedad de reciclaje puede ser utilizada para todas las operaciones aritméticas y otra tipo de acciones realizadas en R. Si bien presenta ventajas, se pueden obtener resultados inesperados si no se tiene en cuenta su funcionamiento.

Los vectores pueden incluir datos ausentes (NA). Cualquier operación que se realice con datos ausentes tendrá como resultado un dato ausente:

```
c(1, 3, 5, NA) * 7  
## [1] 7 21 35 NA  
c(1, 3, 5, NA) * NA  
## [1] NA NA NA NA
```

En caso de que se utilice una función que involucre el uso de operaciones aritméticas, el argumento `na.rm` (NA remove) elimina todos los datos ausentes para que el resultado final no sea NA:

```
mean(c(1, 3, 5, NA))  
## [1] NA  
mean(c(1, 3, 5, NA), na.rm = TRUE)  
## [1] 3
```

3.5.3 Matrices

Son estructuras rectangulares (todas las filas tienen la misma longitud y todas las columnas también respecto a las otras columnas) que agrupan elementos de la misma categoría de datos. De este modo son similares a los vectores, salvo que las matrices pueden organizarse en filas y columnas.

Se construyen mediante la función `matrix()`, el argumento `data` corresponde a los datos que formarán la matriz. Debido a que las matrices son vectores organizados en dos dimensiones, este primer argumento puede ser un vector o cualquier función que crea un vector. Los otros dos argumentos son `ncol` y `nrow`, los cuales se utilizan para establecer el número de columnas o filas que tendrá la matriz. Es posible construirla únicamente indicando el número de filas o de columnas y con base en ello se calcularán automáticamente el número correspondiente de filas o columnas:

```
matrix(data = 1:10, nrow = 5)

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

matrix(data = 1:10, ncol = 5)

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

Por defecto, la función acomoda los valores por columnas, de arriba a abajo, y hacia la derecha una vez que se ha completado cada columna. El número de columnas es determinado por el argumento indicado en la función. Si la longitud del vector no es divisible por el número indicado de columnas o filas, la matriz se completa reciclando el vector indicado en `data` para completar una estructura rectangular:

```
matrix(data = 1:13, ncol = 5)

## Warning in matrix(data = 1:13, ncol = 5): data length [13] is not a sub-multiple
## or multiple of the number of rows [3]

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

La manera en que se llenan cada una de las posiciones puede ser cambiada con el argumento `byrow`, el cual por defecto es falso (`FALSE`). Si su valor se cambia a verdadero (`TRUE` o `T`), la matriz será llenada primero completando las filas de izquierda a derecha y de arriba a abajo:

```
matrix(1:20, nrow = 4, byrow = TRUE)

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
## [4,]   16   17   18   19   20
```

3.5.3.1 acceder y editar los valores de una matriz Al imprimir una matriz en la consola, los bordes muestran números entre corchetes separados por una coma. Esta sintaxis denota las posiciones dentro de la matriz primero indicando las filas y luego las columnas: `[fila, columna]`. De esta manera se pueden recuperar valores individuales dentro de ella. Para acceder a los valores es necesario escribir el nombre de la matriz y luego los corchetes indicando el valor que se desea acceder.

```
matriz_a <- matrix(1:20, nrow = 5 )
matriz_a

##      [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
```

```

## [2,]   2   7  12  17
## [3,]   3   8  13  18
## [4,]   4   9  14  19
## [5,]   5  10  15  20

matriz_a[1,3]
## [1] 11

```

En este caso el valor de la primera fila y la tercera columna corresponde al número 11. El mismo resultado puede obtenerse colocando la función que da origen a la matriz seguida de los corchetes:

```

matrix(1:20, nrow = 5)[1,3]
## [1] 11

```

Sin embargo, es más fácil trabajar con el nombre del objeto para evitar escribir el nombre de la función una y otra vez si es que se desea acceder a más de un valor en repetidas ocasiones.

Es posible acceder a más de un elemento a la vez usando vectores. En el siguiente ejemplo se busca recuperar los datos de la tercera fila para las columnas 1, 3 y 4:

```

matriz_a[3, c(1, 3, 4)]
## [1] 3 13 18

```

Es necesario usar un vector para seleccionar las columnas para respetar la estructura dentro de los corchetes, en los cuales se separan filas y columnas mediante la coma.

Las filas también pueden ser indicadas a través de un vector construido con : para una secuencia de números consecutivos, como serían las columnas 2, 3, y 4 de la fila 2:

```

matriz_a[2, 2:4]
## [1] 7 12 17

```

Los vectores también pueden usarse para acceder a más de una fila. Para acceder a la tercera columna de las filas 1, 2 y 3 se escribe el siguiente comando:

```

matriz_a[1:3, 3]
## [1] 11 12 13

```

Es importante destacar que el resultado de estos comandos es un vector con el número de elementos que se consultaron y no cuentan con la organización observada en la matriz. Para recuperar ese orden es necesario regresar al objeto original.

```

is.vector(matriz_a[1:3, 3])
## [1] TRUE

```

Sin embargo, al acceder a más de una columna y a más de una fila se obtiene una matriz nueva con los valores deseados.

```

matriz_a[c(1,2), 2:4]
##      [,1] [,2] [,3]
## [1,]    6   11   16
## [2,]    7   12   17

class(matriz_a[c(1,2), 2:4])
## [1] "matrix" "array"

```

Finalmente, para acceder a todos los valores de una fila o una columna se indica únicamente el número de fila o columna a consultar y el espacio antes o después de la coma se deja vacío. Para consultar todos los elementos de la segunda fila:

```
matriz_a[2,]  
## [1] 2 7 12 17
```

Para mostrar todas las filas de la cuarta columna:

```
matriz_a[,4]  
## [1] 16 17 18 19 20
```

Así mismo se puede acceder a todos los elementos de más de una fila o columna usando vectores:

```
matriz_a[c(2,4),]  
##      [,1] [,2] [,3] [,4]  
## [1,]     2     7    12    17  
## [2,]     4     9    14    19  
  
matriz_a[,1:3]  
##      [,1] [,2] [,3]  
## [1,]     1     6    11  
## [2,]     2     7    12  
## [3,]     3     8    13  
## [4,]     4     9    14  
## [5,]     5    10    15
```

Las matrices no son tan comúnmente usadas para introducir y manipular datos. Sin embargo, algunas de sus características son útiles para explicar el funcionamiento de R en secciones posteriores.

3.5.4 Data frames

También son estructuras rectangulares conformadas por filas y columnas. La principal diferencia con respecto a las matrices es que los data frames pueden agrupar diferentes tipos de datos en cada una de sus columnas. De este modo, cada columna corresponde a un vector diferente y un data frame puede agrupar datos numéricos, lógicos y de carácter. A través de este arreglo las filas representan observaciones individuales y las columnas muestran a las distintas variables (numéricas o categóricas) para cada una de las observaciones.

Los data frames son los objetos que más se usarán en este manual para introducir, describir, analizar y transformar datos. Cualquier información que se encuentre en una tabla puede ser introducida como un data frame si es que cuenta con la estructura correcta (filas = observaciones, columnas = variables). Esta estructura puede ser observada en el data frame `chickwts`, en donde las variables son el peso y tipo de alimento, mientras que las filas describen cada uno de los individuos pesados y sometidos a los tratamientos:

```
head(chickwts)  
##   weight      feed  
## 1    179 horsebean  
## 2    160 horsebean  
## 3    136 horsebean  
## 4    227 horsebean  
## 5    217 horsebean  
## 6    168 horsebean
```

En el data frame `iris` las variables numéricas son longitud y ancho de sépalos y pétalos. Una variable categórica indica la especie de cada una de las flores estudiadas:

```
head(iris)
```

```

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

```

No importa que las variables categóricas parezcan repetitivas, lo esencial es correctamente identificar cada observación a través de sus variables. De esta manera, cada variable cuenta con su propia columna al igual que cada observación su propia fila y los valores se encuentran en celdas individuales, lo cual facilita el uso de los datos en R.

3.5.4.1 Construcción de data frames en R Usualmente los datos que van a ser analizados no se construyen en R, pero estos ejercicios permiten conocer la estructura de los data frames para facilitar su uso posteriormente.

Los argumentos de la función `data.frame()` son los vectores que van a formar cada una de las columnas, de tal modo que cada argumento es un vector previamente guardado en el ambiente global o una función cuyo resultado es un vector:

```

pinguinoss = paste("Eudyptula", 1:10, sep = "_")
data.frame(pinguinoss,
           altura = c(30.7, 28.2, 33.5, 32, 31.3,
                      32.5, 31.4, 29.7, 35, 32.2),
           peso = c(2.4, 2.7, 3.1, 2.2, 2.7,
                   2.3, 2.5, 2, 2.2, 2.8))

##      pinguinoss altura peso
## 1    Eudyptula_1   30.7  2.4
## 2    Eudyptula_2   28.2  2.7
## 3    Eudyptula_3   33.5  3.1
## 4    Eudyptula_4   32.0  2.2
## 5    Eudyptula_5   31.3  2.7
## 6    Eudyptula_6   32.5  2.3
## 7    Eudyptula_7   31.4  2.5
## 8    Eudyptula_8   29.7  2.0
## 9    Eudyptula_9   35.0  2.2
## 10   Eudyptula_10  32.2  2.8

```

Al usar un vector del ambiente global, su nombre será el mismo de la columna, por lo que no se debe colocar `=` para asignar el vector. En el caso de `altura` y `peso`, esos serán los nombres de los vectores que se están creando dentro de `data.frame()` y se mostrarán como los nombres de las columnas. Si no se colocan estos nombres, la cadena de caracteres que da origen al vector utilizado en cada columna será usado como nombre. En este ejemplo, el nombre de la segunda columna corresponde a `altura = c(30.7, 28.2, 33.5, 32, 31.3, 32.5, 31.4, 29.7, 35, 32.2)` y es un caso similar para el último vector:

```

data.frame(pinguinoss,
           c(30.7, 28.2, 33.5, 32, 31.3,
              32.5, 31.4, 29.7, 35, 32.2),
           c(2.4, 2.7, 3.1, 2.2, 2.7,
              2.3, 2.5, 2, 2.2, 2.8))

##      pinguinoss c.30.7..28.2..33.5..32..31.3..32.5..31.4..29.7..35..32.2.
## 1    Eudyptula_1                         30.7
## 2    Eudyptula_2                         28.2
## 3    Eudyptula_3                         33.5

```

```

## 4   Eudyptula_4                      32.0
## 5   Eudyptula_5                      31.3
## 6   Eudyptula_6                      32.5
## 7   Eudyptula_7                      31.4
## 8   Eudyptula_8                      29.7
## 9   Eudyptula_9                      35.0
## 10  Eudyptula_10                     32.2
##   c.2.4..2.7..3.1..2.2..2.7..2.3..2.5..2..2.2..2.8.
## 1                           2.4
## 2                           2.7
## 3                           3.1
## 4                           2.2
## 5                           2.7
## 6                           2.3
## 7                           2.5
## 8                           2.0
## 9                           2.2
## 10                          2.8

```

Los nombres pueden ser consultados con `colnames()`:

```

a <- data.frame(pinguinos,
                 c(30.7, 28.2, 33.5, 32, 31.3,
                   32.5, 31.4, 29.7, 35, 32.2),
                 c(2.4, 2.7, 3.1, 2.2, 2.7,
                   2.3, 2.5, 2, 2.2, 2.8))
colnames(a)

## [1] "pinguinos"
## [2] "c.30.7..28.2..33.5..32..31.3..32.5..31.4..29.7..35..32.2."
## [3] "c.2.4..2.7..3.1..2.2..2.7..2.3..2.5..2..2.2..2.8."

```

Estos nombres resultan imprácticos ya que hacen complicada la visualización de los datos. Por ello es conveniente colocar el nombre de cada columna antes de especificar el vector, o nombrarlo previamente y usar el nombre del objeto como argumento.

```

genero <- c("Apis", "Puma", "Commelina", "Picea")
riqueza <- c(1500, 12, 150, 50)

b <- data.frame(genero, riqueza)
b

##      genero riqueza
## 1      Apis    1500
## 2      Puma     12
## 3 Commelina    150
## 4      Picea     50

colnames(b)

## [1] "genero"   "riqueza"

```

Al igual que los vectores y las matrices, los data frames pueden ser guardados en el ambiente global mediante el operador de asignación (`<-`) para su posterior uso.

```

registro_pingus <- data.frame(pinguinos,
                                 altura = c(30.7, 28.2, 33.5, 32, 31.3,
                                           32.5, 31.4, 29.7, 35, 32.2),
                                 peso = c(2.4, 2.7, 3.1, 2.2, 2.7,
                                         2.3, 2.5, 2, 2.2, 2.8))

```

```
2.3, 2.5, 2, 2.2, 2.8))
```

```
diversidad <- data.frame(genero, riqueza)
```

Los nombres deben de mostrarse en la ventana **Environment** en la esquina superior derecha bajo el apartado **Data**, en donde se muestran el número de observaciones (filas) y el número de variables (columnas).

Al mandar el nombre del data frame a la consola se imprimen todas las filas, lo cual puede ser problemático si es que se trata de un data frame muy largo. Usualmente solo se busca observar los nombres de las columnas y los primeros datos para tener una idea general de su estructura. Esto puede hacer con la función **head()** y como primer argumento el nombre del data frame que se desea visualizar:

```
head(registro_pingus)

##      pinguinos altura peso
## 1 Eudyptula_1   30.7  2.4
## 2 Eudyptula_2   28.2  2.7
## 3 Eudyptula_3   33.5  3.1
## 4 Eudyptula_4   32.0  2.2
## 5 Eudyptula_5   31.3  2.7
## 6 Eudyptula_6   32.5  2.3
```

De esta manera se muestran únicamente los primeros 6 valores, esto se puede modificar con el argumento **n**, indicando el número de columnas que se desea ver:

```
head(registro_pingus, n = 9)

##      pinguinos altura peso
## 1 Eudyptula_1   30.7  2.4
## 2 Eudyptula_2   28.2  2.7
## 3 Eudyptula_3   33.5  3.1
## 4 Eudyptula_4   32.0  2.2
## 5 Eudyptula_5   31.3  2.7
## 6 Eudyptula_6   32.5  2.3
## 7 Eudyptula_7   31.4  2.5
## 8 Eudyptula_8   29.7  2.0
## 9 Eudyptula_9   35.0  2.2
```

Para acceder a los últimos valores del data frame se utiliza la función **tail()**:

```
tail(registro_pingus)

##      pinguinos altura peso
## 5 Eudyptula_5   31.3  2.7
## 6 Eudyptula_6   32.5  2.3
## 7 Eudyptula_7   31.4  2.5
## 8 Eudyptula_8   29.7  2.0
## 9 Eudyptula_9   35.0  2.2
## 10 Eudyptula_10 32.2  2.8

tail(registro_pingus, n = 2)

##      pinguinos altura peso
## 9 Eudyptula_9   35.0  2.2
## 10 Eudyptula_10 32.2  2.8
```

En RStudio se puede abrir una ventana nueva que muestre únicamente el data frame para navegarlo con mayor facilidad, de manera similar a una tabla en Excel. En la ventana **Environment** únicamente se hace clic sobre el nombre del data frame que se desea ver. Esto mismo puede lograrse en la consola a través de la función **View()**:

```
View(iris)
```

Otra manera de conocer la estructura general de un data frame es usando la función `str()` (estructura):

```
str(registro_pingus)
```

```
## 'data.frame': 10 obs. of 3 variables:  
## $ pinguinos: chr "Eudyptula_1" "Eudyptula_2" "Eudyptula_3" "Eudyptula_4" ...  
## $ altura   : num 30.7 28.2 33.5 32 31.3 32.5 31.4 29.7 35 32.2  
## $ peso     : num 2.4 2.7 3.1 2.2 2.7 2.3 2.5 2 2.2 2.8
```

El resultado indica que el objeto es un data frame con 10 observaciones (filas) y 3 variables (columnas). La siguientes líneas comienzan con un \$, el cual indica se está describiendo el contenido de cada columna. Primero se muestra el nombre de la columna, seguido de el tipo de datos que contiene y los primeros valores. De este modo sabemos que la primera variable de `registro_pingus` corresponde al número de pingüino, indicado por un vector de carácter. Las otras dos variables corresponden a la altura y al peso, ambas con datos numéricos.

Una función similar es `summary()`, la cual muestra una descripción de los datos numéricos:

```
summary(registro_pingus)
```

```
##    pinguinos           altura          peso  
##  Length:10           Min.   :28.20   Min.   :2.000  
##  Class  :character   1st Qu.:30.85   1st Qu.:2.225  
##  Mode   :character   Median :31.70   Median :2.450  
##                      Mean   :31.65   Mean   :2.490  
##                      3rd Qu.:32.42   3rd Qu.:2.700  
##                      Max.   :35.00   Max.   :3.100
```

3.5.4.2 Acceder y manipular datos La información contenida en un data frame puede ser accedida por columnas usando el operador \$ después del nombre del objeto: `data_frame$columna`.

```
registro_pingus$altura  
## [1] 30.7 28.2 33.5 32.0 31.3 32.5 31.4 29.7 35.0 32.2
```

Los nombres de las columnas pueden ser consultados con la función `colnames()`:

```
colnames(registro_pingus)  
## [1] "pinguiños" "altura"    "peso"
```

Esta función es útil cuando los data frames son muy grandes y no se pueden visualizar todas las columnas en la consola, como es el caso de `world_bank_pop` (debe tenerse cargado el paquete `tidyverse` para acceder a esta objeto):

```
colnames(world_bank_pop)  
## [1] "country"   "indicator"  "2000"      "2001"      "2002"      "2003"  
## [7] "2004"       "2005"      "2006"      "2007"      "2008"      "2009"  
## [13] "2010"       "2011"      "2012"      "2013"      "2014"      "2015"  
## [19] "2016"       "2017"
```

Volviendo al data frame `registro_pingus`, el operador \$ se utiliza para acceder a las alturas de los pacientes:

```
registro_pingus$altura  
## [1] 30.7 28.2 33.5 32.0 31.3 32.5 31.4 29.7 35.0 32.2
```

La ventaja de utilizar RStudio es que muestra automáticamente los nombres de las columnas después de escribir \$. De esta manera es posible manipular únicamente los datos de esa columna sin alterar del data frame original e incluso guardarlo como un objeto a parte:

```

mean(registro_pingus$peso)
## [1] 2.49

pesos_pingus <- registro_pingus$peso

```

Los datos pueden también accederse de la misma manera que con las matrices, empleando los corchetes para indicar [filas, columnas]:

Para acceder a las filas 1 y 3 de todas las columnas:

```

registro_pacientes[c(1,3),]

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

```

Para acceder a todas las filas de las columnas 1 y 2:

```

registro_pacientes[,1:2]

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

```

Para acceder únicamente al quinto valor de la segunda columna:

```

registro_pacientes[5,2]

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

```

Los corchetes también permiten sustituir valores en un data frame. El nuevo valor debe ser visible al acceder a la misma posición:

```

registro_pingus[5,2] = 10
registro_pingus[5,2]

## [1] 10

```

Los cambios también deben ser evidentes en el data frame completo debido a que se está modificando el objeto `registro_pingus`:

```

registro_pingus

##      pinguinos altura peso
## 1   Eudyptula_1   30.7  2.4
## 2   Eudyptula_2   28.2  2.7
## 3   Eudyptula_3   33.5  3.1
## 4   Eudyptula_4   32.0  2.2
## 5   Eudyptula_5   10.0  2.7
## 6   Eudyptula_6   32.5  2.3
## 7   Eudyptula_7   31.4  2.5
## 8   Eudyptula_8   29.7  2.0
## 9   Eudyptula_9   35.0  2.2
## 10  Eudyptula_10  32.2  2.8

```

Para restablecer el data frame a su estructura original es necesario volver a correr los comandos que se emplearon para crearlo:

```

registro_pingus <- data.frame(pinguinos,
                                altura = c(30.7, 28.2, 33.5, 32, 31.3,
                                           32.5, 31.4, 29.7, 35, 32.2),
                                peso = c(2.4, 2.7, 3.1, 2.2, 2.7,
                                         2.3, 2.5, 2, 2.2, 2.8))

registro_pingus

##      pinguinos altura peso
## 1   Eudyptula_1   30.7  2.4

```

```

## 2 Eudyptula_2 28.2 2.7
## 3 Eudyptula_3 33.5 3.1
## 4 Eudyptula_4 32.0 2.2
## 5 Eudyptula_5 31.3 2.7
## 6 Eudyptula_6 32.5 2.3
## 7 Eudyptula_7 31.4 2.5
## 8 Eudyptula_8 29.7 2.0
## 9 Eudyptula_9 35.0 2.2
## 10 Eudyptula_10 32.2 2.8

```

Para tener siempre una copia del objeto que se desea modificar se le asigna un nuevo nombre al objeto preexistente que se desea conservar:

```
pingus <- registro_pingus
```

3.5.5 Tibble

Los tibbles son otro tipo de objeto similares a los data frame, es importante mencionarlos porque serán utilizados con las funciones del paquete `tidyverse` en secciones posteriores del manual.

Un tibble tiene la misma estructura que un data frame, pero con algunas diferencias. Visualmente, al imprimir un tibble se muestra una breve descripción que contiene las dimensiones del tibble. Después se muestran los nombres de las columnas así como el tipo de dato que agrupan:

- `int`: número entero (integer)
- `dbl`: número continuo (double)
- `chr`: carácter (character)
- `fct`: factor
- `lgl`: lógico (logical)

Para la creación de tibbles es necesario cargar la librería `tidyverse`. La función `as_tibble()` puede ser aplicada a un `data.frame` para que adquiera estas características:

```

as_tibble(iris)

## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##       <dbl>      <dbl>      <dbl>      <dbl> <fct>
## 1         5.1        3.5       1.4       0.2 setosa
## 2         4.9        3.0       1.4       0.2 setosa
## 3         4.7        3.2       1.3       0.2 setosa
## 4         4.6        3.1       1.5       0.2 setosa
## 5         5.0        3.6       1.4       0.2 setosa
## 6         5.4        3.9       1.7       0.4 setosa
## 7         4.6        3.4       1.4       0.3 setosa
## 8         5.0        3.4       1.5       0.2 setosa
## 9         4.4        2.9       1.4       0.2 setosa
## 10        4.9        3.1       1.5       0.1 setosa
## # ... with 140 more rows

```

Para todos los tibble se imprimen únicamente las 10 primeras filas, de modo que no es necesario usar `head()` cuando se utilizan estos objetos. Al final del tibble se describen los datos que no se muestran, como el número de filas y columnas restantes:

```
world_bank_pop
```

```

## # A tibble: 1,056 x 20
##   country indic~ `2000` `2001` `2002` `2003` `2004` `2005` `2006` `2007` ~
##   <chr>    <chr>    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1 ABW      SP.URB~ 4.24e4 4.30e4 4.37e4 4.42e4 4.47e+4 4.49e+4 4.49e+4 4.47e+4
## 2 ABW      SP.URB~ 1.18e0 1.41e0 1.43e0 1.31e0 9.51e-1 4.91e-1 -1.78e-2 -4.35e-1
## 3 ABW      SP.POP~ 9.09e4 9.29e4 9.50e4 9.70e4 9.87e+4 1.00e+5 1.01e+5 1.01e+5
## 4 ABW      SP.POP~ 2.06e0 2.23e0 2.23e0 2.11e0 1.76e+0 1.30e+0 7.98e-1 3.84e-1
## 5 AFG      SP.URB~ 4.44e6 4.65e6 4.89e6 5.16e6 5.43e+6 5.69e+6 5.93e+6 6.15e+6
## 6 AFG      SP.URB~ 3.91e0 4.66e0 5.13e0 5.23e0 5.12e+0 4.77e+0 4.12e+0 3.65e+0
## 7 AFG      SP.POP~ 2.01e7 2.10e7 2.20e7 2.31e7 2.41e+7 2.51e+7 2.59e+7 2.66e+7
## 8 AFG      SP.POP~ 3.49e0 4.25e0 4.72e0 4.82e0 4.47e+0 3.87e+0 3.23e+0 2.76e+0
## 9 AGO      SP.URB~ 8.23e6 8.71e6 9.22e6 9.77e6 1.03e+7 1.09e+7 1.15e+7 1.21e+7
## 10 AGO     SP.URB~ 5.44e0 5.59e0 5.70e0 5.76e0 5.75e+0 5.69e+0 4.92e+0 4.89e+0
## # ... with 1,046 more rows, 10 more variables: `2008` <dbl>, `2009` <dbl>,
## # `2010` <dbl>, `2011` <dbl>, `2012` <dbl>, `2013` <dbl>, `2014` <dbl>,
## # `2015` <dbl>, `2016` <dbl>, `2017` <dbl>, and abbreviated variable name
## # 1: indicator

```

En este caso, el objeto `world_bank_pop` ya es un tibble, por lo cual no es necesario usar `as_tibble()`:

```

class(world_bank_pop)
## [1] "spec_tbl_df" "tbl_df"        "tbl"          "data.frame"

```

Los tibbles también pueden ser construidos en R mediante `tibble()`, de manera similar a `data.frame()`:

```

peso_seco <- tibble(peso_seco = c(1.4, 1.33, 1.15, 1.23),
                     parte = c("hoja", "hoja", "raiz", "raiz"),
                     sitio = c("A", "B", "A", "B"))
peso_seco

## # A tibble: 4 x 3
##   peso_seco parte sitio
##   <dbl>   <chr> <chr>
## 1 1.4     hoja  A
## 2 1.33    hoja  B
## 3 1.15    raiz  A
## 4 1.23    raiz  B

```

Los datos ausentes (`NA`) y los datos que no son números (`NaN`) se muestran en rojo:

```

tibble(A = c(NA, NA, NA),
       B = c(NaN, NaN, NaN))

## # A tibble: 3 x 2
##   A         B
##   <lg1> <dbl>
## 1 NA      NaN
## 2 NA      NaN
## 3 NA      NaN

```

Un ultimo aspecto relevante de los tibbles es que aceptan nombres no sintácticos para cada columna. Al igual que los nombres de los objetos, los nombres de las columnas de un data frame no pueden empezar con un número y representan un nombre no sintáctico. Sin embargo, los tibbles si aceptan este tipo de nombres, solamente son rodeados por un par de tildes hacia la derecha ('), como se observa en `world_bank_pop`:

```

world_bank_pop

## # A tibble: 1,056 x 20
##   country indic~ `2000` `2001` `2002` `2003` `2004` `2005` `2006` `2007` ~
##   <chr>    <chr>    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1 ABW      SP.URB~ 4.24e4 4.30e4 4.37e4 4.42e4 4.47e+4 4.49e+4 4.49e+4 4.47e+4
## 2 ABW      SP.URB~ 1.18e0 1.41e0 1.43e0 1.31e0 9.51e-1 4.91e-1 -1.78e-2 -4.35e-1
## 3 ABW      SP.POP~ 9.09e4 9.29e4 9.50e4 9.70e4 9.87e+4 1.00e+5 1.01e+5 1.01e+5
## 4 ABW      SP.POP~ 2.06e0 2.23e0 2.23e0 2.11e0 1.76e+0 1.30e+0 7.98e-1 3.84e-1
## 5 AFG      SP.URB~ 4.44e6 4.65e6 4.89e6 5.16e6 5.43e+6 5.69e+6 5.93e+6 6.15e+6
## 6 AFG      SP.URB~ 3.91e0 4.66e0 5.13e0 5.23e0 5.12e+0 4.77e+0 4.12e+0 3.65e+0
## 7 AFG      SP.POP~ 2.01e7 2.10e7 2.20e7 2.31e7 2.41e+7 2.51e+7 2.59e+7 2.66e+7
## 8 AFG      SP.POP~ 3.49e0 4.25e0 4.72e0 4.82e0 4.47e+0 3.87e+0 3.23e+0 2.76e+0
## 9 AGO      SP.URB~ 8.23e6 8.71e6 9.22e6 9.77e6 1.03e+7 1.09e+7 1.15e+7 1.21e+7
## 10 AGO     SP.URB~ 5.44e0 5.59e0 5.70e0 5.76e0 5.75e+0 5.69e+0 4.92e+0 4.89e+0
## # ... with 1,046 more rows, 10 more variables: `2008` <dbl>, `2009` <dbl>,
## # `2010` <dbl>, `2011` <dbl>, `2012` <dbl>, `2013` <dbl>, `2014` <dbl>,
## # `2015` <dbl>, `2016` <dbl>, `2017` <dbl>, and abbreviated variable name
## # 1: indicator

```

```

##   <chr>   <chr>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1 ABW    SP.URB~ 4.24e4 4.30e4 4.37e4 4.42e4 4.47e+4 4.49e+4 4.49e+4
## 2 ABW    SP.URB~ 1.18e0 1.41e0 1.43e0 1.31e0 9.51e-1 4.91e-1 -1.78e-2 -4.35e-1
## 3 ABW    SP.POP~ 9.09e4 9.29e4 9.50e4 9.70e4 9.87e+4 1.00e+5 1.01e+5 1.01e+5
## 4 ABW    SP.POP~ 2.06e0 2.23e0 2.23e0 2.11e0 1.76e+0 1.30e+0 7.98e-1 3.84e-1
## 5 AFG    SP.URB~ 4.44e6 4.65e6 4.89e6 5.16e6 5.43e+6 5.69e+6 5.93e+6 6.15e+6
## 6 AFG    SP.URB~ 3.91e0 4.66e0 5.13e0 5.23e0 5.12e+0 4.77e+0 4.12e+0 3.65e+0
## 7 AFG    SP.POP~ 2.01e7 2.10e7 2.20e7 2.31e7 2.41e+7 2.51e+7 2.59e+7 2.66e+7
## 8 AFG    SP.POP~ 3.49e0 4.25e0 4.72e0 4.82e0 4.47e+0 3.87e+0 3.23e+0 2.76e+0
## 9 AGO    SP.URB~ 8.23e6 8.71e6 9.22e6 9.77e6 1.03e+7 1.09e+7 1.15e+7 1.21e+7
## 10 AGO   SP.URB~ 5.44e0 5.59e0 5.70e0 5.76e0 5.75e+0 5.69e+0 4.92e+0 4.89e+0
## # ... with 1,046 more rows, 10 more variables: `2008` <dbl>, `2009` <dbl>,
## #   `2010` <dbl>, `2011` <dbl>, `2012` <dbl>, `2013` <dbl>, `2014` <dbl>,
## #   `2015` <dbl>, `2016` <dbl>, `2017` <dbl>, and abbreviated variable name
## #   1: indicator

```

Estas columnas con nombres no sintácticos pueden ser accedidos con \$ al colocar también las tildes hacia la derecha o entre comillas:

```

head(world_bank_pop$`2000`)
## [1] 4.244400e+04 1.182632e+00 9.085300e+04 2.055027e+00 4.436299e+06
## [6] 3.912228e+00

head(world_bank_pop$"2000")
## [1] 4.244400e+04 1.182632e+00 9.085300e+04 2.055027e+00 4.436299e+06
## [6] 3.912228e+00

world_bank_pop$`2000`
## Error: <text>:1:16: unexpected numeric constant
## 1: world_bank_pop$`2000`                                ^
## 
```

3.5.6 Objetos con tres dimensiones: Listas, arreglos y más

Son aquellos que pueden contener cualquier tipo de objeto en cualquier orden. Las listas se crean con `list()`, colocando los nombres de los objetos a almacenar, igualmente se pueden usar las funciones que crean a cada elemento de la lista:

```

lista_1 <- list(vector_1,
                  matriz_a,
                  registro_pingus)
lista_1
## [[1]]
## [1] 2 4 6 8
##
## [[2]]
## [,1] [,2] [,3] [,4]
## [1,]    1    6   11   16
## [2,]    2    7   12   17
## [3,]    3    8   13   18
## [4,]    4    9   14   19
## [5,]    5   10   15   20
##
## [[3]]
##      pinguinos altura peso

```

```

## 1 Eudyptula_1 30.7 2.4
## 2 Eudyptula_2 28.2 2.7
## 3 Eudyptula_3 33.5 3.1
## 4 Eudyptula_4 32.0 2.2
## 5 Eudyptula_5 31.3 2.7
## 6 Eudyptula_6 32.5 2.3
## 7 Eudyptula_7 31.4 2.5
## 8 Eudyptula_8 29.7 2.0
## 9 Eudyptula_9 35.0 2.2
## 10 Eudyptula_10 32.2 2.8

```

La manera de acceder a los elementos de una lista es a través de dobles corchetes `[]`, indicando el número de elemento al cual se desea acceder. Debido a que el vector fue el primer elemento en introducirse en `list()`, ese corresponde al primer elemento de la lista:

```

lista_1[[1]]
## [1] 2 4 6 8

```

Para acceder a los elementos del vector se tiene que usar un solo par de corchetes una vez que se ha especificado el elemento de la lista:

```

lista_1[[1]][3]
## [1] 6

```

Se puede usar la misma sintaxis para una matriz:

```

lista_1[[2]][2,]
## [1] 2 7 12 17

```

Para un data frame también se puede usar el operador `$`:

```

lista_1[[3]]$pinguiños
## [1] "Eudyptula_1"  "Eudyptula_2"  "Eudyptula_3"  "Eudyptula_4"  "Eudyptula_5"
## [6] "Eudyptula_6"  "Eudyptula_7"  "Eudyptula_8"  "Eudyptula_9"  "Eudyptula_10"

```

Para tener una idea general de todos los elementos de la lista, se puede usar `str()` para obtener un resumen del objeto:

```

str(lista_1)
## List of 3
## $ : num [1:4] 2 4 6 8
## $ : int [1:5, 1:4] 1 2 3 4 5 6 7 8 9 10 ...
## $ : 'data.frame':   10 obs. of  3 variables:
##   ..$ pinguiños: chr [1:10] "Eudyptula_1" "Eudyptula_2" "Eudyptula_3" "Eudyptula_4" ...
##   ..$ altura   : num [1:10] 30.7 28.2 33.5 32 31.3 32.5 31.4 29.7 35 32.2
##   ..$ peso     : num [1:10] 2.4 2.7 3.1 2.2 2.7 2.3 2.5 2 2.2 2.8

```

El resultado no indica que se trata de un vector y una matriz para los dos primeros objetos, pero se muestran sus dimensiones entre los corchetes, en donde se puede ver que el primer elemento solo tiene una dimensión, mientras que el segundo tiene 5 filas y cuatro columnas. Para el data frame la descripción es más detallada, indicando sus dimensiones y una descripción de cada columna:

```

class(lista_1)
## [1] "list"

```

Los objetos tridimensionales serán principalmente utilizados en las pruebas de hipótesis, cuyos resultados son almacenados en listas con estructuras particulares para cada función. En esos casos, lo más importante es

entender que una lista puede almacenar cualquier tipo de objeto y las maneras en las que se pueden acceder a ellos. Es por ello que no serán descritos con mucho detalle.

3.6 Introducción de datos

Los datos que se trabajan en R no suelen ser creados a través de las funciones para cada uno de los objetos anteriores, generalmente se crean las bases de datos en hojas de cálculo usando programas como Excel, Calc o Google Sheets.

3.6.1 Formato del archivo

Los datos deben ser exportados en formato .csv (comma separated values) para que puedan ser correctamente leídos por R. De manera general, para evitar problemas al introducirlos, antes de exportarlos es necesario revisar que las columnas tengan nombre y que los nombres no tengan espacios, las palabras pueden ser separadas con guión bajo (_) o con puntos (.). No es necesario numerar las filas debido a que R mostrará el número de cada observación como parte del data frame.

El archivo no debe contener comas ni tabulaciones debido a que eso podría interferir con la delimitación de columnas. Finalmente, se deben evitar acentos y letras ñ debido a que R podría no reconocer esos caracteres dependiendo del formato de codificación utilizado. Este formato puede ser cambiado en el menú de opciones globales o mediante la función `options()`. Para evitar problemas con algunos caracteres que podrían no ser reconocidos se debe utilizar el formato (encoding) UTF8.

```
options(encoding = "UTF8")
```

En caso de que se utilicen datos con un formato desconocido se puede usar la función `Encoding()` para identificar el formato que debe usarse

```
Encoding("¿Qué formato es este con caracteres extraños como ñ y ç?")
## [1] "UTF-8"
```

Debido a que los datos van a ser introducidos como un data frame, es necesario que su estructura sea rectangular. En caso de que algunas observaciones tengan variables ausentes, estos deben ser indicados como NA. Las celdas que se dejan vacías en Excel o en cualquier otro programa para manipular hojas de cálculo, son interpretadas como NA al introducir el archivo, pero aún así es recomendable escribir en cada celda vacía NA antes de su uso en R para evitar problemas con caracteres invisibles como espacios o tabuladores que pudieran encontrarse en las celdas.

De igual manera deben tenerse en cuenta que los nombres de las columnas no pueden ser únicamente números, a menos de que el archivo se introduzca como un tibble.

3.6.2 Cargar datos a R

Para introducir datos en R se utilizan funciones que indiquen el lugar en el que se encuentra el archivo dentro de una unidad de almacenamiento, es decir, la ruta del archivo. Debido a esto es útil contar con una carpeta fácilmente identifiable en donde se encuentren todos los archivos que se usarán en R y allí mismo se almacenarán todos los resultados que se obtengan (gráficas, nuevas tablas, etc). De igual manera es útil colocar allí el archivo del script que se está usando para manipular los datos dentro de la carpeta.

3.6.2.1 Ruta del archivo Una vez identificada la carpeta (o directorio) de trabajo, se debe colocar esa ruta en una función que permita leer un archivo .csv. Para evitar escribir la ruta completa de cada uno, se puede indicar un directorio de trabajo a través de `setwd()` (set working directory). Mediante esta función se le indica a R que las rutas de todos los archivos deberán empezar en la carpeta designada, en lugar de que comiencen desde la unidad de almacenamiento. Por ejemplo, en mi caso la ruta de la carpeta en la que estarán todos los archivos necesarios para este manual es `/home/leot/Documents/programacion/R/introduccion_R_biotologia`, por lo que la función debe ser:

```
setwd("/home/leot/Documents/programacion/R/introduccion_R_biotologia")
```

En el caso de Windows:

```
setwd("C:/leot/Documents/programacion/R/introduccion_R_biotologia")
```

En cualquier caso, la ruta del archivo deseado puede ser consultado en las propiedades del archivo (clic derecho>propiedades).

Para consultar o corroborar el directorio de trabajo se utiliza la función `getwd()` (get working directory), sin ningún argumento entre ambos paréntesis.

```
getwd()
```

```
## [1] "/home/leot/Documents/programacion/R/introduccion_R_biotologia"
```

RStudio cuenta con la posibilidad de asignar el directorio de trabajo mediante una interfaz gráfica que puede usarse como cualquier otro buscador de archivos. En el panel de la esquina inferior derecha, la primera pestaña muestra todos los archivos del sistema y tiene opciones para crear carpetas y archivos, borrarlos o cambiar su nombre. Hacia la derecha se encuentra un botón para mostrar más opciones, en las cuales se puede seleccionar la carpeta actual como directorio de trabajo al hacer clic en la opción: **Set working directory**. Esta acción tiene el mismo efecto que usar la función `setwd()`, ya que se imprime en la consola la función con la ruta de la carpeta seleccionada.

Ahora que se tiene establecido el directorio de trabajo, se usa `read.csv()` y se especifica la ruta en `file` entre comillas:

```
read.csv(file = "datos_manual/pesos_seriola.csv")
```

```
##      peso   especie     sexo
## 1  15.10  zonata masculino
## 2  15.38  zonata masculino
## 3  15.79  zonata masculino
## 4  15.35  zonata masculino
## 5  15.96  zonata masculino
## 6  16.54  zonata masculino
## 7  16.55  zonata masculino
## 8  14.62  zonata masculino
## 9  16.28  zonata masculino
## 10 15.70  zonata masculino
## 11 15.42  zonata masculino
## 12 15.07  zonata masculino
## 13 14.96  zonata masculino
## 14 14.64  zonata masculino
## 15 16.11  zonata masculino
## 16 15.56  zonata masculino
## 17 15.47  zonata masculino
## 18 15.75  zonata masculino
## 19 13.61  zonata masculino
## 20 14.76  zonata masculino
## 21 13.91  zonata femenino
## 22 12.82  zonata femenino
## 23 13.53  zonata femenino
## 24 13.16  zonata femenino
## 25 13.97  zonata femenino
## 26 13.73  zonata femenino
## 27 14.03  zonata femenino
## 28 13.65  zonata femenino
## 29 15.12  zonata femenino
## 30 13.07  zonata femenino
```

```

## 31 12.93    zonata  femenino
## 32 13.43    zonata  femenino
## 33 14.73    zonata  femenino
## 34 13.55    zonata  femenino
## 35 14.77    zonata  femenino
## 36 13.57    zonata  femenino
## 37 14.06    zonata  femenino
## 38 14.18    zonata  femenino
## 39 13.40    zonata  femenino
## 40 13.85    zonata  femenino
## 41 8.42     rivoliana masculino
## 42 11.00    rivoliana masculino
## 43 9.14     rivoliana masculino
## 44 9.86     rivoliana masculino
## 45 9.34     rivoliana masculino
## 46 11.22    rivoliana masculino
## 47 9.61     rivoliana masculino
## 48 8.46     rivoliana masculino
## 49 11.63    rivoliana masculino
## 50 10.07    rivoliana masculino
## 51 12.96    rivoliana masculino
## 52 9.19     rivoliana masculino
## 53 6.21     rivoliana masculino
## 54 9.07     rivoliana masculino
## 55 10.07   rivoliana masculino
## 56 10.16   rivoliana masculino
## 57 10.56   rivoliana masculino
## 58 9.24     rivoliana masculino
## 59 8.40     rivoliana masculino
## 60 11.41   rivoliana masculino
## 61 11.14   rivoliana femenino
## 62 10.71   rivoliana femenino
## 63 11.37   rivoliana femenino
## 64 10.60   rivoliana femenino
## 65 10.14   rivoliana femenino
## 66 11.32   rivoliana femenino
## 67 9.98     rivoliana femenino
## 68 9.90     rivoliana femenino
## 69 11.16   rivoliana femenino
## 70 10.65   rivoliana femenino
## 71 10.95   rivoliana femenino
## 72 10.47   rivoliana femenino
## 73 11.09   rivoliana femenino
## 74 9.72     rivoliana femenino
## 75 10.46   rivoliana femenino
## 76 10.38   rivoliana femenino
## 77 11.49   rivoliana femenino
## 78 10.46   rivoliana femenino
## 79 10.45   rivoliana femenino
## 80 10.76   rivoliana femenino

```

Cabe destacar que como resultado de la función únicamente se lee el archivo como un data frame, pero para trabajar con él es necesario asignarle un nombre en el ambiente global. A través de ese nombre se puede acceder al objeto para su posterior uso.

```

pesos_seriola <- read.csv(file = "datos_manual/pesos_seriola.csv")
head(pesos_seriola)

##     peso especie      sexo
## 1 15.10  zonata masculino
## 2 15.38  zonata masculino
## 3 15.79  zonata masculino
## 4 15.35  zonata masculino
## 5 15.96  zonata masculino
## 6 16.54  zonata masculino

```

La función `read.csv()` solo necesita el argumento `file` en donde se coloca la ruta del archivo entre comillas. La ventaja de usar Rstudio es que al escribir las comillas y presionar la tecla de tabulador se mostrarán las carpetas disponibles a elegir. Al seleccionar una de ellas se puede presionar de nuevo tabulador para elegir un archivo o alguna una de las carpetas contenidas dentro del directorio actual. De este modo es posible navegar a través de una serie de carpetas sin tener que conocer toda la ruta.

Algo a tomar en cuenta es que Windows muestra las rutas de los directorios con diagonales hacia atrás (backslashes) \, pero R necesita que sean introducidas con diagonales hacia enfrente (forwardslashes) /. Si la ruta se copia desde la ventana de propiedades del archivo deseado y se pega en la función, será necesario cambiar cada una de las diagonales.

3.6.2.2 Funciones para introducir datos Previamente se mencionó la función `read.csv()` para introducir un archivo en R. Sin embargo, existen otras opciones para lograr el mismo resultado. En general, es conveniente introducir los datos con una función que no requiera más interacción manual que escribir la función y después mandarla a la consola, como se realiza con `read.csv()`.

Sin embargo, algunas funciones operan de manera distinta, como es el caso de `file.choose()`, la cual abre una ventana para que el archivo se elija manualmente:

```
file.choose()
```

De esta manera se evita escribir la ruta del archivo y en cambio muestra una interfaz gráfica que es más familiar. La “desventaja” de esta función es que cada vez que se utilice el script, los archivos empleados tendrán que seleccionarse manualmente en lugar de solo correr las funciones correspondientes.

Es por esto que usualmente conviene introducir los archivos con funciones similares a `read.csv()`. Su documentación muestra que está asociada con `read.table()`, `read.delim()`, `read.csv2()` etc.

```
?read.csv()
```

Lo importante de tomar en cuenta es que `read.csv()` es una variación de `read.delim()` para leer únicamente archivos separados por comas. En caso de que se cuente con un archivo separado por espacios, tabuladores o por cualquier otro carácter, es posible seleccionarlo mediante `read.delim()` mediante el argumento `sep`, e indicando el carácter de separación entre comillas. Por ejemplo, si se buscara leer un archivo separado por espacios, se puede usar `read.delim()` de la siguiente manera:

```

peso_altura <- read.delim(file = "datos_manual/peso_y_altura.txt",
                           sep = " ")
## Warning in file(file, "rt"): cannot open file 'datos_manual/peso_y_altura.txt':
## No such file or directory
## Error in file(file, "rt"): cannot open the connection
head(peso_altura)
## Error in head(peso_altura): object 'peso_altura' not found

```

En el argumento `sep` se coloca un espacio. Así es como se puede leer cualquier archivo si es que se conoce su estructura y el separador que se está usando. En caso de que no se tenga claro esto último, también se

puede usar `read.delim()` sin especificar nada en `sep` y la función interpretará la estructura a partir de las primeras líneas, pero no siempre se obtiene el resultado deseado.

Este grupo de funciones cuenta con otros argumentos que pueden ser útiles para cualquier archivo que se introduzca, los cuales pueden ser revisados en la documentación, algunos que podrían ser útiles se describirán a continuación:

- `header`: puede tener un valor falso o verdadero, pero el valor por defecto es verdadero. Así se indica que la primera fila del archivo contiene a los nombres de las columnas que serán asignados en R. Si se indica como falso las columnas se nombrarán como V1, V2, V3, ... (variable 1, variable 2, variable 3, ...).
- `rownames`: permite especificar un vector numérico o de carácter para enumerar o nombrar a las filas, respectivamente. El vector indicado puede ser un objeto de R o una columna dentro del archivo seleccionado que contenga los números o nombres de cada fila. Si no se indica en la función, las filas serán numeradas automáticamente, por lo que no necesario usar este argumento a menos de que las filas deban ser nombradas de manera específica. Un ejemplo de un data frame con columnas nombradas es `swiss`.
- `col.names`: permite introducir un vector con los nombres de las columnas para especificar que son diferentes a los que se encuentra en la primer fila del archivo, si es que los hay o si el argumento `header` es falso.
- `na.strings`: indica qué conjunto de caracteres debe ser interpretado como un valor ausente (NA), además de celdas vacías. En general, si se dejan las celdas vacías.

Las funciones previamente mencionadas son parte de la paquetería básica de R y están disponibles desde que se inicia la sesión. Sin embargo, existen funciones similares que son más modernas. En particular, aquellas del paquete `readr` cuentan con la ventaja de que introducen los datos como un tibble y permiten seleccionar el tipo de datos para cada columna, lo cual es útil para evitar problemas posteriores.

La función más relevante es `read_csv()`, la única diferencia con `read.csv()` es el uso de guión bajo en lugar del punto. Esto resulta útil y a la vez confuso debido a que una función podría ser utilizada en lugar de otra y generar un error no deseado al usar argumentos no disponibles en alguna de ellas. Pero más allá de reconocer esta particularidad no existe ningún conflicto, solo es necesario cargar el paquete `readr` antes de usar `read_csv()`. De igual manera se puede cargar la paquetería `tidyverse` la cual contiene a `readr`. Este paquete cuenta también con las funciones `read_delim()` y `read_tsv()`, esta última se utiliza únicamente para leer archivos separados por tabuladores.

Al usar cualquiera de estas funciones, los datos son interpretados únicamente como datos numéricos y como datos de carácter:

```
read_csv(file = "datos_manual/pesos_seriola.csv")
## Rows: 80 Columns: 3
## -- Column specification -----
## Delimiter: ","
## chr (2): especie, sexo
## dbl (1): peso
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

## # A tibble: 80 x 3
##       peso especie sexo
##   <dbl> <chr>   <chr>
## 1 15.1  zonata masculino
## 2 15.4  zonata masculino
## 3 15.8  zonata masculino
```

```

## 4 15.4 zonata masculino
## 5 16.0 zonata masculino
## 6 16.5 zonata masculino
## 7 16.6 zonata masculino
## 8 14.6 zonata masculino
## 9 16.3 zonata masculino
## 10 15.7 zonata masculino
## # ... with 70 more rows

```

Es conveniente tener clara cual es la estructura de los datos para indicar también aquellas columnas que contienen datos categóricos. El ejemplo anterior cuenta con los registros de peces del género *Seriola*, de acuerdo con el sexo y especie. De este modo, la primera columna corresponde a datos numéricos y las otras dos son factores no ordenados con dos categorías cada uno. Al usar la función se imprime en la consola un mensaje con detalles sobre el archivo, incluido el tipo de datos que la función ha interpretado y ambos factores han sido introducidos como datos de carácter.

Para el adecuado funcionamiento de estos datos en R, es necesario especificar los factores dentro del archivo mediante el argumento `col_types`. La manera más sencilla de utilizarlo es colocando una cadena de caracteres en la que cada letra corresponde al tipo de datos en cada columna de acuerdo con la siguiente clave:

- c: carácter
- i: números enteros (integer)
- n: números continuos
- d: números continuos
- l: lógicos
- f: factor
- D: fecha (date)
- t: hora (time)
- T: fecha y hora

Debe tenerse en cuenta el orden de las columnas en el archivo. En nuestro caso, primero se encuentran los pesos y luego los dos factores, por lo que la cadena de caracteres debe ser `nff`:

```

read_csv(file = "datos_manual/pesos_seriola.csv", col_types = "nff")

## # A tibble: 80 x 3
##       peso especie sexo
##   <dbl> <fct>   <fct>
## 1 15.1 zonata masculino
## 2 15.4 zonata masculino
## 3 15.8 zonata masculino
## 4 15.4 zonata masculino
## 5 16.0 zonata masculino
## 6 16.5 zonata masculino
## 7 16.6 zonata masculino
## 8 14.6 zonata masculino
## 9 16.3 zonata masculino
## 10 15.7 zonata masculino
## # ... with 70 more rows

```

En caso de que se requiera una especificación más detallada, se utiliza la función `cols()` en el argumento `col_types()`, la cual agrupa funciones subsecuentes para establecer los parámetros de cada columna:

- `col_integer()`: números enteros

- `col_double()`: números continuos
- `col_factor(levels, ordered)`: factores

La función dedicada a los factores tiene los argumentos `levels` y `ordered`. El primero requiere un vector de carácter que contenga los niveles de la columna, mientras que el segundo únicamente necesita un valor lógico (verdadero o falso) para indicar si el vector proporcionado es ordenado o no. En caso de que sea ordenado, el vector proporcionado debe tener a los valores en orden creciente:

```
read_csv(file = "datos_manual/pesos_seriola.csv",
         col_types = cols(col_double(),
                          col_factor(levels = c("masculino", "femenino"),
                                      ordered = FALSE),
                          col_factor(levels = c("zonata", "rivoliana"),
                                      ordered = FALSE)))

## Warning: One or more parsing issues, call `problems()` on your data frame for details,
## e.g.:
##   dat <- vroom(...)
##   problems(dat)

## # A tibble: 80 x 3
##       peso especie sexo
##   <dbl> <fct>   <fct>
## 1 15.1  <NA>    <NA>
## 2 15.4  <NA>    <NA>
## 3 15.8  <NA>    <NA>
## 4 15.4  <NA>    <NA>
## 5 16.0  <NA>    <NA>
## 6 16.5  <NA>    <NA>
## 7 16.6  <NA>    <NA>
## 8 14.6  <NA>    <NA>
## 9 16.3  <NA>    <NA>
## 10 15.7 <NA>    <NA>
## # ... with 70 more rows
```

En este ejemplo el uso de los argumentos dentro de `col_factor()` no es esencial, debido a que no se tratan de factores con un orden jerárquico.

4 El paquete Tidyverse

En las secciones posteriores del manual, el manejo de los datos será a través de las librerías contenidas en el paquete `tidyverse`. Al correr el comando `library(tidyverse)` serán cargados los siguientes paquetes:

- `ggplot2`: visualización de datos.
- `dplyr`: manipulación de datos.
- `tidyr`: reorganización de datos.
- `readr`: importar datos.
- `purrr`: programación funcional.
- `tibble`: uso de tibbles.
- `stringr`: uso de cadenas de caracteres.
- `forcats`: uso de factores.

Estos paquetes han sido diseñados para ser utilizados en conjunto bajo una serie de principios que pueden consultarse aquí. Los paquetes de `tidyverse` que serán utilizados en el manual serán principalmente `ggplot2`, `dplyr`, `readr` y `tibble`. Los dos últimos ya han sido utilizados previamente, debido a que permiten cargar datos como tibbles y crear tibbles directamente en R, los cuales también serán utilizados por los demás paquetes.

Otra de las características importantes de estos paquetes es que utilizan el operador `%>%` conocido como tubería (pipe), el cual puede ser utilizado con todos los paquetes en R, no solamente con aquellos de `tidyverse`, pero es necesario primero cargar este paquete para habilitar su uso. La tubería permite simplificar el código de manera que el resultado de una función pueda ser dirigido a una nueva función en la línea inferior. De esta manera no se tienen que anidar las funciones.

Por ejemplo, para crear un vector con `seq()` y obtener la media de ese vector, se debe de anidar a `seq()` dentro de `mean()`:

```
mean(seq(from = 0, to = 10, by = .2))  
## [1] 5
```

Usando `%>%`, es posible separar ambas funciones en líneas diferentes, creando primero el objeto que se desea utilizar y después la función que se desea aplicar sobre el objeto creado en la primera línea:

```
seq(from = 0, to = 10, by = .2) %>%  
  mean()  
  
## [1] 5
```

No es necesario colocar ningún argumento en `mean()` debido a que se toma como primer argumento toda la línea anterior que es conectada con `%>%`, pero si se pueden colocar argumentos adicionales:

```
seq(from = 0, to = 10, by = .2) %>%  
  mean(na.rm = T)  
  
## [1] 5
```

4.1 dplyr

Este paquete permite transformar los datos a través de funciones nombradas como verbos, las cuales junto con el operador `%>%` crean una sintaxis que comienza con el data frame que se desea modificar y posteriormente se aplican las transformaciones deseadas mediante:

- `select()`: selecciona columnas.
- `arrange()`: organiza los datos de acuerdo con una columna.
- `filter()`: crea un subconjunto a partir de comparaciones.
- `mutate()`: crea nuevas variables mediante operaciones aplicadas sobre los datos preexistentes.
- `group_by()`: agrupa los datos de acuerdo con una variable categórica.
- `summarize()`: realiza cálculos usando el agrupamiento creado por `group_by()`.

4.1.1 Select

Permite seleccionar las columnas que se desean visualizar en el orden que se especifica en la función. El data frame `world_bank_pop` contiene datos de población provenientes del Banco Mundial. Las primera columna cuenta con el código de tres letras para cada país. Para cada país hay cuatro observaciones, que se indican en la segunda columna y cada una indica crecimiento poblacional (`SP.POP.GROW`), población total (`SP.POP.TOTL`), crecimiento de población urbana (`SP.URB.GROW`) y población urbana total (`SP.URB.TOTL`). Las variables restantes corresponden a la población de cada país de acuerdo con los indicadores desde 2000 hasta 2018.


```

## .. `1971` = col_double(),
## .. `1972` = col_double(),
## .. `1973` = col_double(),
## .. `1974` = col_double(),
## .. `1975` = col_double(),
## .. `1976` = col_double(),
## .. `1977` = col_double(),
## .. `1978` = col_double(),
## .. `1979` = col_double(),
## .. `1980` = col_double(),
## .. `1981` = col_double(),
## .. `1982` = col_double(),
## .. `1983` = col_double(),
## .. `1984` = col_double(),
## .. `1985` = col_double(),
## .. `1986` = col_double(),
## .. `1987` = col_double(),
## .. `1988` = col_double(),
## .. `1989` = col_double(),
## .. `1990` = col_double(),
## .. `1991` = col_double(),
## .. `1992` = col_double(),
## .. `1993` = col_double(),
## .. `1994` = col_double(),
## .. `1995` = col_double(),
## .. `1996` = col_double(),
## .. `1997` = col_double(),
## .. `1998` = col_double(),
## .. `1999` = col_double(),
## .. `2000` = col_double(),
## .. `2001` = col_double(),
## .. `2002` = col_double(),
## .. `2003` = col_double(),
## .. `2004` = col_double(),
## .. `2005` = col_double(),
## .. `2006` = col_double(),
## .. `2007` = col_double(),
## .. `2008` = col_double(),
## .. `2009` = col_double(),
## .. `2010` = col_double(),
## .. `2011` = col_double(),
## .. `2012` = col_double(),
## .. `2013` = col_double(),
## .. `2014` = col_double(),
## .. `2015` = col_double(),
## .. `2016` = col_double(),
## .. `2017` = col_double(),
## .. `2018` = col_double(),
## .. X64 = col_logical()
## .. )

```

En este caso podrían seleccionarse los datos respecto a un solo año. Debido a que los números de las columnas son no sintácticos, deben escribirse entre comillas, pero para las primeras dos columnas no es necesario:

```
world_bank_pop %>%
```

```

  select(country, indicator, "2010")

## # A tibble: 1,056 x 3
##   country indicator      `2010` 
##   <chr>    <chr>        <dbl>
## 1 ABW     SP.URB.TOTL  43778  
## 2 ABW     SP.URB.GROW  -0.624  
## 3 ABW     SP.POP.TOTL  101669  
## 4 ABW     SP.POP.GROW  0.213  
## 5 AFG     SP.URB.TOTL  6837008 
## 6 AFG     SP.URB.GROW  3.70    
## 7 AFG     SP.POP.TOTL  28803167 
## 8 AFG     SP.POP.GROW  2.81    
## 9 AGO     SP.URB.TOTL  13970768 
## 10 AGO    SP.URB.GROW  4.83    
## # ... with 1,046 more rows

```

El resultado de `select()` es un nuevo tibble que contiene únicamente las columnas seleccionadas mientras que el original está intacto. El resultado de la función puede guardarse como un nuevo objeto con `<-`:

```
poblacion_2010 <- world_bank_pop %>%
  select(country, indicator, "2010")
```

`poblacion_2010`

```

## # A tibble: 1,056 x 3
##   country indicator      `2010` 
##   <chr>    <chr>        <dbl>
## 1 ABW     SP.URB.TOTL  43778  
## 2 ABW     SP.URB.GROW  -0.624  
## 3 ABW     SP.POP.TOTL  101669  
## 4 ABW     SP.POP.GROW  0.213  
## 5 AFG     SP.URB.TOTL  6837008 
## 6 AFG     SP.URB.GROW  3.70    
## 7 AFG     SP.POP.TOTL  28803167 
## 8 AFG     SP.POP.GROW  2.81    
## 9 AGO     SP.URB.TOTL  13970768 
## 10 AGO    SP.URB.GROW  4.83    
## # ... with 1,046 more rows

```

Para seleccionar múltiples columnas que están ordenadas consecutivamente se puede usar `:`. El orden puede consultarse con `str()` o `colnames()`:

```

world_bank_pop %>%
  select(country, indicator, "2010": "2015")

## # A tibble: 1,056 x 8
##   country indicator      `2010`      `2011`      `2012`      `2013`      `2014`      `2015` 
##   <chr>    <chr>        <dbl>       <dbl>       <dbl>       <dbl>       <dbl>       <dbl> 
## 1 ABW     SP.URB.TOTL  43778     43822     4.41e+4  4.44e+4  4.47e+4  4.50e+4
## 2 ABW     SP.URB.GROW  -0.624    0.100     5.51e-1  6.70e-1  7.05e-1  6.80e-1
## 3 ABW     SP.POP.TOTL  101669    102053    1.03e+5  1.03e+5  1.04e+5  1.04e+5
## 4 ABW     SP.POP.GROW  0.213     0.377     5.12e-1  5.93e-1  5.87e-1  5.25e-1
## 5 AFG     SP.URB.TOTL  6837008   7114615   7.42e+6  7.73e+6  8.05e+6  8.37e+6
## 6 AFG     SP.URB.GROW  3.70      3.98      4.15e+0  4.19e+0  4.06e+0  3.82e+0
## 7 AFG     SP.POP.TOTL  28803167  29708599  3.07e+7  3.17e+7  3.28e+7  3.37e+7
## 8 AFG     SP.POP.GROW  2.81      3.10      3.27e+0  3.32e+0  3.18e+0  2.94e+0

```

```

## 9 AGO      SP.URB.TOTL 13970768      14659013      1.54e+7 1.61e+7 1.69e+7 1.77e+7
## 10 AGO     SP.URB.GROW       4.83        4.81 4.77e+0 4.72e+0 4.65e+0 4.56e+0
## # ... with 1,046 more rows

```

Esta sintaxis puede combinarse para seleccionar las columnas deseadas:

```

world_bank_pop %>%
  select(country, indicator, "2000":"2005", "2007", "2010":"2015")

## # A tibble: 1,056 x 15
##   country indicator `2000` `2001` `2002` `2003` `2004` `2005` `2007` `2010`
##   <chr>    <chr>    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1 ABW      SP.URB~ 4.24e4 4.30e4 4.37e4 4.42e4 4.47e+4 4.49e+4 4.47e+4 4.38e+4
## 2 ABW      SP.URB~ 1.18e0 1.41e0 1.43e0 1.31e0 9.51e-1 4.91e-1 -4.35e-1 -6.24e-1
## 3 ABW      SP.POP~ 9.09e4 9.29e4 9.50e4 9.70e4 9.87e+4 1.00e+5 1.01e+5 1.02e+5
## 4 ABW      SP.POP~ 2.06e0 2.23e0 2.23e0 2.11e0 1.76e+0 1.30e+0 3.84e-1 2.13e-1
## 5 AFG      SP.URB~ 4.44e6 4.65e6 4.89e6 5.16e6 5.43e+6 5.69e+6 6.15e+6 6.84e+6
## 6 AFG      SP.URB~ 3.91e0 4.66e0 5.13e0 5.23e0 5.12e+0 4.77e+0 3.65e+0 3.70e+0
## 7 AFG      SP.POP~ 2.01e7 2.10e7 2.20e7 2.31e7 2.41e+7 2.51e+7 2.66e+7 2.88e+7
## 8 AFG      SP.POP~ 3.49e0 4.25e0 4.72e0 4.82e0 4.47e+0 3.87e+0 2.76e+0 2.81e+0
## 9 AGO      SP.URB~ 8.23e6 8.71e6 9.22e6 9.77e6 1.03e+7 1.09e+7 1.21e+7 1.40e+7
## 10 AGO     SP.URB~ 5.44e0 5.59e0 5.70e0 5.76e0 5.75e+0 5.69e+0 4.89e+0 4.83e+0
## # ... with 1,046 more rows, 5 more variables: `2011` <dbl>, `2012` <dbl>,
## #   `2013` <dbl>, `2014` <dbl>, `2015` <dbl>, and abbreviated variable name
## #   1: indicator

```

Si se desea un conjunto en el cual es más fácil remover una cuantas columnas en lugar de escribir todas las restantes, se puede realizar de la siguiente manera:

```

world_bank_pop %>%
  select(-"2000")

## # A tibble: 1,056 x 19
##   country indicator `2001` `2002` `2003` `2004` `2005` `2006` `2007` `2008` `2009` `2010` `2011` `2012` `2013` `2014` `2015` `2016` `2017` 
##   <chr>    <chr>    <dbl>   <dbl>
## 1 ABW      SP.URB.TOTL 4.30e4 4.37e4 4.42e4 4.47e+4 4.49e+4 4.49e+4 4.47e+4 4.47e+4
## 2 ABW      SP.URB.GROW 1.41e0 1.43e0 1.31e0 9.51e-1 4.91e-1 -1.78e-2 -4.35e-1 -1.78e-2 -1.78e-2
## 3 ABW      SP.POP.TOTL 9.29e4 9.50e4 9.70e4 9.87e+4 1.00e+5 1.01e+5 1.01e+5
## 4 ABW      SP.POP.GROW 2.23e0 2.23e0 2.11e0 1.76e+0 1.30e+0 7.98e-1 3.84e-1 3.84e-1
## 5 AFG      SP.URB.TOTL 4.65e6 4.89e6 5.16e6 5.43e+6 5.69e+6 5.93e+6 6.15e+6 6.15e+6
## 6 AFG      SP.URB.GROW 4.66e0 5.13e0 5.23e0 5.12e+0 4.77e+0 4.12e+0 3.65e+0 3.65e+0
## 7 AFG      SP.POP.TOTL 2.10e7 2.20e7 2.31e7 2.41e+7 2.51e+7 2.59e+7 2.66e+7 2.66e+7
## 8 AFG      SP.POP.GROW 4.25e0 4.72e0 4.82e0 4.47e+0 3.87e+0 3.23e+0 2.76e+0 2.76e+0
## 9 AGO      SP.URB.TOTL 8.71e6 9.22e6 9.77e6 1.03e+7 1.09e+7 1.15e+7 1.21e+7 1.21e+7
## 10 AGO     SP.URB.GROW 5.59e0 5.70e0 5.76e0 5.75e+0 5.69e+0 4.92e+0 4.89e+0 4.89e+0
## # ... with 1,046 more rows, and 10 more variables: `2008` <dbl>, `2009` <dbl>,
## #   `2010` <dbl>, `2011` <dbl>, `2012` <dbl>, `2013` <dbl>, `2014` <dbl>,
## #   `2015` <dbl>, `2016` <dbl>, `2017` <dbl>

world_bank_pop %>%
  select(-("2000":"2010"))

## # A tibble: 1,056 x 9
##   country indicator `2011` `2012` `2013` `2014` `2015` `2016` `2017` 
##   <chr>    <chr>    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1 ABW      SP.URB.TOTL 4.38e4 4.41e4 4.44e4 4.47e4 4.50e4 4.53e4 4.56e+4
## 2 ABW      SP.URB.GROW 1.00e-1 5.51e-1 6.70e-1 7.05e-1 6.80e-1 6.56e-1 6.54e-1
## 3 ABW      SP.POP.TOTL 1.02e+5 1.03e+5 1.03e+5 1.04e+5 1.04e+5 1.05e+5 1.05e+5

```

```

## 4 ABW    SP.POP.GROW 3.77e-1 5.12e-1 5.93e-1 5.87e-1 5.25e-1 4.60e-1 4.21e-1
## 5 AFG    SP.URB.TOTL 7.11e+6 7.42e+6 7.73e+6 8.05e+6 8.37e+6 8.67e+6 8.97e+6
## 6 AFG    SP.URB.GROW 3.98e+0 4.15e+0 4.19e+0 4.06e+0 3.82e+0 3.56e+0 3.41e+0
## 7 AFG    SP.POP.TOTL 2.97e+7 3.07e+7 3.17e+7 3.28e+7 3.37e+7 3.47e+7 3.55e+7
## 8 AFG    SP.POP.GROW 3.10e+0 3.27e+0 3.32e+0 3.18e+0 2.94e+0 2.69e+0 2.49e+0
## 9 AGO    SP.URB.TOTL 1.47e+7 1.54e+7 1.61e+7 1.69e+7 1.77e+7 1.85e+7 1.93e+7
## 10 AGO   SP.URB.GROW 4.81e+0 4.77e+0 4.72e+0 4.65e+0 4.56e+0 4.47e+0 4.38e+0
## # ... with 1,046 more rows

```

`select()` también permite cambiar el orden en el que aparecen las columnas al cambiar el orden dentro de la función:

```

world_bank_pop %>%
  select(indicator, country, "2015", "2005")

## # A tibble: 1,056 x 4
##   indicator country     `2015`     `2005`
##   <chr>      <chr>      <dbl>      <dbl>
## 1 SP.URB.TOTL ABW        44979      44889
## 2 SP.URB.GROW ABW        0.680      0.491
## 3 SP.POP.TOTL ABW       104341     100031
## 4 SP.POP.GROW ABW        0.525      1.30
## 5 SP.URB.TOTL AFG       8367663     5691823
## 6 SP.URB.GROW AFG        3.82       4.77
## 7 SP.POP.TOTL AFG      33736494    25070798
## 8 SP.POP.GROW AFG        2.94       3.87
## 9 SP.URB.TOTL AGO      17675615    10949424
## 10 SP.URB.GROW AGO       4.56       5.69
## # ... with 1,046 more rows

```

4.1.2 Arrange

`arrange()` ordena las filas del data frame de acuerdo con el orden ascendente o descendente de las columnas especificadas. En el data frame `chickwts` se pueden ordenar los pesos de manera ascendente:

```

chickwts %>%
  arrange(weight)

##   weight     feed
## 1    108 horsebean
## 2    124 horsebean
## 3    136 horsebean
## 4    140 horsebean
## 5    141 linseed
## 6    143 horsebean
## 7    148 linseed
## 8    153 meatmeal
## 9    158 soybean
## 10   160 horsebean
## 11   168 horsebean
## 12   169 linseed
## 13   171 soybean
## 14   179 horsebean
## 15   181 linseed
## 16   193 soybean
## 17   199 soybean
## 18   203 linseed

```

```
## 19    206 meatmeal
## 20    213 linseed
## 21    216 casein
## 22    217 horsebean
## 23    222 casein
## 24    226 sunflower
## 25    227 horsebean
## 26    229 linseed
## 27    230 soybean
## 28    242 meatmeal
## 29    243 soybean
## 30    244 linseed
## 31    248 soybean
## 32    248 soybean
## 33    250 soybean
## 34    257 linseed
## 35    257 meatmeal
## 36    258 meatmeal
## 37    260 linseed
## 38    260 casein
## 39    263 meatmeal
## 40    267 soybean
## 41    271 linseed
## 42    271 soybean
## 43    283 casein
## 44    295 sunflower
## 45    297 sunflower
## 46    303 meatmeal
## 47    309 linseed
## 48    315 meatmeal
## 49    316 soybean
## 50    318 sunflower
## 51    318 casein
## 52    320 sunflower
## 53    322 sunflower
## 54    325 meatmeal
## 55    327 soybean
## 56    329 soybean
## 57    332 casein
## 58    334 sunflower
## 59    339 sunflower
## 60    340 sunflower
## 61    341 sunflower
## 62    344 meatmeal
## 63    352 casein
## 64    359 casein
## 65    368 casein
## 66    379 casein
## 67    380 meatmeal
## 68    390 casein
## 69    392 sunflower
## 70    404 casein
## 71    423 sunflower
```

Para acomodarlos en orden descendente se debe usar `desc()`:

```
chickwts %>%
  arrange(desc(weight))

##      weight      feed
## 1      423 sunflower
## 2      404   casein
## 3      392 sunflower
## 4      390   casein
## 5      380 meatmeal
## 6      379   casein
## 7      368   casein
## 8      359   casein
## 9      352   casein
## 10     344 meatmeal
## 11     341 sunflower
## 12     340 sunflower
## 13     339 sunflower
## 14     334 sunflower
## 15     332   casein
## 16     329 soybean
## 17     327 soybean
## 18     325 meatmeal
## 19     322 sunflower
## 20     320 sunflower
## 21     318 sunflower
## 22     318   casein
## 23     316 soybean
## 24     315 meatmeal
## 25     309 linseed
## 26     303 meatmeal
## 27     297 sunflower
## 28     295 sunflower
## 29     283   casein
## 30     271 linseed
## 31     271 soybean
## 32     267 soybean
## 33     263 meatmeal
## 34     260 linseed
## 35     260   casein
## 36     258 meatmeal
## 37     257 linseed
## 38     257 meatmeal
## 39     250 soybean
## 40     248 soybean
## 41     248 soybean
## 42     244 linseed
## 43     243 soybean
## 44     242 meatmeal
## 45     230 soybean
## 46     229 linseed
## 47     227 horsebean
## 48     226 sunflower
## 49     222   casein
```

```

## 50    217 horsebean
## 51    216   casein
## 52    213 linseed
## 53    206 meatmeal
## 54    203 linseed
## 55    199 soybean
## 56    193 soybean
## 57    181 linseed
## 58    179 horsebean
## 59    171 soybean
## 60    169 linseed
## 61    168 horsebean
## 62    160 horsebean
## 63    158 soybean
## 64    153 meatmeal
## 65    148 linseed
## 66    143 horsebean
## 67    141 linseed
## 68    140 horsebean
## 69    136 horsebean
## 70    124 horsebean
## 71    108 horsebean

```

Cuando se selecciona una columna que contiene datos de carácter solo se organizará en orden alfabético ascendente o descendente:

```

chickwts %>%
  arrange(feed)

##      weight      feed
## 1     368   casein
## 2     390   casein
## 3     379   casein
## 4     260   casein
## 5     404   casein
## 6     318   casein
## 7     352   casein
## 8     359   casein
## 9     216   casein
## 10    222   casein
## 11    283   casein
## 12    332   casein
## 13    179 horsebean
## 14    160 horsebean
## 15    136 horsebean
## 16    227 horsebean
## 17    217 horsebean
## 18    168 horsebean
## 19    108 horsebean
## 20    124 horsebean
## 21    143 horsebean
## 22    140 horsebean
## 23    309 linseed
## 24    229 linseed
## 25    181 linseed

```

```

## 26    141 linseed
## 27    260 linseed
## 28    203 linseed
## 29    148 linseed
## 30    169 linseed
## 31    213 linseed
## 32    257 linseed
## 33    244 linseed
## 34    271 linseed
## 35    325 meatmeal
## 36    257 meatmeal
## 37    303 meatmeal
## 38    315 meatmeal
## 39    380 meatmeal
## 40    153 meatmeal
## 41    263 meatmeal
## 42    242 meatmeal
## 43    206 meatmeal
## 44    344 meatmeal
## 45    258 meatmeal
## 46    243 soybean
## 47    230 soybean
## 48    248 soybean
## 49    327 soybean
## 50    329 soybean
## 51    250 soybean
## 52    193 soybean
## 53    271 soybean
## 54    316 soybean
## 55    267 soybean
## 56    199 soybean
## 57    171 soybean
## 58    158 soybean
## 59    248 soybean
## 60    423 sunflower
## 61    340 sunflower
## 62    392 sunflower
## 63    339 sunflower
## 64    341 sunflower
## 65    226 sunflower
## 66    320 sunflower
## 67    295 sunflower
## 68    334 sunflower
## 69    322 sunflower
## 70    297 sunflower
## 71    318 sunflower

chickwts %>%
  arrange(desc(feed))

##      weight      feed
## 1    423 sunflower
## 2    340 sunflower
## 3    392 sunflower
## 4    339 sunflower

```

```
## 5    341 sunflower
## 6    226 sunflower
## 7    320 sunflower
## 8    295 sunflower
## 9    334 sunflower
## 10   322 sunflower
## 11   297 sunflower
## 12   318 sunflower
## 13   243 soybean
## 14   230 soybean
## 15   248 soybean
## 16   327 soybean
## 17   329 soybean
## 18   250 soybean
## 19   193 soybean
## 20   271 soybean
## 21   316 soybean
## 22   267 soybean
## 23   199 soybean
## 24   171 soybean
## 25   158 soybean
## 26   248 soybean
## 27   325 meatmeal
## 28   257 meatmeal
## 29   303 meatmeal
## 30   315 meatmeal
## 31   380 meatmeal
## 32   153 meatmeal
## 33   263 meatmeal
## 34   242 meatmeal
## 35   206 meatmeal
## 36   344 meatmeal
## 37   258 meatmeal
## 38   309 linseed
## 39   229 linseed
## 40   181 linseed
## 41   141 linseed
## 42   260 linseed
## 43   203 linseed
## 44   148 linseed
## 45   169 linseed
## 46   213 linseed
## 47   257 linseed
## 48   244 linseed
## 49   271 linseed
## 50   179 horsebean
## 51   160 horsebean
## 52   136 horsebean
## 53   227 horsebean
## 54   217 horsebean
## 55   168 horsebean
## 56   108 horsebean
## 57   124 horsebean
## 58   143 horsebean
```

```

## 59    140 horsebean
## 60    368 casein
## 61    390 casein
## 62    379 casein
## 63    260 casein
## 64    404 casein
## 65    318 casein
## 66    352 casein
## 67    359 casein
## 68    216 casein
## 69    222 casein
## 70    283 casein
## 71    332 casein

```

4.1.3 Filter

Crea subconjuntos mediante comparaciones que usan operadores lógicos para evaluar los contenidos de la columna o columnas seleccionadas. Solo aquellos valores que cumplan con el parámetro establecido serán seleccionados en el nuevo subconjunto. Los operadores lógicos que se pueden usar son:

- ==: igual a.
- !=: diferente de.
- >: mayor que.
- >=: mayor o igual que.
- <: menor que.
- <=: menor o igual que.

De igual manera se pueden usar operadores booleanos:

- &: AND
- |: OR
- !: NOT

La manera más sencilla de usar `filter()` es para seleccionar todas las observaciones de una columna con datos categóricos. Por ejemplo, para seleccionar únicamente los datos correspondientes a México en `world_bank_pop` se debe indicar la columna, el operador lógico y luego el valor con el que se realizará la comparación:

```

world_bank_pop %>%
  filter(country == "MEX")

## # A tibble: 4 x 20
##   country indic~1 `2000` `2001` `2002` `2003` `2004` `2005` `2006` `2007` `2008` 
##   <chr>     <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl> 
## 1 MEX      7.60e7 7.73e7 7.86e7 8.00e7 8.13e7 8.28e7 8.43e7 8.60e7 8.78e7
## 2 MEX      1.78e0 1.75e0 1.67e0 1.64e0 1.69e0 1.78e0 1.89e0 1.97e0 2.01e0
## 3 MEX      1.02e8 1.03e8 1.04e8 1.06e8 1.07e8 1.08e8 1.10e8 1.12e8 1.14e8
## 4 MEX      1.40e0 1.32e0 1.24e0 1.22e0 1.27e0 1.37e0 1.48e0 1.57e0 1.62e0
## # ... with 9 more variables: `2009` <dbl>, `2010` <dbl>, `2011` <dbl>,
## #   `2012` <dbl>, `2013` <dbl>, `2014` <dbl>, `2015` <dbl>, `2016` <dbl>,
## #   `2017` <dbl>, and abbreviated variable name 1: indicator

```

En este caso, la comparación se lee: países es igual a México. Por el contrario, si se desean excluir todos los datos de México se utiliza ! (NOT). País no es igual a México:

```

world_bank_pop %>%
  filter(country != "MEX")

## # A tibble: 1,052 x 20
##   country indicator `2000` `2001` `2002` `2003` `2004` `2005` `2006` `2007` 
##   <chr>    <chr>    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl> 
## 1 ABW      SP.URB~ 4.24e4 4.30e4 4.37e4 4.42e4 4.47e+4 4.49e+4 4.49e+4 4.47e+4
## 2 ABW      SP.URB~ 1.18e0 1.41e0 1.43e0 1.31e0 9.51e-1 4.91e-1 -1.78e-2 -4.35e-1
## 3 ABW      SP.POP~ 9.09e4 9.29e4 9.50e4 9.70e4 9.87e+4 1.00e+5 1.01e+5 1.01e+5
## 4 ABW      SP.POP~ 2.06e0 2.23e0 2.23e0 2.11e0 1.76e+0 1.30e+0 7.98e-1 3.84e-1
## 5 AFG      SP.URB~ 4.44e6 4.65e6 4.89e6 5.16e6 5.43e+6 5.69e+6 5.93e+6 6.15e+6
## 6 AFG      SP.URB~ 3.91e0 4.66e0 5.13e0 5.23e0 5.12e+0 4.77e+0 4.12e+0 3.65e+0
## 7 AFG      SP.POP~ 2.01e7 2.10e7 2.20e7 2.31e7 2.41e+7 2.51e+7 2.59e+7 2.66e+7
## 8 AFG      SP.POP~ 3.49e0 4.25e0 4.72e0 4.82e0 4.47e+0 3.87e+0 3.23e+0 2.76e+0
## 9 AGO      SP.URB~ 8.23e6 8.71e6 9.22e6 9.77e6 1.03e+7 1.09e+7 1.15e+7 1.21e+7
## 10 AGO     SP.URB~ 5.44e0 5.59e0 5.70e0 5.76e0 5.75e+0 5.69e+0 4.92e+0 4.89e+0
## # ... with 1,042 more rows, 10 more variables: `2008` <dbl>, `2009` <dbl>,
## #   `2010` <dbl>, `2011` <dbl>, `2012` <dbl>, `2013` <dbl>, `2014` <dbl>,
## #   `2015` <dbl>, `2016` <dbl>, `2017` <dbl>, and abbreviated variable name
## #   1: indicator
```

Se pueden indicar múltiples comparaciones en la misma instancia de `filter()`, solo es necesario separar cada una mediante una coma:

```

world_bank_pop %>%
  filter(country == "MEX",
        indicator != "SP.URB.TOTL")

## # A tibble: 3 x 20
##   country indicator `2000` `2001` `2002` `2003` `2004` `2005` `2006` `2007` `2008` 
##   <chr>    <chr>    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl> 
## 1 MEX      SP.URB~ 1.78e0 1.75e0 1.67e0 1.64e0 1.69e0 1.78e0 1.89e0 1.97e0 2.01e0
## 2 MEX      SP.POP~ 1.02e8 1.03e8 1.04e8 1.06e8 1.07e8 1.08e8 1.10e8 1.12e8 1.14e8
## 3 MEX      SP.POP~ 1.40e0 1.32e0 1.24e0 1.22e0 1.27e0 1.37e0 1.48e0 1.57e0 1.62e0
## # ... with 9 more variables: `2009` <dbl>, `2010` <dbl>, `2011` <dbl>,
## #   `2012` <dbl>, `2013` <dbl>, `2014` <dbl>, `2015` <dbl>, `2016` <dbl>,
## #   `2017` <dbl>, and abbreviated variable name 1: indicator
```

Si se desea seleccionar más de un valor por que debe ser seleccionado en una columna se usa operador `|` (OR), para indicar que se busca alguno de esos dos valores, pero no los dos al mismo tiempo. La comparación se lee país es igual a México o a España:

```

world_bank_pop %>%
  filter(country == "MEX" | country == "ESP")

## # A tibble: 8 x 20
##   country indicator `2000` `2001` `2002` `2003` `2004` `2005` `2006` `2007` 
##   <chr>    <chr>    <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl> 
## 1 ESP      SP.URB.TOTL 3.09e+7 3.12e+7 3.17e+7 3.24e+7 3.31e+7 3.37e+7 3.44e+7 3.52e+7
## 2 ESP      SP.URB.GROW 5.53e-1 8.00e-1 1.66e0 2.13e0 2.04e0 2.00e0 2.00e0 2.16e0
## 3 ESP      SP.POP.TOTL 4.06e+7 4.09e+7 4.14e+7 4.22e+7 4.29e+7 4.37e+7 4.44e+7 4.52e+7
## 4 ESP      SP.POP.GROW 4.47e-1 6.94e-1 1.41e0 1.81e0 1.73e0 1.69e0 1.69e0 1.85e0
## 5 MEX      SP.URB.TOTL 7.60e+7 7.73e+7 7.86e+7 8.00e+7 8.13e+7 8.28e+7 8.43e+7 8.60e+7
## 6 MEX      SP.URB.GROW 1.78e+0 1.75e+0 1.67e0 1.64e0 1.69e0 1.78e0 1.89e0 1.97e0
## 7 MEX      SP.POP.TOTL 1.02e+8 1.03e+8 1.04e+8 1.06e+8 1.07e+8 1.08e+8 1.10e+8 1.12e+8
## 8 MEX      SP.POP.GROW 1.40e+0 1.32e+0 1.24e0 1.22e0 1.27e0 1.37e0 1.48e0 1.57e0
## # ... with 10 more variables: `2008` <dbl>, `2009` <dbl>, `2010` <dbl>,
```

```
## # `2011` <dbl>, `2012` <dbl>, `2013` <dbl>, `2014` <dbl>, `2015` <dbl>,
## # `2016` <dbl>, `2017` <dbl>
```

Esto mismo puede hacerse para aún más valores:

```
world_bank_pop %>%
  filter(country == "MEX" | country == "ESP" | country == "TUR")

## # A tibble: 12 x 20
##   country indicator    `2000` `2001` `2002` `2003` `2004` `2005` `2006` `2007` 
##   <chr>     <chr>      <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl> 
## 1 ESP       SP.URB.TOTL 3.09e+7 3.12e+7 3.17e7 3.24e7 3.31e7 3.37e7 3.44e7 3.52e7
## 2 ESP       SP.URB.GROW 5.53e-1 8.00e-1 1.66e0 2.13e0 2.04e0 2.00e0 2.00e0 2.16e0
## 3 ESP       SP.POP.TOTL 4.06e+7 4.09e+7 4.14e7 4.22e7 4.29e7 4.37e7 4.44e7 4.52e7
## 4 ESP       SP.POP.GROW 4.47e-1 6.94e-1 1.41e0 1.81e0 1.73e0 1.69e0 1.69e0 1.85e0
## 5 MEX       SP.URB.TOTL 7.60e+7 7.73e+7 7.86e7 8.00e7 8.13e7 8.28e7 8.43e7 8.60e7
## 6 MEX       SP.URB.GROW 1.78e+0 1.75e+0 1.67e0 1.64e0 1.69e0 1.78e0 1.89e0 1.97e0
## 7 MEX       SP.POP.TOTL 1.02e+8 1.03e+8 1.04e8 1.06e8 1.07e8 1.08e8 1.10e8 1.12e8
## 8 MEX       SP.POP.GROW 1.40e+0 1.32e+0 1.24e0 1.22e0 1.27e0 1.37e0 1.48e0 1.57e0
## 9 TUR       SP.URB.TOTL 4.09e+7 4.19e+7 4.30e7 4.40e7 4.50e7 4.61e7 4.71e7 4.81e7
## 10 TUR      SP.URB.GROW 2.32e+0 2.41e+0 2.44e0 2.38e0 2.32e0 2.24e0 2.15e0 2.08e0
## 11 TUR      SP.POP.TOTL 6.32e+7 6.42e+7 6.51e7 6.61e7 6.70e7 6.79e7 6.88e7 6.96e7
## 12 TUR      SP.POP.GROW 1.52e+0 1.49e+0 1.47e0 1.44e0 1.39e0 1.33e0 1.26e0 1.21e0
## # ... with 10 more variables: `2008` <dbl>, `2009` <dbl>, `2010` <dbl>,
## # `2011` <dbl>, `2012` <dbl>, `2013` <dbl>, `2014` <dbl>, `2015` <dbl>,
## # `2016` <dbl>, `2017` <dbl>
```

Esta sintaxis puede ser simplificada usando un vector, el cual requiere el uso del operador `%in%` para que la comparación sea realizada adecuadamente:

```
world_bank_pop %>%
  filter(country %in% c("MEX", "ESP", "TUR", "BRA", "CUB"))

## # A tibble: 20 x 20
##   country indicator    `2000` `2001` `2002` `2003` `2004` `2005` `2006` `2007` 
##   <chr>     <chr>      <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl> 
## 1 BRA       SP.URB.TOTL 1.42e+8 1.45e+8 1.48e+8 1.50e+8 1.52e+8 1.55e+8 1.57e+8
## 2 BRA       SP.URB.GROW 2.31e+0 1.84e+0 1.74e+0 1.68e+0 1.61e+0 1.55e+0 1.49e+0
## 3 BRA       SP.POP.TOTL 1.75e+8 1.78e+8 1.80e+8 1.82e+8 1.85e+8 1.87e+8 1.89e+8
## 4 BRA       SP.POP.GROW 1.45e+0 1.40e+0 1.34e+0 1.29e+0 1.23e+0 1.17e+0 1.11e+0
## 5 CUB       SP.URB.TOTL 8.40e+6 8.45e+6 8.50e+6 8.54e+6 8.57e+6 8.59e+6 8.61e+6
## 6 CUB       SP.URB.GROW 6.77e-1 6.28e-1 5.85e-1 4.53e-1 3.20e-1 2.76e-1 2.28e-1
## 7 CUB       SP.POP.TOTL 1.12e+7 1.12e+7 1.12e+7 1.12e+7 1.13e+7 1.13e+7 1.13e+7
## 8 CUB       SP.POP.GROW 3.66e-1 3.21e-1 2.81e-1 2.39e-1 1.96e-1 1.54e-1 1.06e-1
## 9 ESP       SP.URB.TOTL 3.09e+7 3.12e+7 3.17e+7 3.24e+7 3.31e+7 3.37e+7 3.44e+7
## 10 ESP      SP.URB.GROW 5.53e-1 8.00e-1 1.66e0 2.13e0 2.04e0 2.00e0 2.00e0
## 11 ESP      SP.POP.TOTL 4.06e+7 4.09e+7 4.14e7 4.22e7 4.29e7 4.37e7 4.44e7
## 12 ESP      SP.POP.GROW 4.47e-1 6.94e-1 1.41e0 1.81e0 1.73e0 1.69e0 1.69e0
## 13 MEX      SP.URB.TOTL 7.60e+7 7.73e+7 7.86e7 8.00e7 8.13e7 8.28e7 8.43e7
## 14 MEX      SP.URB.GROW 1.78e+0 1.75e+0 1.67e0 1.64e0 1.69e0 1.78e0 1.89e0
## 15 MEX      SP.POP.TOTL 1.02e+8 1.03e+8 1.04e8 1.06e8 1.07e8 1.08e8 1.10e8
## 16 MEX      SP.POP.GROW 1.40e+0 1.32e+0 1.24e0 1.22e0 1.27e0 1.37e0 1.48e0
## 17 TUR      SP.URB.TOTL 4.09e+7 4.19e+7 4.30e7 4.40e7 4.50e7 4.61e7 4.71e7
## 18 TUR      SP.URB.GROW 2.32e+0 2.41e+0 2.44e0 2.38e0 2.32e0 2.24e0 2.15e0
## 19 TUR      SP.POP.TOTL 6.32e+7 6.42e+7 6.51e7 6.61e7 6.70e7 6.79e7 6.88e7
## 20 TUR      SP.POP.GROW 1.52e+0 1.49e+0 1.47e0 1.44e0 1.39e0 1.33e0 1.26e0
## # ... with 11 more variables: `2007` <dbl>, `2008` <dbl>, `2009` <dbl>,
```

```
## # `2010` <dbl>, `2011` <dbl>, `2012` <dbl>, `2013` <dbl>, `2014` <dbl>,
## # `2015` <dbl>, `2016` <dbl>, `2017` <dbl>
```

Parecería lógico que el mismo resultado se obtuviera indicando el vector con “==”, pero no resulta en la selección deseada. Por eso es necesario usar `%in%` para indicar que el resultado de la comparación se encuentra dentro del vector, en lugar de que el valor de la columna país es igual a todo el vector.

```
world_bank_pop %>%
  filter(country == c("MEX", "ESP", "TUR", "BRA", "CUB"))

## Warning in country == c("MEX", "ESP", "TUR", "BRA", "CUB"): longer object length
## is not a multiple of shorter object length

## # A tibble: 3 x 20
##   country indic~1 `2000` `2001` `2002` `2003` `2004` `2005` `2006` `2007` `2008` ~
##   <chr>     <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 BRA      SP.URB~ 1.42e8 1.45e8 1.48e8 1.50e8 1.52e8 1.55e8 1.57e8 1.59e8 1.62e8
## 2 CUB      SP.POP~ 1.12e7 1.12e7 1.12e7 1.12e7 1.13e7 1.13e7 1.13e7 1.13e7 1.13e7
## 3 MEX      SP.POP~ 1.02e8 1.03e8 1.04e8 1.06e8 1.07e8 1.08e8 1.10e8 1.12e8 1.14e8
## # ... with 9 more variables: `2009` <dbl>, `2010` <dbl>, `2011` <dbl>,
## # `2012` <dbl>, `2013` <dbl>, `2014` <dbl>, `2015` <dbl>, `2016` <dbl>,
## # `2017` <dbl>, and abbreviated variable name 1: indicator
```

Si es que se desea excluir un grupo de valores indicados por un vector (algo similar a lo que se hace con `!=`), se debe usar también `%in%` pero junto con `!`, el cual se coloca antes de la columna seleccionada para indicar que el país no es ninguno de los valores dentro del vector:

```
world_bank_pop %>%
  filter(!country %in% c("MEX", "ESP", "TUR", "BRA", "CUB"))

## # A tibble: 1,036 x 20
##   country indic~1 `2000` `2001` `2002` `2003` `2004` `2005` `2006` `2007` ~
##   <chr>     <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 ABW      SP.URB~ 4.24e4 4.30e4 4.37e4 4.42e4 4.47e+4 4.49e+4 4.49e+4 4.47e+4
## 2 ABW      SP.URB~ 1.18e0 1.41e0 1.43e0 1.31e0 9.51e-1 4.91e-1 -1.78e-2 -4.35e-1
## 3 ABW      SP.POP~ 9.09e4 9.29e4 9.50e4 9.70e4 9.87e+4 1.00e+5 1.01e+5 1.01e+5
## 4 ABW      SP.POP~ 2.06e0 2.23e0 2.23e0 2.11e0 1.76e+0 1.30e+0 7.98e-1 3.84e-1
## 5 AFG      SP.URB~ 4.44e6 4.65e6 4.89e6 5.16e6 5.43e+6 5.69e+6 5.93e+6 6.15e+6
## 6 AFG      SP.URB~ 3.91e0 4.66e0 5.13e0 5.23e0 5.12e+0 4.77e+0 4.12e+0 3.65e+0
## 7 AFG      SP.POP~ 2.01e7 2.10e7 2.20e7 2.31e7 2.41e+7 2.51e+7 2.59e+7 2.66e+7
## 8 AFG      SP.POP~ 3.49e0 4.25e0 4.72e0 4.82e0 4.47e+0 3.87e+0 3.23e+0 2.76e+0
## 9 AGO      SP.URB~ 8.23e6 8.71e6 9.22e6 9.77e6 1.03e+7 1.09e+7 1.15e+7 1.21e+7
## 10 AGO     SP.URB~ 5.44e0 5.59e0 5.70e0 5.76e0 5.75e+0 5.69e+0 4.92e+0 4.89e+0
## # ... with 1,026 more rows, 10 more variables: `2008` <dbl>, `2009` <dbl>,
## # `2010` <dbl>, `2011` <dbl>, `2012` <dbl>, `2013` <dbl>, `2014` <dbl>,
## # `2015` <dbl>, `2016` <dbl>, `2017` <dbl>, and abbreviated variable name
## # 1: indicator
```

El uso de `filter()` con valores numéricos puede ser realizado con operadores que permitan elegir valores mayores o menores. El data frame `OrchardSprays` muestra la potencia de diferentes tratamientos como repelentes de abejas. La primera columna muestra la reducción de abejas de un total de 100, las siguientes dos muestran la posición de cada tratamiento en un arreglo de cuadro latino de 8x8 y la última corresponde al tratamiento usado.

```
str(OrchardSprays)

## 'data.frame': 64 obs. of 4 variables:
## $ decrease : num 57 95 8 69 92 90 15 2 84 6 ...
## $ rowpos   : num 1 2 3 4 5 6 7 8 1 2 ...
```

```
## $ colpos : num 1 1 1 1 1 1 1 1 2 2 ...
## $ treatment: Factor w/ 8 levels "A","B","C","D",...: 4 5 2 8 7 6 3 1 3 2 ...
```

En este caso, se pueden seleccionar únicamente las observaciones que muestren reducción de más de la mitad de las abejas, es decir aquellos que sean mayores a 50:

```
OrchardSprays %>%
  filter(decrease > 50)

##    decrease rowpos colpos treatment
## 1        57      1      1       D
## 2        95      2      1       E
## 3        69      4      1       H
## 4        92      5      1       G
## 5        90      6      1       F
## 6        84      1      2       C
## 7       127      3      2       H
## 8        51      5      2       E
## 9        69      7      2       F
## 10       71      8      2       G
## 11       87      1      3       F
## 12       72      2      3       H
## 13       72      7      3       G
## 14      130      1      4       H
## 15      114      3      4       E
## 16       51      8      4       D
## 17       60      3      5       G
## 18       81      7      5       H
## 19       71      8      5       F
## 20       77      4      6       G
## 21       76      8      6       H
## 22       72      2      7       G
## 23       57      4      7       F
## 24       81      6      7       H
## 25       61      8      7       E
## 26       80      1      8       G
## 27      114      2      8       F
## 28       86      5      8       H
## 29       55      6      8       E
```

En caso de que se desee establecer un rango entre dos valores es necesario establecer ambos límites usando `&`. Para seleccionar los valores entre 25 y 75 de la reducción de abejas se especifica:

```
OrchardSprays %>%
  filter(decrease >= 25 & decrease <= 75)

##    decrease rowpos colpos treatment
## 1        57      1      1       D
## 2        69      4      1       H
## 3        36      4      2       D
## 4        51      5      2       E
## 5        69      7      2       F
## 6        71      8      2       G
## 7        72      2      3       H
## 8        39      4      3       E
## 9        72      7      3       G
## 10       51      8      4       D
```

```

## 11      43      1      5      E
## 12      28      2      5      D
## 13      60      3      5      G
## 14      71      8      5      F
## 15      29      2      6      C
## 16      44      3      6      F
## 17      27      6      6      D
## 18      47      7      6      E
## 19      72      2      7      G
## 20      57      4      7      F
## 21      61      8      7      E
## 22      39      3      8      D
## 23      55      6      8      E

```

De este modo, combinando `filter()` y `select()`, el flujo de trabajo que permite crear un nuevo data frame se ve así:

```

OrchardSprays %>%
  select(decrease, treatment) %>%
  filter(treatment %in% c("A", "B", "C", "D"),
         decrease >= 50)

##   decrease treatment
## 1      57          D
## 2      84          C
## 3      51          D

```

Primero se seleccionan las columnas con `select()` y luego se aplican los criterios para crear el subconjunto.

4.1.4 Mutate

`mutate()` permite crear nuevas variables usando datos presentes en las variables del data frame, usualmente realizando una operación aritmética que transforma los datos y los coloca en una nueva columna. El data frame `storms` contiene datos de los huracanes registrados en el océano Atlántico desde 1975 hasta 2020:

```

str(storms)

## #tibble [11,859 x 13] (S3:tbl_df/tbl/data.frame)
## $ name           : chr [1:11859] "Amy" "Amy" "Amy" "Amy" ...
## $ year          : num [1:11859] 1975 1975 1975 1975 1975 ...
## $ month         : num [1:11859] 6 6 6 6 6 6 6 6 6 ...
## $ day            : int [1:11859] 27 27 27 27 28 28 28 28 29 ...
## $ hour           : num [1:11859] 0 6 12 18 0 6 12 18 0 6 ...
## $ lat            : num [1:11859] 27.5 28.5 29.5 30.5 31.5 32.4 33.3 34 34.4 34 ...
## $ long           : num [1:11859] -79 -79 -79 -79 -78.8 -78.7 -78 -77 -75.8 -74.8 ...
## $ status         : chr [1:11859] "tropical depression" "tropical depression" "tropical ...
## $ category       : Ord.factor w/ 7 levels "-1"<"0"<"1"<"2"<... 1 1 1 1 1 1 1 2 2 ...
## $ wind           : int [1:11859] 25 25 25 25 25 25 30 35 40 ...
## $ pressure       : int [1:11859] 1013 1013 1013 1013 1012 1012 1011 1006 1004 1002 ...
## $ tropicalstorm_force_diameter: int [1:11859] NA NA NA NA NA NA NA NA NA ...
## $ hurricane_force_diameter   : int [1:11859] NA NA NA NA NA NA NA NA NA ...

```

La columna `wind` muestra la velocidad máxima de viento registrada en nudos. Para convertir esta columna a otras unidades se puede usar `mutate()`, la cual creará una nueva columna para mostrar las nuevas unidades, preservando los dato originales. Tomando en cuenta que un nudo equivale a .514 m/s, la conversión únicamente requiere que se multipliquen los nudos por esta equivalencia. Primero se debe colocar el nombre de la nueva variable dentro de `mutate()` y después se indica la operación que se va a realizar:

```

storms %>%
  select(name, wind) %>%
  mutate(wind_ms = wind * .514)

## # A tibble: 11,859 x 3
##   name   wind wind_ms
##   <chr> <int>    <dbl>
## 1 Amy      25     12.8
## 2 Amy      25     12.8
## 3 Amy      25     12.8
## 4 Amy      25     12.8
## 5 Amy      25     12.8
## 6 Amy      25     12.8
## 7 Amy      25     12.8
## 8 Amy      30     15.4
## 9 Amy      35     18.0
## 10 Amy     40     20.6
## # ... with 11,849 more rows

```

Si no se usa `filter()` en el ejemplo anterior la conversión aún ocurrirá, solo que será difícil de ver debido a que las nuevas variables se añaden en el extremo derecho del data frame:

```

storms %>%
  mutate(wind_ms = wind * .514)

## # A tibble: 11,859 x 14
##   name   year month day hour   lat   long status categ~1  wind press~2
##   <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <chr>   <ord>   <int>   <int>
## 1 Amy    1975     6    27     0 27.5 -79  tropical dep~ -1      25   1013
## 2 Amy    1975     6    27     6 28.5 -79  tropical dep~ -1      25   1013
## 3 Amy    1975     6    27    12 29.5 -79  tropical dep~ -1      25   1013
## 4 Amy    1975     6    27    18 30.5 -79  tropical dep~ -1      25   1013
## 5 Amy    1975     6    28     0 31.5 -78.8 tropical dep~ -1      25   1012
## 6 Amy    1975     6    28     6 32.4 -78.7 tropical dep~ -1      25   1012
## 7 Amy    1975     6    28    12 33.3 -78  tropical dep~ -1      25   1011
## 8 Amy    1975     6    28    18 34   -77  tropical dep~ -1      30   1006
## 9 Amy    1975     6    29     0 34.4 -75.8 tropical sto~ 0      35   1004
## 10 Amy   1975     6    29     6 34   -74.8 tropical sto~ 0      40   1002
## # ... with 11,849 more rows, 3 more variables:
## #   tropicalstorm_force_diameter <int>, hurricane_force_diameter <int>,
## #   wind_ms <dbl>, and abbreviated variable names 1: category, 2: pressure

```

Su posición puede comprobarse usando `colnames()`:

```

storms %>%
  mutate(wind_ms = wind * .514) %>%
  colnames()

## [1] "name"                      "year"
## [3] "month"                     "day"
## [5] "hour"                      "lat"
## [7] "long"                      "status"
## [9] "category"                  "wind"
## [11] "pressure"                  "tropicalstorm_force_diameter"
## [13] "hurricane_force_diameter"  "wind_ms"

```

De la misma manera se pueden convertir las velocidades en m/s a km/h:

```

storms %>%
  select(name, wind) %>%
  mutate(wind_ms = wind * .514,
        winds_kh = (wind_ms * 36) / 10)

## # A tibble: 11,859 x 4
##   name    wind wind_ms winds_kh
##   <chr> <int>   <dbl>     <dbl>
## 1 Amy      25     12.8     46.3
## 2 Amy      25     12.8     46.3
## 3 Amy      25     12.8     46.3
## 4 Amy      25     12.8     46.3
## 5 Amy      25     12.8     46.3
## 6 Amy      25     12.8     46.3
## 7 Amy      25     12.8     46.3
## 8 Amy      30     15.4     55.5
## 9 Amy      35     18.0     64.8
## 10 Amy     40     20.6     74.0
## # ... with 11,849 more rows

```

En este ejemplo se puede ver que es posible crear más de una variable en la misma función, solo es necesario separar cada operación con comas. Incluso se pueden usar las variables creadas en la misma función para realizar operaciones, como fue el caso de `winds_kh`, en el cual se usaron los datos de `winds_ms`.

4.1.5 `group_by()` y `summarize()`

Este par de funciones permiten realizar operaciones por grupos, por ejemplo, si se desea conocer el promedio de las medidas de sépalo y tépalo de acuerdo con cada especie en el data frame `iris`. Si se utiliza la función `mean`, solo se puede obtener el promedio de todos los datos:

```

mean(iris$Sepal.Length)
## [1] 5.843333

```

Aplicar la función `mean()` por grupo implica clasificar cada observación de acuerdo con una categoría, en este caso el factor `Species` y realizar el cálculo de manera separada para cada categoría. Con `group_by()` se indica el nombre de la variable o variables que se van a utilizar para agrupar los datos. La función por si misma no tiene ningún efecto visible en el conjunto de datos seleccionado debido a que no se ha realizado ninguna operación:

```

iris %>%
  group_by(Species)

## # A tibble: 150 x 5
## # Groups:   Species [3]
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##   <dbl>       <dbl>       <dbl>       <dbl>   <fct>
## 1 5.1         3.5         1.4         0.2     setosa
## 2 4.9         3           1.4         0.2     setosa
## 3 4.7         3.2         1.3         0.2     setosa
## 4 4.6         3.1         1.5         0.2     setosa
## 5 5           3.6         1.4         0.2     setosa
## 6 5.4         3.9         1.7         0.4     setosa
## 7 4.6         3.4         1.4         0.3     setosa
## 8 5           3.4         1.5         0.2     setosa
## 9 4.4         2.9         1.4         0.2     setosa
## 10 4.9        3.1         1.5         0.1    setosa
## # ... with 140 more rows

```

Sin embargo, las observaciones ya se encuentran agrupadas, para consultar las variables de agrupamiento se usa `group_vars()`:

```
iris %>%
  group_by(Species) %>%
  group_vars()

## [1] "Species"
```

Si se desea remover el agrupamiento se usa `ungroup()`:

```
iris %>%
  group_by(Species) %>%
  ungroup() %>%
  group_vars()

## character(0)
```

Las operaciones que se desean realizar se especifican `summarise()` para indicar que deben llevarse a cabo por grupo. De manera similar a `mutate()`, se debe colocar el nombre de la nueva columna que resultará de la operación seguido de la función que se desea aplicar:

```
iris %>%
  group_by(Species) %>%
  summarise(longitud_sepalo = mean(Sepal.Length))

## # A tibble: 3 x 2
##   Species    longitud_sepalo
##   <fct>          <dbl>
## 1 setosa        5.01
## 2 versicolor    5.94
## 3 virginica     6.59
```

Como resultado se obtiene un nuevo data frame que únicamente tiene la variable de agrupamiento y la columna creada con `summarize()`.

Se pueden especificar más operaciones dentro de `summarize()`, solo se deben nombrar nuevas columnas con sus operaciones y separarlas con comas:

```
iris %>%
  group_by(Species) %>%
  summarise(longitud_sepalo = mean(Sepal.Length),
            ancho_sepalo = mean(Sepal.Width),
            longitud_petalo = mean(Petal.Length),
            ancho_petalo = mean(Petal.Width))

## # A tibble: 3 x 5
##   Species    longitud_sepalo ancho_sepalo longitud_petalo ancho_petalo
##   <fct>          <dbl>        <dbl>          <dbl>        <dbl>
## 1 setosa        5.01         3.43          1.46         0.246
## 2 versicolor    5.94         2.77          4.26         1.33
## 3 virginica     6.59         2.97          5.55         2.03
```

En un data frame con más factores es posible especificar más de una variable de agrupamiento. En el data frame `storms`, las observaciones para cada huracán corresponden a medidas tomadas cada seis horas a lo largo de su duración. Para conocer el promedio de velocidad de viento y presión total de cada huracán, es necesario agrupar las observaciones usando la variable `name` para juntar todos los datos correspondientes a cada tormenta:

```
storms %>%
  group_by(name) %>%
```

```

summarise(prom_vel_viento = mean(wind),
          prom_presion = mean(pressure))

## # A tibble: 214 x 3
##   name      prom_vel_viento prom_presion
##   <chr>          <dbl>        <dbl>
## 1 AL011993       27.5       1000.
## 2 AL012000        25         1009.
## 3 AL021992        29         1007.
## 4 AL021994       24.2       1016.
## 5 AL021999       28.8       1005.
## 6 AL022000       29.2       1009.
## 7 AL022001        25         1011
## 8 AL022003        30         1009.
## 9 AL022006        38         1002.
## 10 AL031987      21.2       1010.
## # ... with 204 more rows

```

Este data frame muestra el promedio de velocidad de viento y presión para toda la duración de cada tormenta. Si se desea más resolución sobre estos datos, por ejemplo, el promedio para cada día, se deben de especificar variables de agrupamiento adicionales:

```

storms %>%
  group_by(name, year, month, day) %>%
  summarise(prom_vel_viento = mean(wind),
            prom_presion = mean(pressure))

## `summarise()` has grouped output by 'name', 'year', 'month'. You can override
## using the ` `.groups` argument.

## # A tibble: 3,296 x 6
## # Groups:   name, year, month [613]
##   name      year month   day prom_vel_viento prom_presion
##   <chr>     <dbl> <dbl> <int>        <dbl>        <dbl>
## 1 AL011993  1993     5    31        25       1002.
## 2 AL011993  1993     6     1        27.5      1000.
## 3 AL011993  1993     6     2        30        999
## 4 AL012000  2000     6     7        25       1008
## 5 AL012000  2000     6     8        25       1010.
## 6 AL021992  1992     6    25        27.5      1008
## 7 AL021992  1992     6    26        30       1007
## 8 AL021994  1994     7    20        27.5      1016.
## 9 AL021994  1994     7    21        17.5      1016
## 10 AL021999 1999     7     2        30       1006
## # ... with 3,286 more rows

```

Es necesario colocar las variables año y mes para adecuadamente identificar cada observación y evitar que el agrupamiento ocurra incorrectamente. Por ejemplo, si un huracán tuviera una duración de varios meses, de tal manera que las observaciones del día 12 correspondieran a dos meses diferentes. Sin especificar `month` como variable de agrupación, no habría manera de separar dichas observaciones y el promedio no correspondería a un solo día, sino al mismo día del calendario de dos meses diferentes.

4.1.6 Exportar data frames y tibbles

Cuando se obtiene un objeto que se desea usar fuera de R puede ser exportado usando `write_csv()` del paquete `readr`. Esta función por defecto crea un archivo delimitado por comas, pero es posible usar otro carácter de separación usando `write_delim()`. En cualquier caso es necesario guardar primero el objeto en

el ambiente global, el cual se coloca en el argumento `x` y después se escribe el nombre del archivo en `file`. En este último argumento se puede colocar una carpeta dentro del directorio de trabajo para guardar el archivo. Usando el último objeto creado:

```
vel_y_presion <- storms %>%
  group_by(name, year, month, day) %>%
  summarise(prom_vel_viento = mean(wind),
            prom_presion = mean(pressure))

## `summarise()` has grouped output by 'name', 'year', 'month'. You can override
## using the ` `.groups` argument.

vel_y_presion

## # A tibble: 3,296 x 6
## # Groups:   name, year, month [613]
##   name      year month   day prom_vel_viento prom_presion
##   <chr>    <dbl> <dbl> <int>        <dbl>        <dbl>
## 1 AL011993  1993     5    31        25       1002.
## 2 AL011993  1993     6     1        27.5      1000.
## 3 AL011993  1993     6     2        30        999
## 4 AL012000  2000     6     7        25       1008
## 5 AL012000  2000     6     8        25       1010.
## 6 AL021992  1992     6    25        27.5      1008
## 7 AL021992  1992     6    26        30       1007
## 8 AL021994  1994     7    20        27.5      1016.
## 9 AL021994  1994     7    21        17.5      1016
## 10 AL021999 1999     7     2        30       1006
## # ... with 3,286 more rows

write_csv(x = vel_y_presion,
          file = "datos_manual/velocidad_y_presion.csv")
```

Es importante colocar la terminación `.csv`, para que el archivo sea exportado con la extensión, la función no la coloca automáticamente.

4.2 tidyr: datos largos y anchos

Este paquete permite manipular el arreglo de los datos en la estructura rectangular de un data frame. Las únicas dos funciones relevantes para los propósitos de este manual son `pivot_longer` y `pivot_wider`. Los nombres de ambas son bastante claros, ya que permiten reorganizar la estructura de los datos haciéndolos más largos o más anchos.

Los datos con formato largo son aquellos que en las cuales cada fila representa una observación y cada columna una variable que describe propiedades de esa observación. Sin embargo, no es la única manera de organizar datos.

El siguiente data frame contiene los conteos de semillas de frutos crecidos en invernadero y otros colectados en campo:

```
semillas <- read_csv(file = "datos_manual/semillas.csv",
                      col_types = "ii")
semillas

## # A tibble: 50 x 2
##   invernadero campo
##   <int>     <int>
## 1 32         24
## 2 34         33
```

```

## 3      39    30
## 4      44    25
## 5      28    28
## 6      24    23
## 7      29    23
## 8      39    28
## 9      26    23
## 10     42    28
## # ... with 40 more rows

```

Los datos se encuentran en formato ancho porque ambas columnas tienen datos de la misma variable: número de semillas. Cuando una variable está distribuida en más de una columna los datos son anchos, para hacerlos largos es necesario juntar los datos de la misma variable en una sola columna y al mismo tiempo crear otra que describa cada observación. La función `pivot_longer()` permite reorganizar el data frame para que tenga formato largo.

Para elegir las columnas que serán utilizadas para la transformación se utiliza el argumento `cols`. En `names_to` y `values_to` se escriben los nombres de las nuevas columnas que serán creadas. `names_to` toma el nombre de las columnas indicadas en `cols` para que cada uno de los valores numéricos tenga asociado `invernadero` o `campo`, según le corresponda, en este caso un nombre adecuado para la nueva columna con el lugar de procedencia del conteo es “sitio”. Los valores numéricos se colocarán en la columna `values_to`, la cual llevará el nombre de “`num_semillas`”:

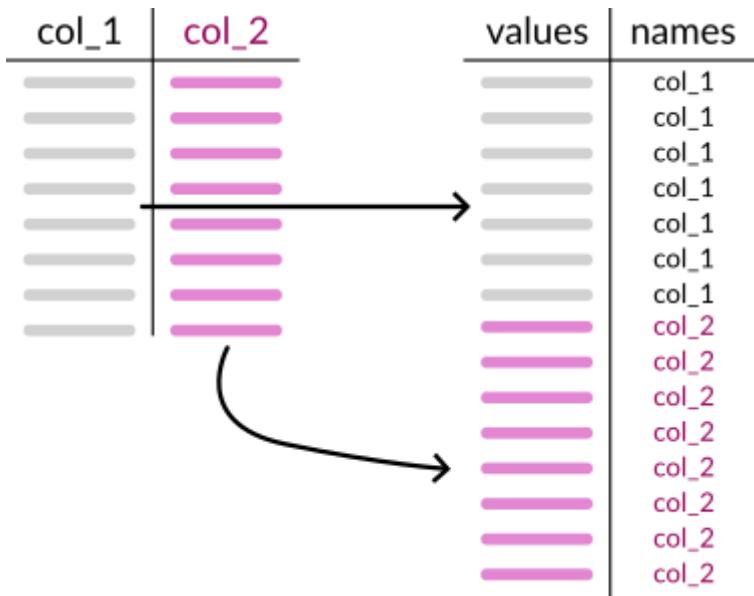
```

semillas %>%
  pivot_longer(cols = c(invernadero, campo),
               names_to = "sitio",
               values_to = "num_semillas")

## # A tibble: 100 x 2
##       sitio      num_semillas
##   <chr>           <int>
## 1 invernadero      32
## 2 campo            24
## 3 invernadero      34
## 4 campo            33
## 5 invernadero      39
## 6 campo            30
## 7 invernadero      44
## 8 campo            25
## 9 invernadero      28
## 10 campo           28
## # ... with 90 more rows

```

Los valores de `names_to` y `num_semillas` deben estar escritos entre comillas debido a que no son parte del data frame original. El proceso realizado por `pivot_longer()` se muestra en la siguiente figura:



Usualmente es necesario usar `pivot_longer()` para ajustar datos externos a la estructura necesaria en R. Sin embargo, en ocasiones se debe hacer la operación contraria para llevar a cabo pruebas estadísticas o porque una estructura ancha es una manera más conveniente de mostrar la información. `pivot_wider()` funciona de manera similar a `pivot_longer()`, pero al revés. El siguiente data frame contiene los conteos de tres tipos de trampas de insectos para tres sitios, debido a que todos los datos relacionados con el conteo están en la misma columna, los datos tienen un formato largo:

```
insectos <- read_csv(file = "datos_manual/insectos.csv",
                      col_types = "fhi")
insectos

## # A tibble: 9 x 3
##   sitio trampa conteo
##   <fct> <fct>   <int>
## 1 A     rojo      21
## 2 A     azul      23
## 3 A     amarillo  23
## 4 B     rojo      25
## 5 B     azul      28
## 6 B     amarillo  24
## 7 C     rojo      29
## 8 C     azul      25
## 9 C     amarillo  19
```

`pivot_wider()` requiere los argumentos `names_from` y `values_from`. El primero toma los valores de un vector de carácter para crear una columna a partir de cada valor único, en cada una de estas se acomodan los valores de la columna seleccionada con `values_from`. Para organizar los datos de acuerdo con el tipo de trampa, se debe usar el siguiente código:

```
insectos %>%
  pivot_wider(names_from = trampa, values_from = conteo)

## # A tibble: 3 x 4
##   sitio rojo azul amarillo
##   <fct> <int> <int>   <int>
## 1 A       21    23     23
## 2 B       25    28     24
```

```
## 3 C      29      25      19
```

De igual manera se pueden acomodar de acuerdo con el sitio:

```
insectos %>%
  pivot_wider(names_from = sitio, values_from = conteo)

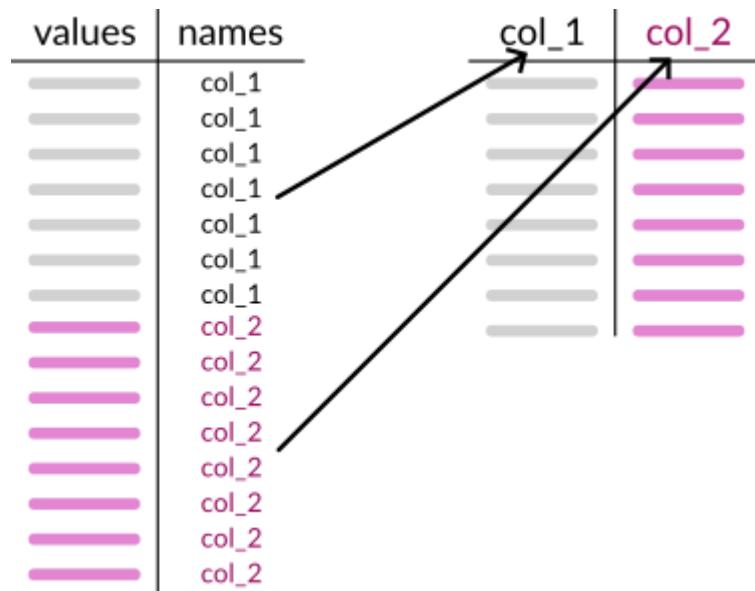
## # A tibble: 3 x 4
##   trampa     A     B     C
##   <fct>    <int> <int> <int>
## 1 rojo        21     25    29
## 2 azul        23     28    25
## 3 amarillo   23     24    19
```

O usando el sitio y tipo de trampa:

```
insectos %>%
  pivot_wider(names_from = c(trampa, sitio), values_from = conteo)

## # A tibble: 1 x 9
##   rojo_A azul_A amarillo_A rojo_B azul_B amarillo_B rojo_C azul_C amarillo_C
##   <int>   <int>   <int>   <int>   <int>   <int>   <int>   <int>
## 1     21     23     23     25     28     24     29     25     19
```

El proceso realizado es opuesto a `pivot_longer()`:



Sin embargo, la función como se mostró anteriormente solo sirve cuando cada una de las observaciones puede ser identificada como una combinación única de sus variables. En el siguiente data frame se muestran las longitudes de la cola de lagartijas provenientes de dos islas:

```
lagartijas <- read_csv(file = "datos_manual/lagartijas.csv",
                       col_types = "nf")
lagartijas

## # A tibble: 60 x 2
##   largo_cola isla
##   <dbl>     <fct>
## 1 3.13     norte
## 2 2.81     norte
## 3 3.69     norte
```

```

## 4      2.81 norte
## 5      1.97 norte
## 6      3.56 norte
## 7      2.83 norte
## 8      3.34 norte
## 9      2.92 norte
## 10     3.28 norte
## # ... with 50 more rows

```

Los datos están en formato largo y si se usa `pivot_wider()` como se describió anteriormente solo se muestra un error:

```

lagartijas %>%
  pivot_wider(names_from = isla, values_from = largo_col)

## Warning: Values from `largo_col` are not uniquely identified; output will contain list-cols.
## * Use `values_fn = list` to suppress this warning.
## * Use `values_fn = {summary_fun}` to summarise duplicates.
## * Use the following dplyr code to identify duplicates.
##   {data} %>%
##     dplyr::group_by(isla) %>%
##     dplyr::summarise(n = dplyr::n(), .groups = "drop") %>%
##     dplyr::filter(n > 1L)

## # A tibble: 1 x 2
##   norte     este
##   <list>    <list>
## 1 <dbl [30]> <dbl [30]>

```

La advertencia indica que las observaciones contienen valores duplicados y no pueden ser identificados adecuadamente para realizar la operación. En el caso de `insectos`, cada observación era única porque estaban conformadas por combinaciones de `sitio` y `trampa` que no se repetían. En el caso de `lagartijas`, los nombres de las islas se repiten múltiples veces y las observaciones no son una combinación única de variables. Para usar `pivot_wider()` es necesario crear una nueva variable que haga a cada observación única, no tiene que ser nada relacionado con los datos, puede ser solamente una enumeración de las filas.

```

lagartijas %>%
  mutate(fila = row_number())

## # A tibble: 60 x 3
##   largo_col isla    fila
##   <dbl>    <fct> <int>
## 1 3.13 norte 1
## 2 2.81 norte 2
## 3 3.69 norte 3
## 4 2.81 norte 4
## 5 1.97 norte 5
## 6 3.56 norte 6
## 7 2.83 norte 7
## 8 3.34 norte 8
## 9 2.92 norte 9
## 10 3.28 norte 10
## # ... with 50 more rows

```

Sin embargo, para que la operación funcione adecuadamente es necesario primero agrupar los datos con `group_by()` con la variable que se usará en `names_from`:

```
lagartijas %>%
```

```

group_by(isla) %>%
  mutate(filas = row_number()) %>%
  pivot_wider(names_from = isla, values_from = largo_col)

## # A tibble: 30 x 3
##   filas norte este
##   <int> <dbl> <dbl>
## 1     1   3.13  5.37
## 2     2   2.81  4.73
## 3     3   3.69  4.76
## 4     4   2.81  4.8 
## 5     5   1.97  5.11
## 6     6   3.56  4.59
## 7     7   2.83  5.37
## 8     8   3.34  5.15
## 9     9   2.92  4.92
## 10    10   3.28  4.63
## # ... with 20 more rows

```

El uso de ambas funciones es complicado al inicio, pero la capacidad de transformar los data frames de este modo es de gran ayuda para evitar realizar una conversión manual que es propensa a errores y que toma mucho tiempo.

5 Gráficas con ggplot2

`ggplot2` es una librería para realizar gráficas. Si bien el paquete `graphics` que está presente por defecto en cada sesión de R cuenta con métodos para hacer gráficas, `ggplot2` es una alternativa más moderna y con una sintaxis sencilla y adaptable que puede usarse para situaciones complejas. La paquetería provee funciones para manipular los datos y sus atributos visuales (aesthetics) como el color, forma, tamaño, etc. a través de capas.

5.1 Primeros pasos

Para realizar una gráfica es necesario tener los datos a utilizar guardados como un objeto, el cual se introduce a la función `ggplot()`, con el argumento `data`. Después es necesario establecer las atributos visuales o estéticas (aesthetics). Las estéticas que siempre deben especificarse son las variables del data frame que van a formar los ejes x y y en la gráfica.

Tomando como ejemplo el conjunto de datos `iris`:

```

str(iris)

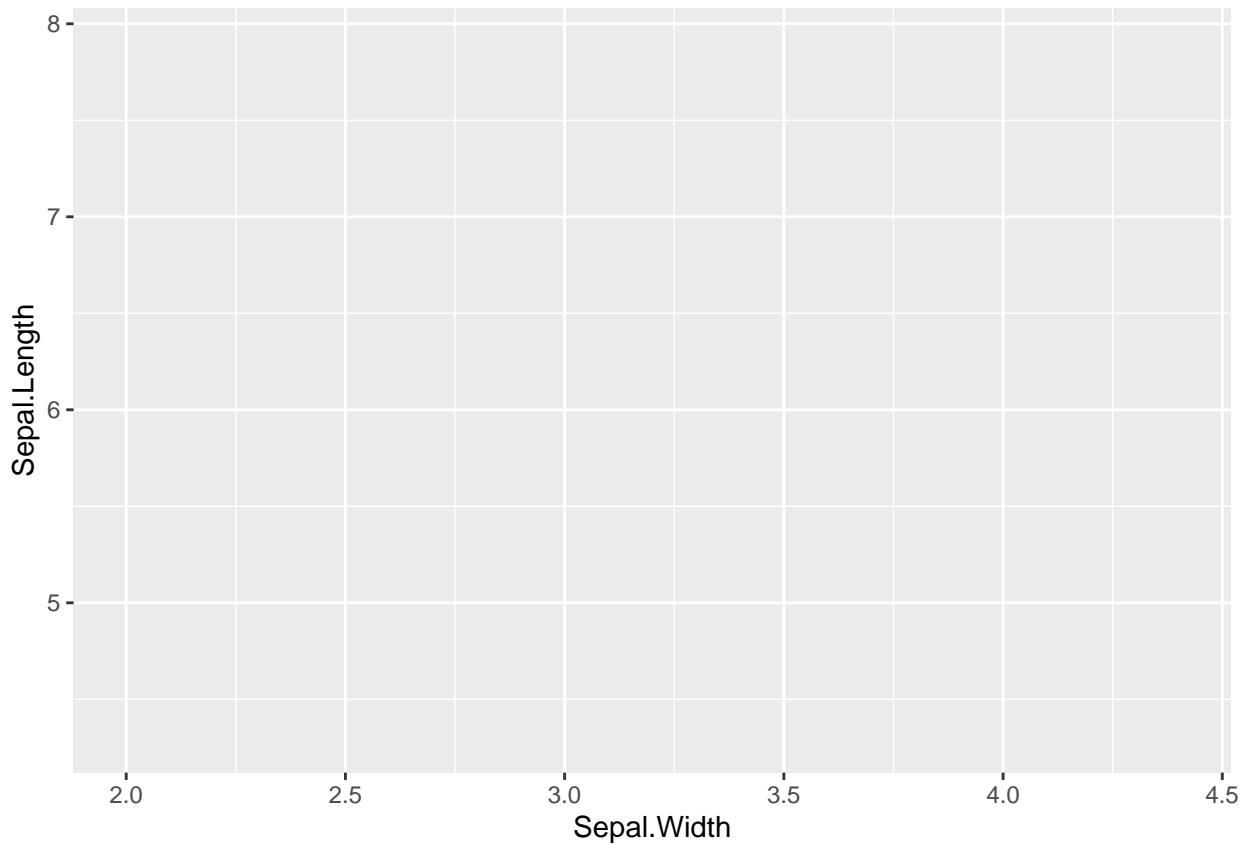
## 'data.frame':   150 obs. of  5 variables:
## $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
## $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
## $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
## $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
## $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...

```

Usando las variables numéricas se puede realizar una gráfica de dispersión, colocando la longitud del sépalo en función de su ancho, de modo que cada observación será representada por un punto que representa la intersección de ambos valores.

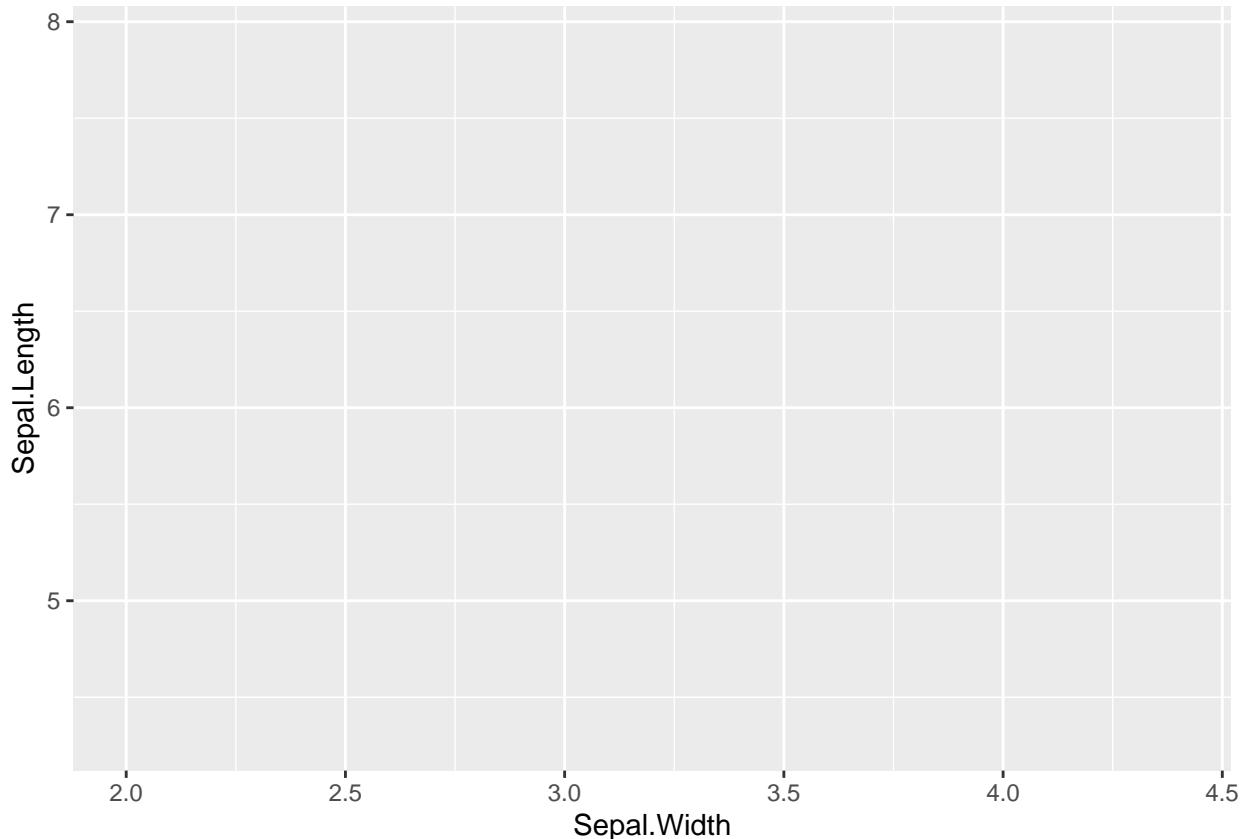
Como se mencionó anteriormente, se debe colocar el nombre del objeto en `data`, posteriormente se utiliza el argumento `mapping` para establecer las estéticas con la función `aes()` (aesthetics), dentro de la cual se colocan los valores de x y y. Se deben utilizar únicamente los nombres tal cual como aparecen en `str(iris)` o en `colnames(iris)`, no es necesario utilizar el operador `$` para que `ggplot()` identifique los nombres como columnas del data frame:

```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length))
```



Debido a que todas las gráficas creadas con `ggplot()` usan una sintaxis similar, es común escribirla sin especificar los argumentos:

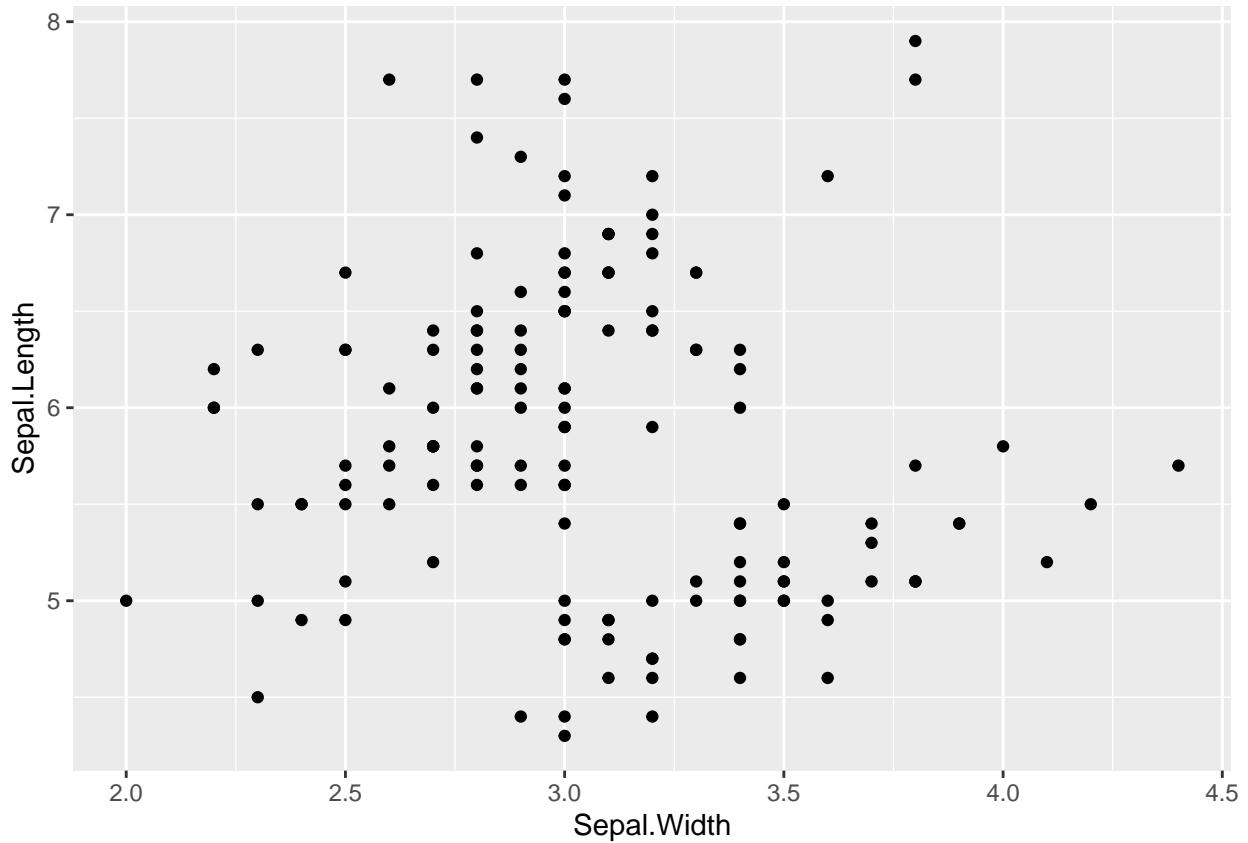
```
ggplot(iris, aes(Sepal.Width, Sepal.Length))
```



En este manual solo se omitirá el uso de `mapping`, pero se mantendrán los demás nombres para priorizar la claridad del código.

Al correr ese primer comando no aparecen los datos, únicamente el espacio en el cual se van a mostrar las observaciones una vez que se especifiquen las capas. Debido a que buscamos hacer una gráfica de dispersión, la función a utilizarse es `geom_point()`. Se coloca por fuera de `ggplot()` y para indicar que ambas funciones deben ser interpretadas juntas se coloca `+` al finalizar `ggplot()` y antes de empezar `geom_point()`. Usualmente se colocan en líneas diferentes para mayor claridad y en RStudio, todas las funciones por debajo de `ggplot()` tienen un ligero desplazamiento hacia la derecha:

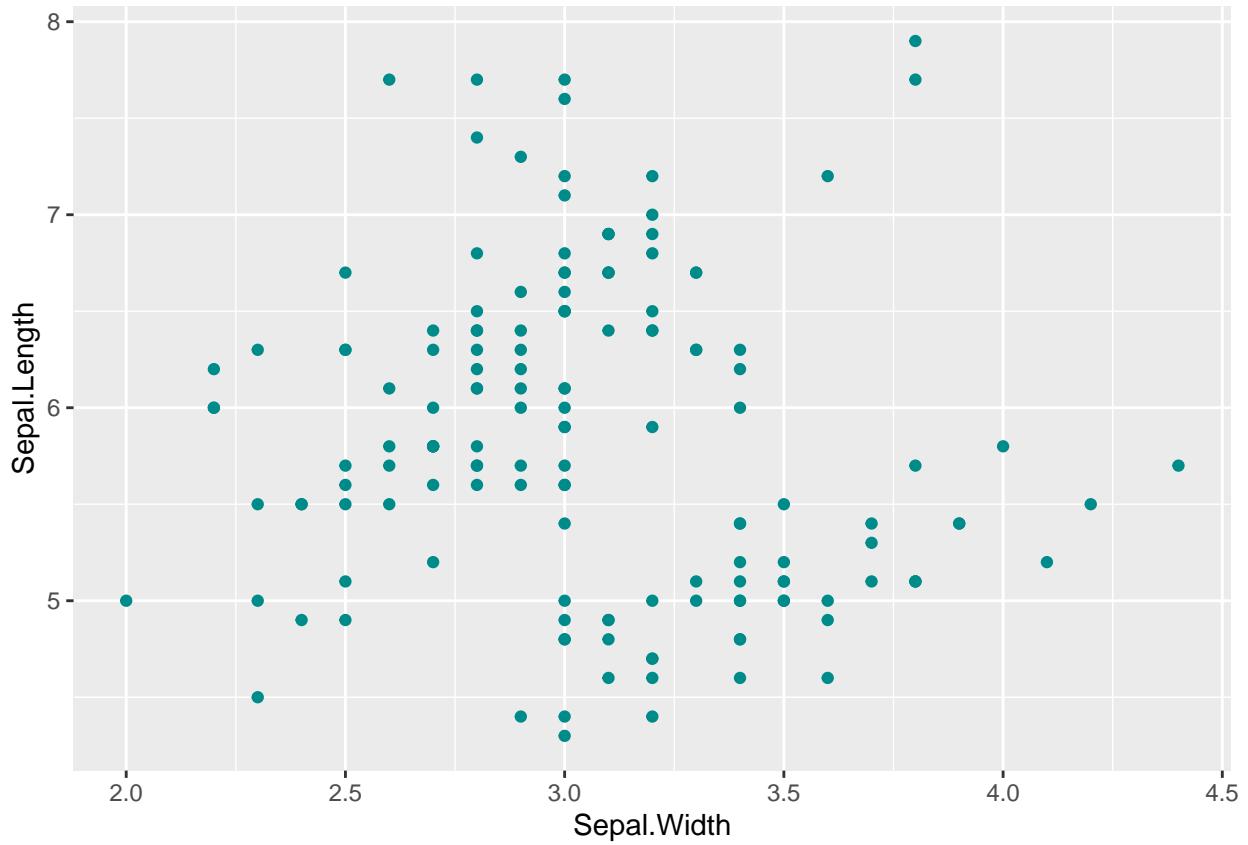
```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length)) +
  geom_point()
```



5.2 Estéticas

`geom_point()` puede ser configurada mediante sus argumentos. Por ejemplo, para cambiar el color de todos los puntos se utiliza `color`:

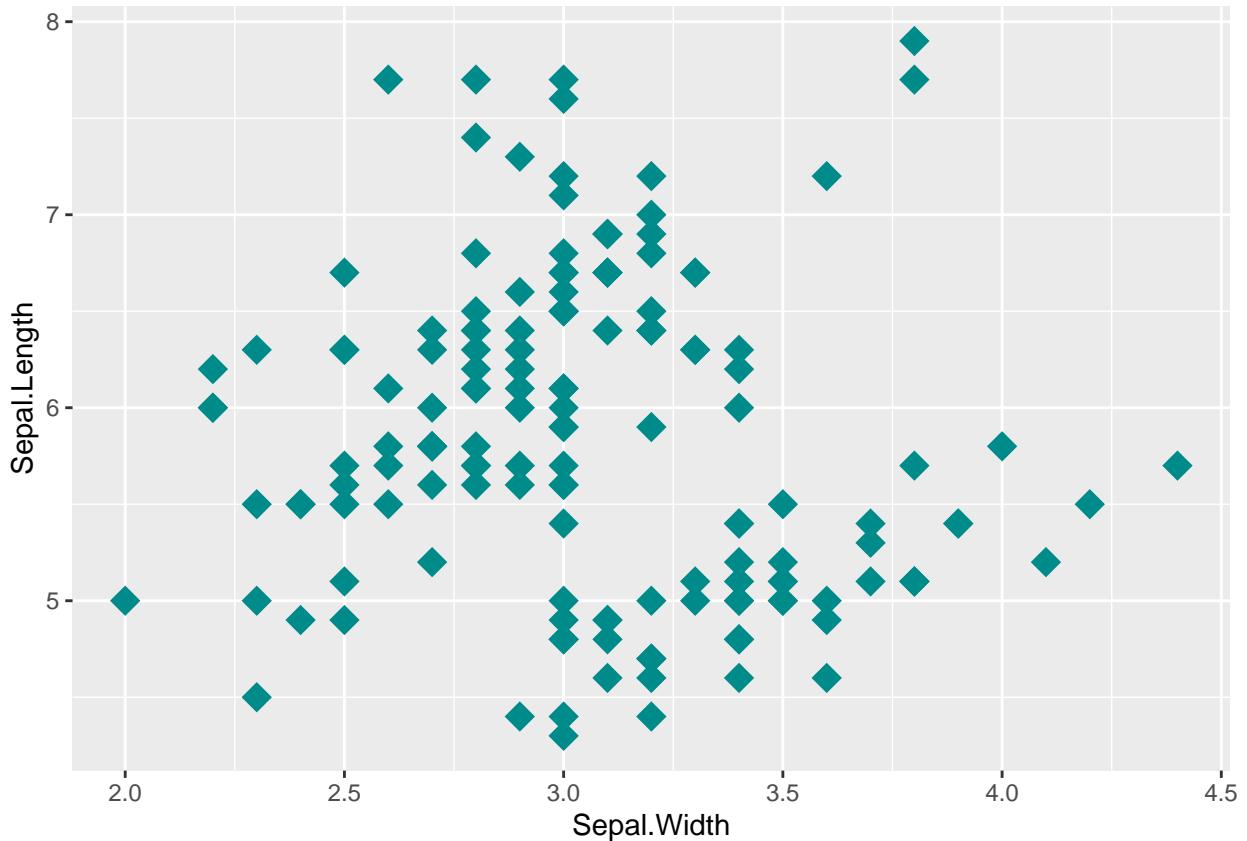
```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length)) +  
  geom_point(color = "cyan4")
```



El color de los puntos también puede ser especificado mediante `colour` (inglés británico) o `col` (termino de la paquetería básica).

El tamaño y tipo de carácter son modificados con `size` y `pch`, respectivamente.

```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length)) +
  geom_point(col = "cyan4", pch = 18, size = 5)
```



El tipo de carácter (plot character (pch)) se elige mediante números del 0 al 25, los cuales se muestran aquí y también pueden ser consultados en ?pch:

```
caracteres <- tibble(x = 0:25,
                      y = rep(x = 1, times = 26),
                      pch = 0:25)

ggplot(data = caracteres, aes(x = x, y = y)) +
  geom_point(pch = 0:25, size = 5) +
  geom_text(aes(label = pch), nudge_y = .3,
            color = "firebrick") +
  ylim(0, 2.3) +
  theme_void()

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

□ ○ △ + × ◇ ▽ □ * ◆ ⊕ △ □ ■ ● ▲ ♦ ● ○ □ ◇ △ ▽

Todos los caracteres se ven afectados por color:

```
ggplot(data = caracteres, aes(x = x, y = y)) +
  geom_point(pch = 0:25, size = 5, color = "purple") +
  geom_text(aes(label = pch), nudge_y = .3,
            color = "firebrick") +
  ylim(0, 2.3) +
  theme_void()
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

□ ○ △ + × ◇ ▽ □ * △ □ □ □ ■ ● ▲ ◆ ● ○ □ ◇ △ ▽

Otra estética de estos caracteres puede ser `fill`, que determina el color de relleno. Sin embargo, únicamente los caracteres 21 a 25 son afectados por esta estética:

```
ggplot(data = caracteres, aes(x = x, y = y)) +
  geom_point(pch = 0:25, size = 5, color = "purple", fill = "orange") +
  geom_text(aes(label = pch), nudge_y = .3,
            color = "firebrick") +
  ylim(0, 2.3) +
  theme_void()
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

□ ○ △ + × ◇ ▽ □ * △ □ □ □ ■ ● ▲ ◆ ● ○ ○ □ ◇ △ ▽

Al utilizar un carácter que carece de relleno, no se aplicará ningún color al usar `fill`:

```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length)) +
  geom_point(pch = 5, fill = "purple", size = 5)
```



Para que el color sea aplicado al contorno se debe colocar en `color`:

```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length)) +
  geom_point(pch = 5, color = "purple", size = 5)
```



Estos ejemplos anteriores solo son una manera de mostrar las diferentes opciones de `geom_point()`, ya que las gráficas anteriores no son necesariamente útiles.

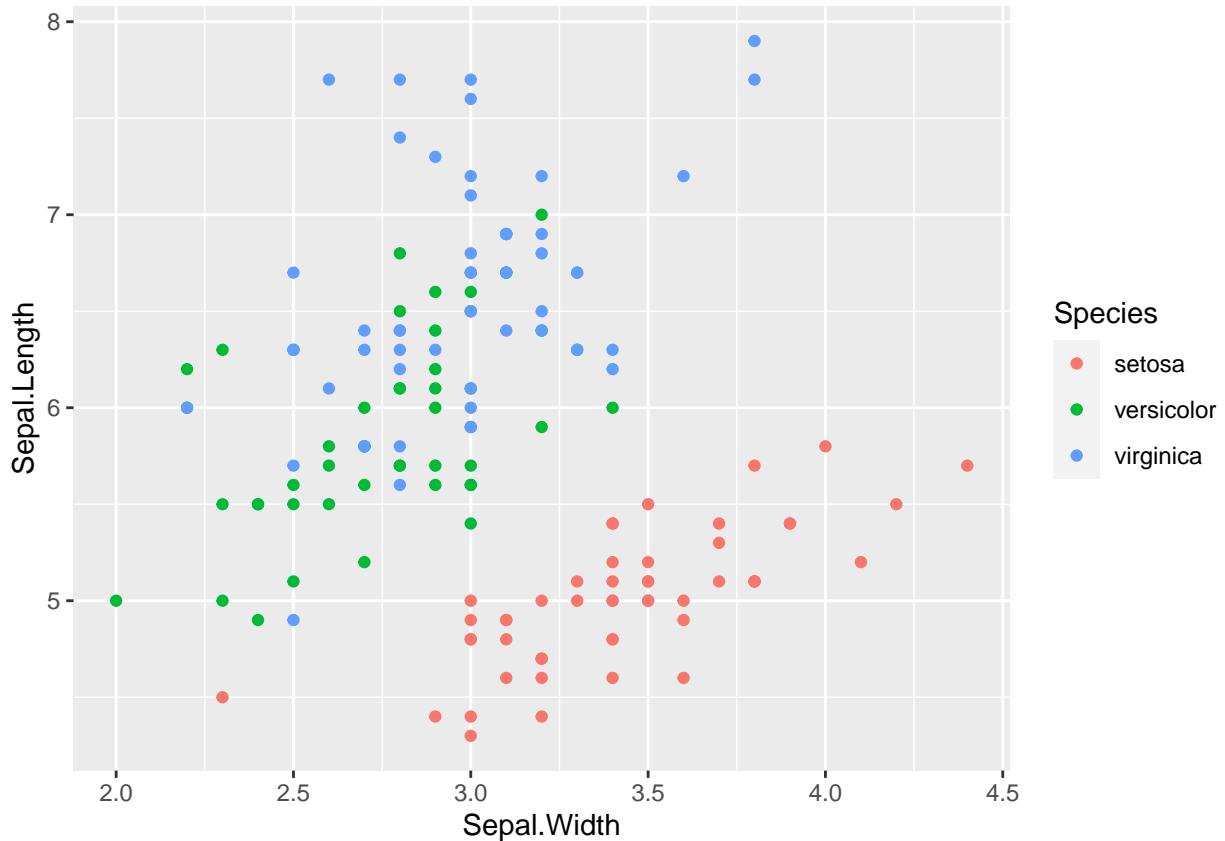
Volviendo a la estructura de los datos, podemos observar que `iris` cuenta con una variable categórica que indica la especie de cada una de las observaciones:

```
str(iris)

## 'data.frame':    150 obs. of  5 variables:
## $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
## $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
## $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
## $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
## $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...
```

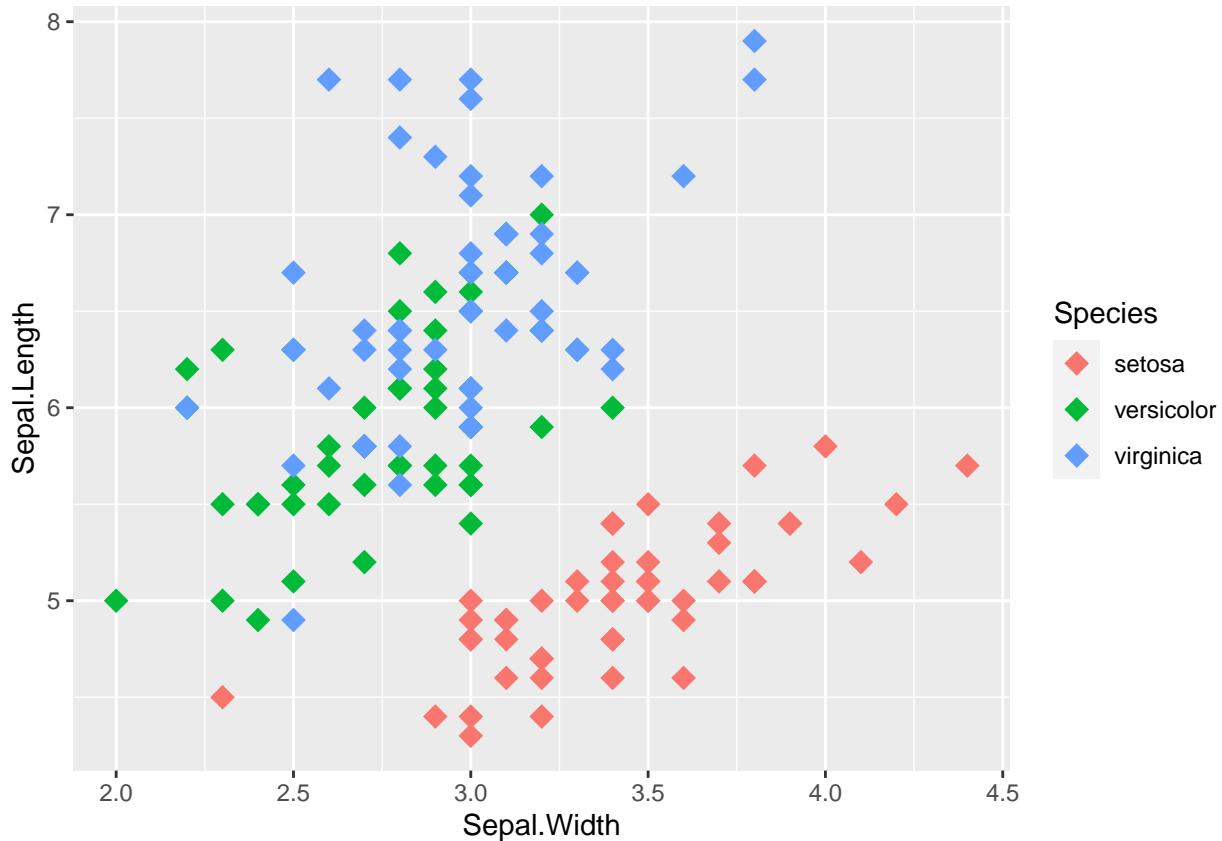
Para representar esta variable dentro de la gráfica de dispersión se puede usar uno de los atributos gráficos, una de las estéticas. Ello se especifica dentro de `aes()`, indicando qué estética va a representar a la variable `Species`, por ejemplo el color:

```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length,
                                    color = Species)) +
  geom_point()
```



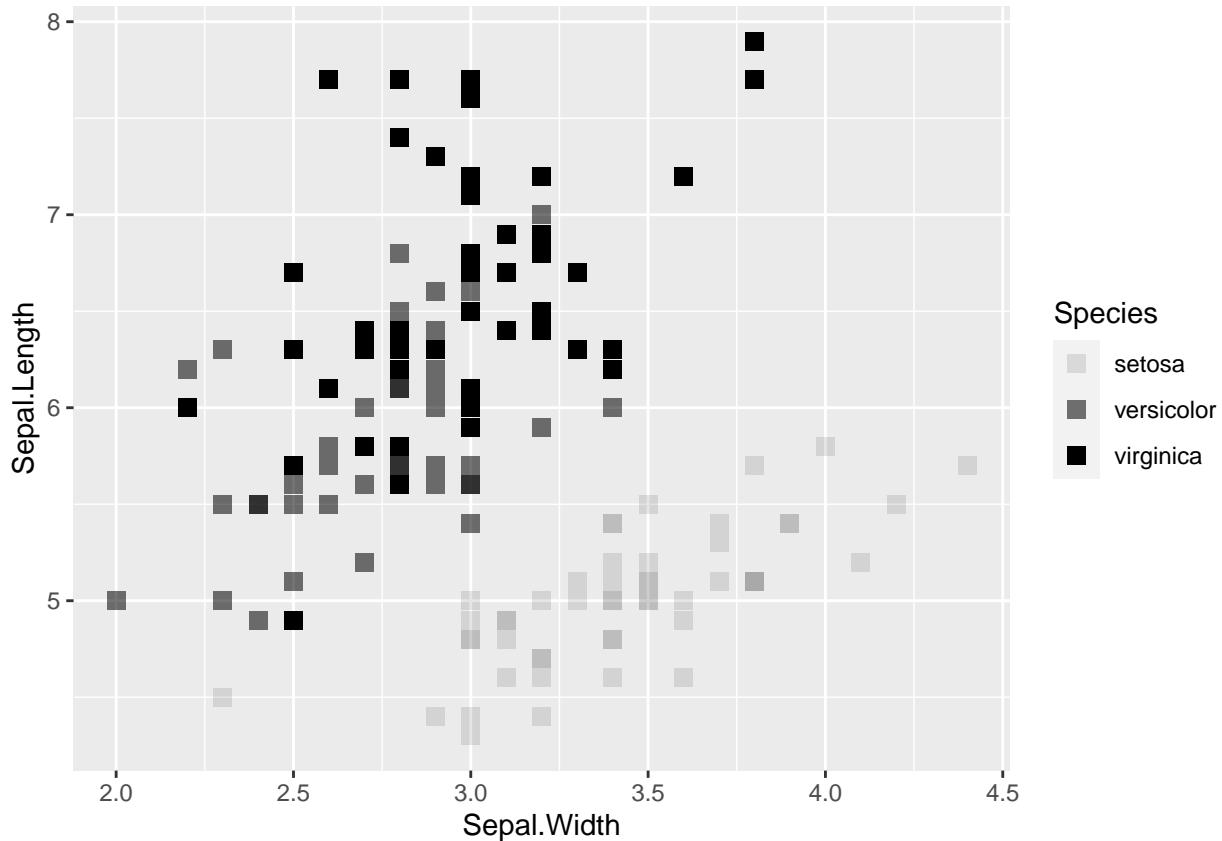
Como resultado se obtiene una gráfica más informativa, debido a que muestra tres variables en lugar de las dos ubicadas en los ejes. Los argumentos de `geom_point()` se utilizan para cambiar atributos que no están asociados a ninguna variable, como el tamaño y el tipo de carácter:

```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length,
                                    col = Species)) +
  geom_point(pch = 18, size = 4)
```



Dependiendo de lo que se desea comunicar con la gráfica, el color puede o no representar una variable propia de los datos y eso determina si se modifica en `aes()` o en `geom_point()`. Cabe mencionar que no todas las gráficas que pueden realizarse con ggplot2 son necesariamente útiles, por ejemplo, si se desea asociar la especie de acuerdo con la transparencia de cada punto se obtiene una gráfica que no es muy fácil de interpretar:

```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length,
                                   alpha = Species)) +
  geom_point(pch = 15, size = 3)
## Warning: Using alpha for a discrete variable is not advised.
```

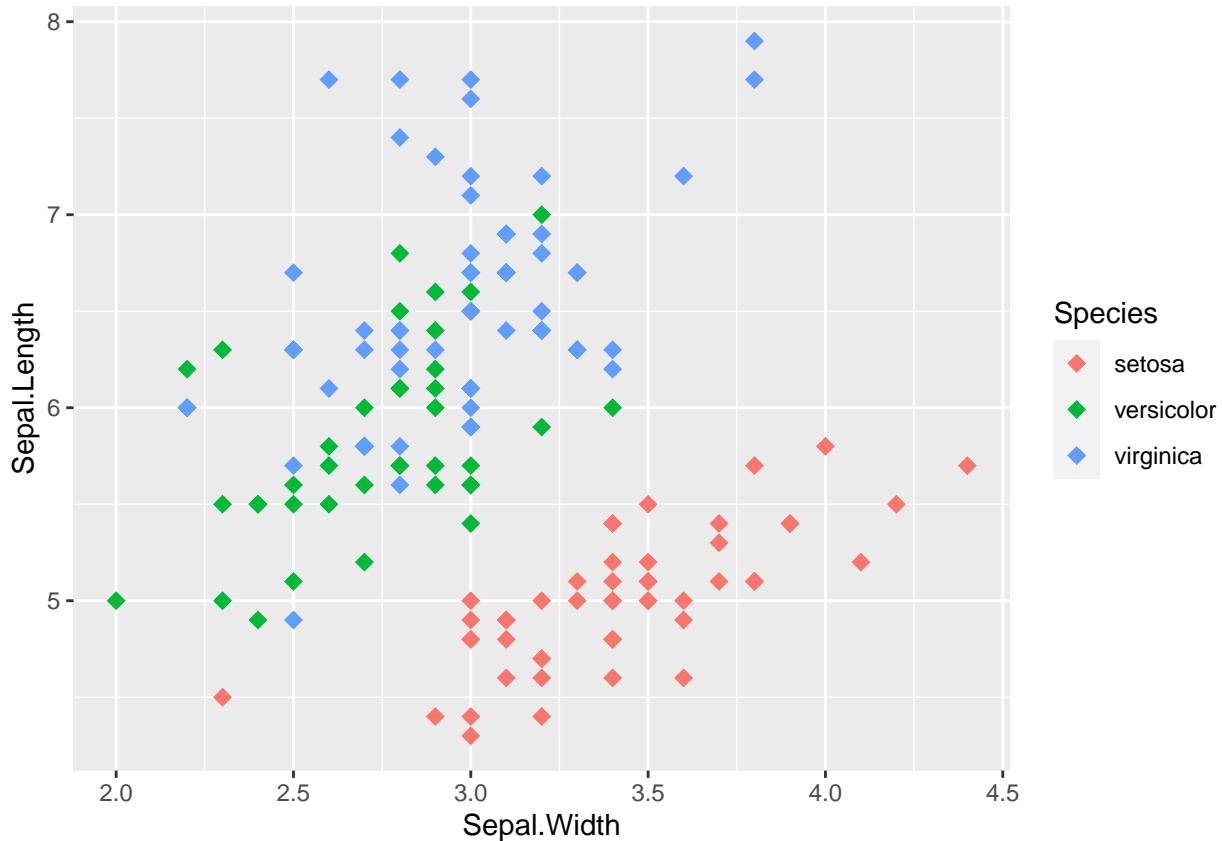


Las estéticas disponibles además de `x` y `y` son:

- `alpha`: opacidad del color
- `color`: color de contorno
- `fill`: color de relleno
- `shape / pch`: tipo de carácter
- `size`: tamaño de carácter, línea, etc.
- `stroke / linetype`: tipo de línea

Las opciones globales se especifican en `ggplot()`, aquellas que serán usadas para todas las capas de la gráfica. Sin embargo, es posible indicar un conjunto de datos y las estéticas dentro de `geom_point()`. Esto es útil cuando los datos que se van a graficar están distribuidos en más de un data frame. Para cada capa se puede especificar un conjunto de datos diferente que únicamente será usado para esa capa. Solo es importante tener en cuenta que las estéticas colocadas dentro de `aes()` serán representadas de acuerdo con una variable, ya sea que se encuentren en `ggplot()` o en alguna de las capas:

```
ggplot() +
  geom_point(data = iris, aes(x = Sepal.Width, y = Sepal.Length,
                               color = Species), pch = 18, size = 3)
```



A través de conocer las distintas opciones disponibles se pueden crear gráficas útiles para cualquier situación y tipo de datos. A continuación se mencionan algunos aspectos generales sobre la apariencia de las gráficas antes de introducir los distintos tipos disponibles en `ggplot2`.

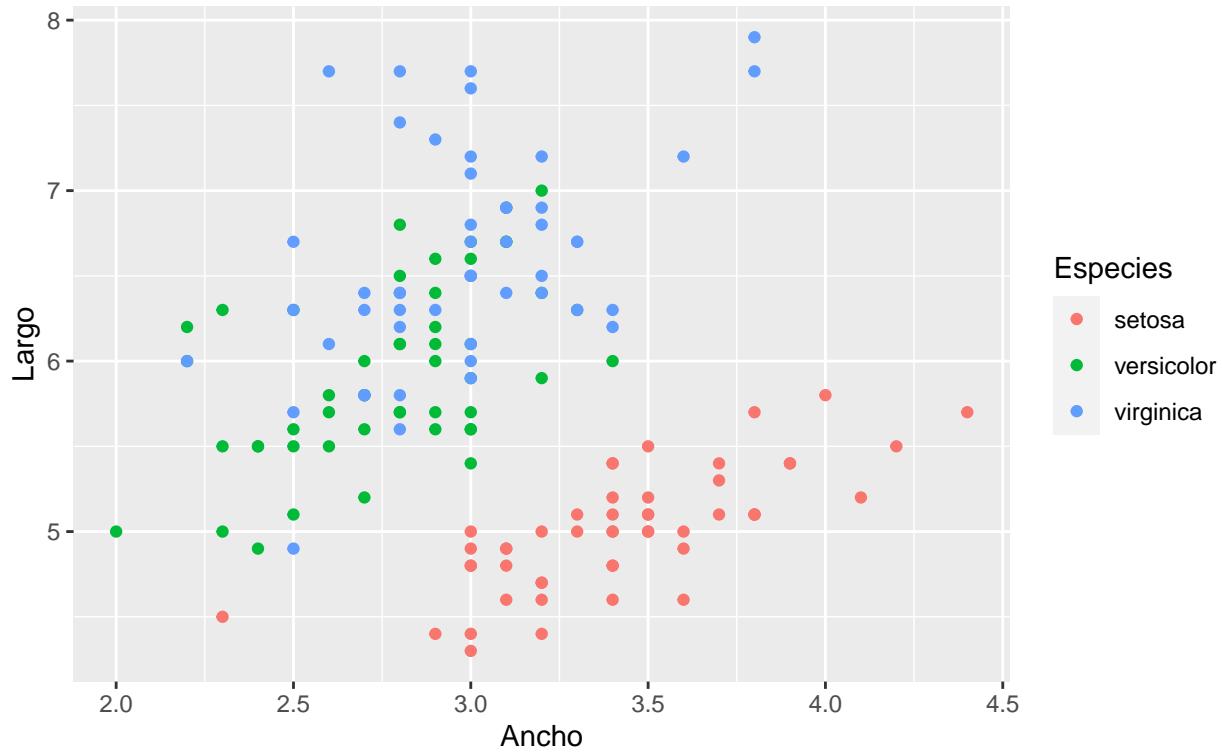
5.3 Ejes

Los nombres de los ejes se modifican con la capa `labs()`, mediante la cual también se pueden cambiar el título, subtítulo y el título de la leyenda:

```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length,
                                    color = Species)) +
  geom_point() +
  labs(title = "Iris",
       x = "Ancho",
       y = "Largo",
       subtitle = "Ancho VS Largo de sépalo",
       color = "Especies")
```

Iris

Ancho VS Largo de sépalo



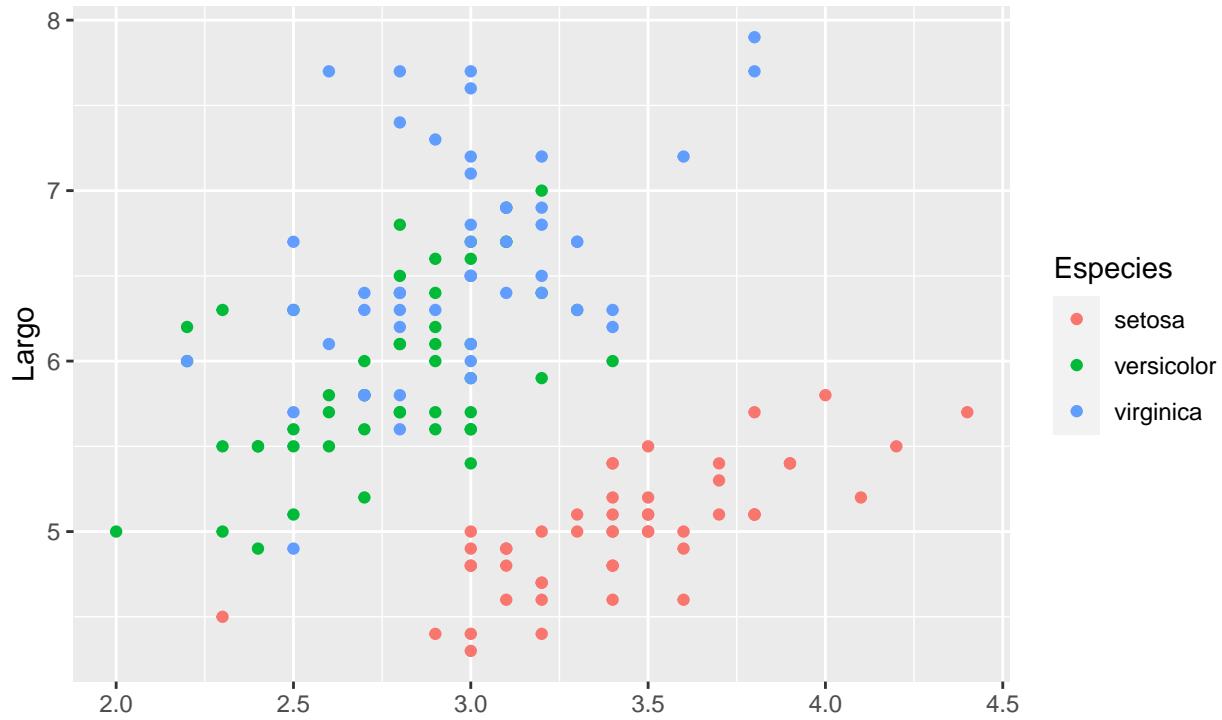
Para modificar el título de la leyenda es necesario colocar la estética con la que se asoció la variable representada, en este caso `color`.

Si por alguna razón se quiere dejar alguno de los ejes sin nombre, se puede colocar únicamente un espacio:

```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length,
                                    col = Species)) +
  geom_point() +
  labs(title = "Iris",
       x = " ",
       y = "Largo",
       subtitle = "Ancho VS Largo de sépalo",
       col = "Especies")
```

Iris

Ancho VS Largo de sépalo

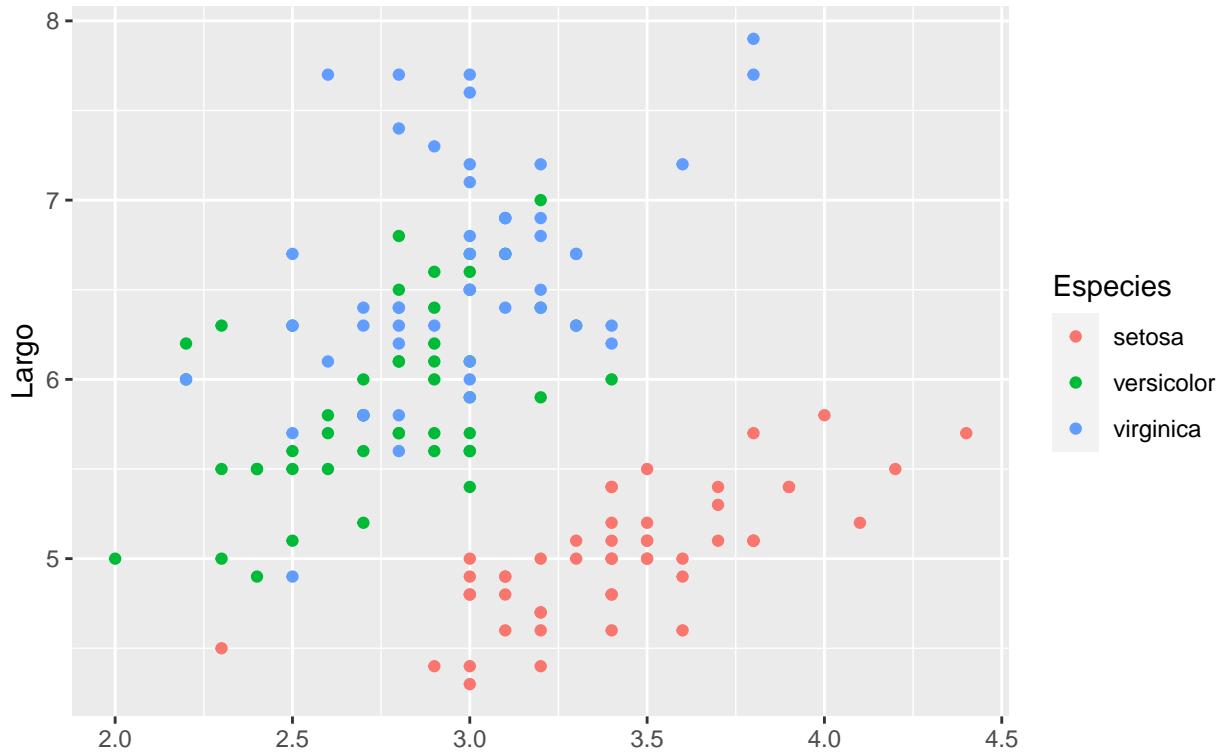


Esto también se puede hacer con NULL, lo cual elimina el espacio reservado para el eje:

```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length,
                                    col = Species)) +
  geom_point() +
  labs(title = "Iris",
       x = NULL,
       y = "Largo",
       subtitle = "Ancho VS Largo de sépalo",
       col = "Especies")
```

Iris

Ancho VS Largo de sépalo



La escala de los ejes continuos puede ser modificada con `scale_x_continuous()` y `scale_y_continuous()`, usando el argumento `breaks`, el cual requiere un vector que indique las posiciones que se desean marcar en el eje. La manera más sencilla de hacerlo es mediante la función `seq()`:

```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length,
                                    color = Species)) +
  geom_point() +
  labs(title = "Iris",
       x = " ",
       y = "Largo",
       subtitle = "Ancho VS Largo de sépalo",
       color = "Especies") +
  scale_x_continuous(breaks = seq(from = 2, to = 4.5, by = .2)) +
  scale_y_continuous(breaks = seq(from = 4, to = 9, by = .5))
```

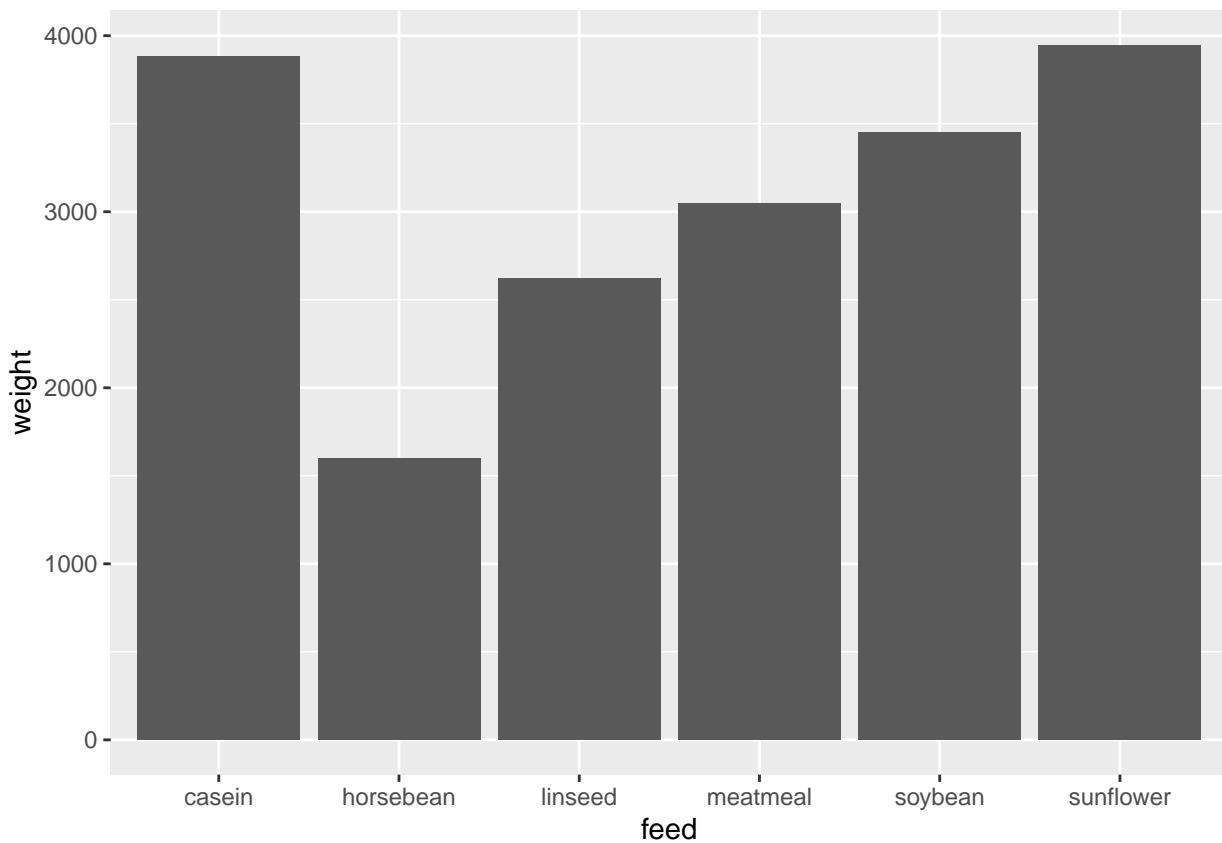
Iris

Ancho VS Largo de sépalo



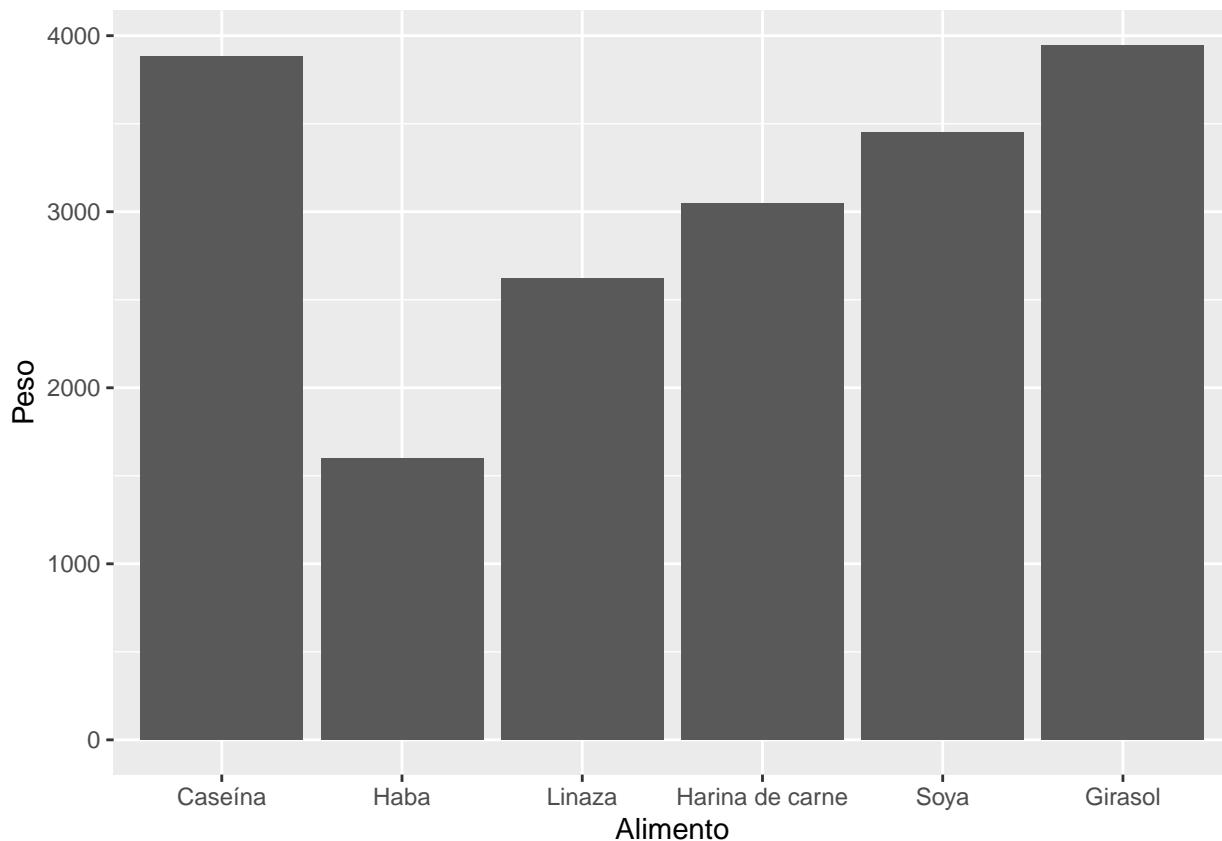
En el caso de los ejes discretos, es posible cambiar los nombres de cada una de las categorías. Por ejemplo, en la siguiente gráfica los nombres se encuentran en inglés:

```
ggplot(data = chickwts, aes(x = feed, y = weight)) +  
  geom_col()
```



Para cambiar los nombres a español se usa un vector con los nuevos nombres en el argumento `labels` dentro de `scale_x_discrete()`:

```
ggplot(data = chickwts, aes(x = feed, y = weight)) +
  geom_col() +
  labs(
    x = "Alimento",
    y = "Peso"
  ) +
  scale_x_discrete(labels = c("Caseína", "Haba", "Linaza",
                             "Harina de carne", "Soya",
                             "Girasol"))
```



5.4 Leyenda

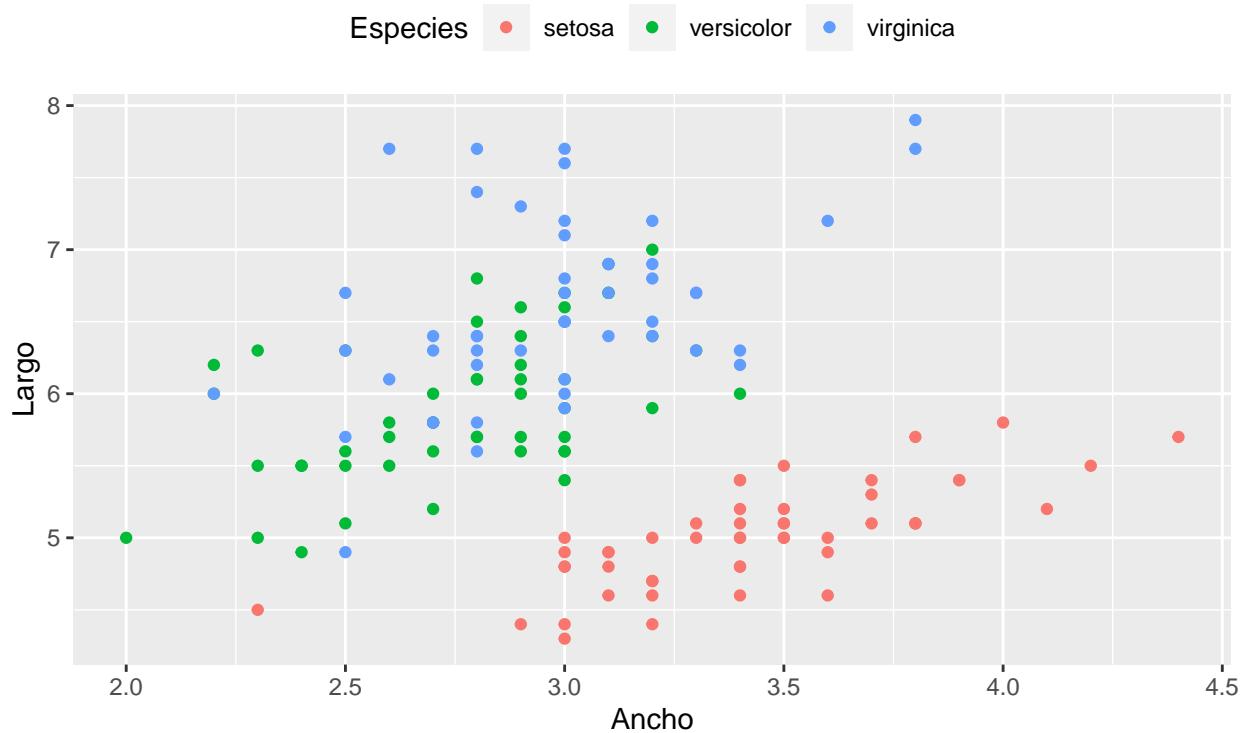
Previamente se mencionó que el título de la leyenda puede cambiarse en la capa `labs()`, utilizando el nombre de la estética que se empleó para generar la leyenda en `aes()`. Existen otras opciones relacionadas con la leyenda que deben especificarse dentro de `theme()`, una función que agrupa múltiples atributos relacionados con la leyenda y otros aspectos de la gráfica. A continuación se describirán algunos de estos atributos.

La posición de la leyenda se modifica con el argumento `legend.position`, el cual puede tomar los valores `left`, `right`, `bottom` y `top` para indicar la posición y `none` para eliminar completamente la leyenda:

```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length,
                                    color = Species)) +
  geom_point() +
  labs(title = "Iris",
       x = "Ancho",
       y = "Largo",
       subtitle = "Ancho VS Largo de sépalo",
       color = "Especies") +
  theme(legend.position = "top")
```

Iris

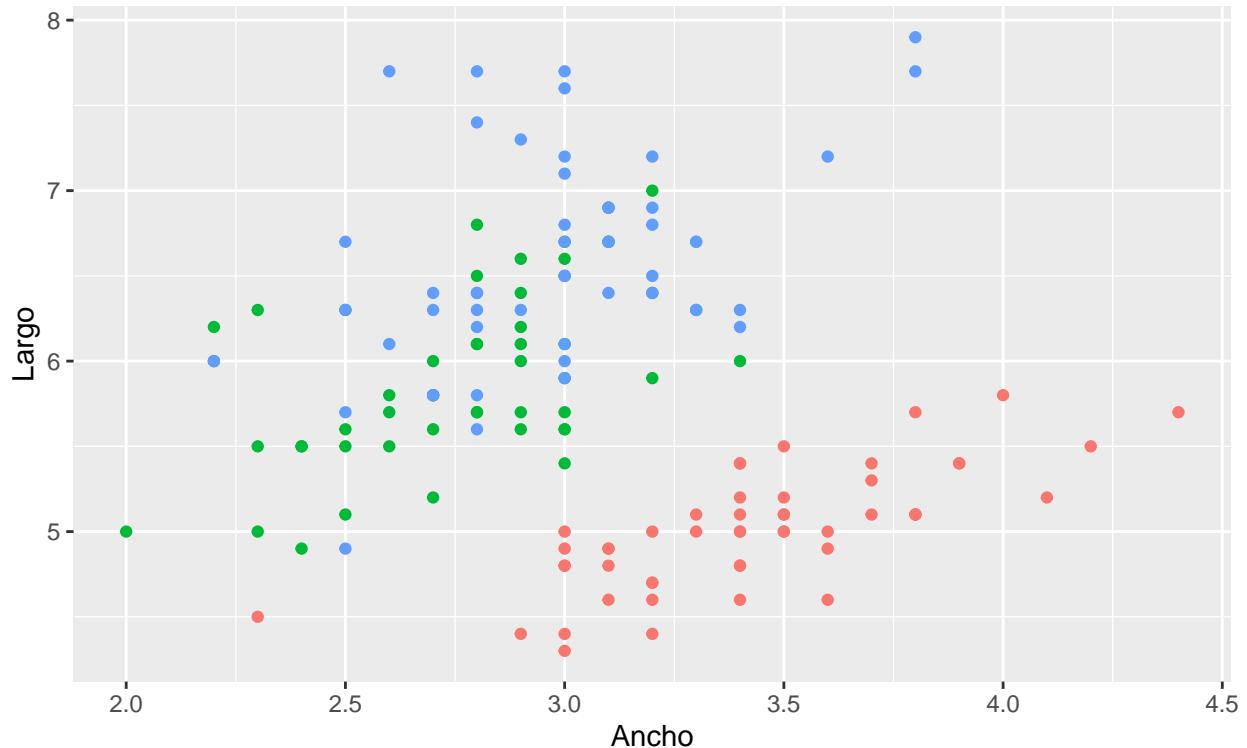
Ancho VS Largo de sépalo



```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length,
                                    color = Species)) +
  geom_point() +
  labs(title = "Iris",
       x = "Ancho",
       y = "Largo",
       subtitle = "Ancho VS Largo de sépalo",
       color = "Especies") +
  theme(legend.position = "none")
```

Iris

Ancho VS Largo de sépalo

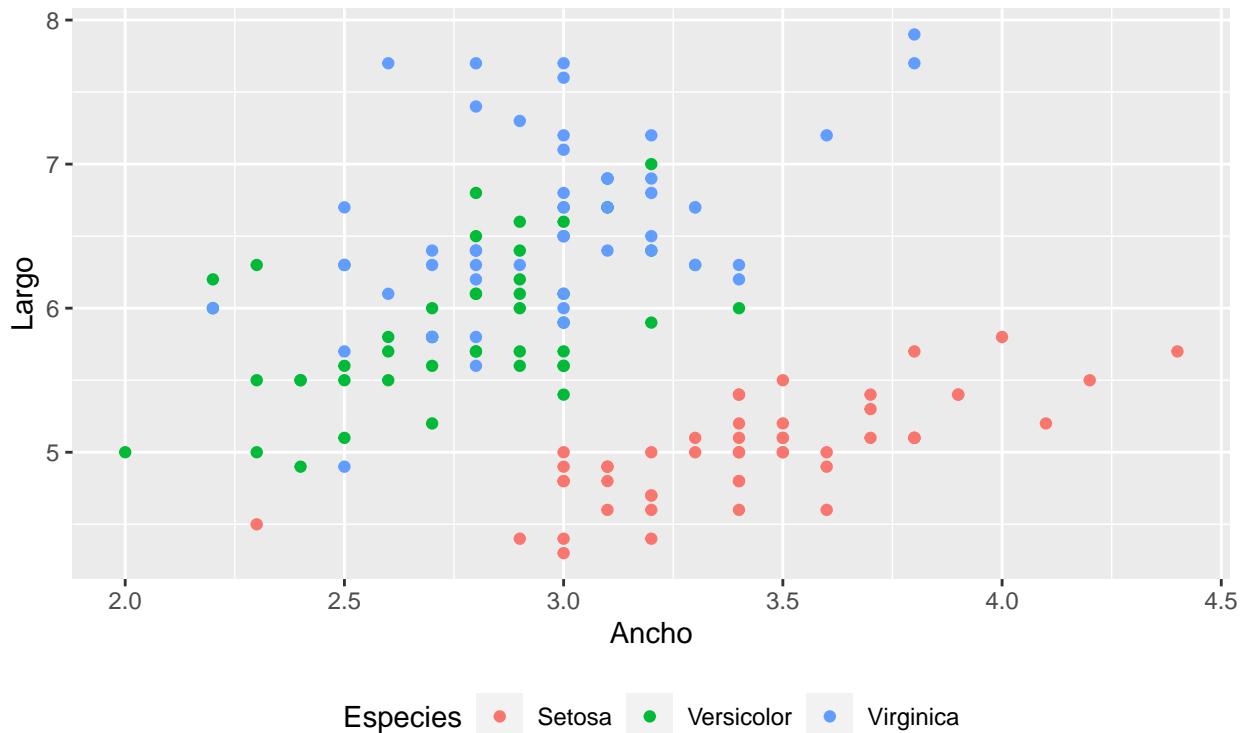


Para cambiar el texto de la leyenda es necesario modificar un atributo que no se encuentra `theme()`, el cual permite manualmente cambiar los valores asociados con la variable correspondiente, en este caso `color`, por lo tanto se debe usar la capa `scale_color_discrete()`. En el argumento `labels` se coloca un vector con los nuevos nombres:

```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length,
                                    col = Species)) +
  geom_point() +
  labs(title = "Iris",
       x = "Ancho",
       y = "Largo",
       subtitle = "Ancho VS Largo de sépalo",
       col = "Especies") +
  theme(legend.position = "bottom") +
  scale_color_discrete(labels = c("Setosa", "Versicolor", "Virginica"))
```

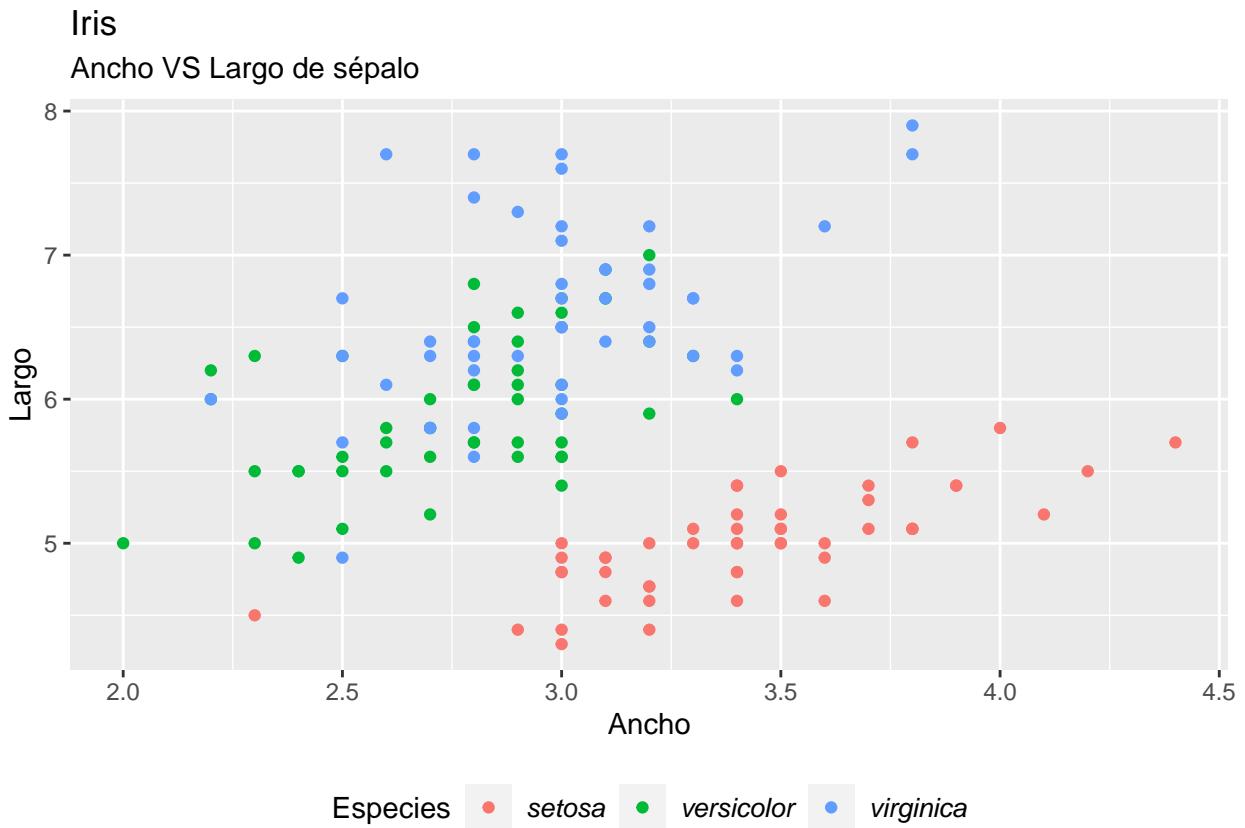
Iris

Ancho VS Largo de sépalo



Sin embargo esto no es completamente correcto, el epíteto específico debería de estar en cursivas y sin mayúscula. Mediante `scale_col_discrete()` solo se pueden cambiar los caracteres que se mostrarán en la leyenda, pero no se les puede dar formato en la misma función. Si se desean alterar el tamaño o tipografía se usa el argumento `legend.text` dentro de `theme()`. Las características del texto se especifican con la función `element_text()`. Para cambiar las letras a cursivas se debe usar el argumento `face` con el valor `"italic"`:

```
ggplot(data = iris, mapping = aes(x = Sepal.Width, y = Sepal.Length,
                                    color = Species)) +
  geom_point() +
  labs(title = "Iris",
       x = "Ancho",
       y = "Largo",
       subtitle = "Ancho VS Largo de sépalo",
       color = "Especies") +
  theme(legend.position = "bottom",
        legend.text = element_text(face = "italic", size = 10))
```



Esto cubre alguno de los usos que parecen ser los más comunes para mí. La capa `theme()` cuenta con muchos otros argumentos que permiten cambiar la leyenda, los ejes y los paneles de la gráfica, los cuales se encuentran explicados en `?theme()`.

5.5 Colores

Los colores en R se deben introducir como nombres entre comillas, debido a que son obtenidos del vector `colors()` que se imprime con la siguiente función:

`colors()`

```
## [1] "white"          "aliceblue"        "antiquewhite"
## [4] "antiquewhite1"  "antiquewhite2"    "antiquewhite3"
## [7] "antiquewhite4"  "aquamarine"       "aquamarine1"
## [10] "aquamarine2"     "aquamarine3"      "aquamarine4"
## [13] "azure"           "azure1"           "azure2"
## [16] "azure3"          "azure4"          "beige"
## [19] "bisque"          "bisque1"          "bisque2"
## [22] "bisque3"          "bisque4"          "black"
## [25] "blanchedalmond"  "blue"             "blue1"
## [28] "blue2"           "blue3"            "blue4"
## [31] "blueviolet"       "brown"            "brown1"
## [34] "brown2"          "brown3"          "brown4"
## [37] "burlywood"       "burlywood1"      "burlywood2"
## [40] "burlywood3"       "burlywood4"      "cadetblue"
## [43] "cadetblue1"       "cadetblue2"      "cadetblue3"
## [46] "cadetblue4"       "chartreuse"      "chartreuse1"
## [49] "chartreuse2"      "chartreuse3"      "chartreuse4"
## [52] "chocolate"       "chocolate1"      "chocolate2"
```

```

## [55] "chocolate3"
## [58] "coral1"
## [61] "coral4"
## [64] "cornsilk1"
## [67] "cornsilk4"
## [70] "cyan2"
## [73] "darkblue"
## [76] "darkgoldenrod1"
## [79] "darkgoldenrod4"
## [82] "darkgrey"
## [85] "darkolivegreen"
## [88] "darkolivegreen3"
## [91] "darkorange1"
## [94] "darkorange4"
## [97] "darkorchid2"
## [100] "darkred"
## [103] "darkseagreen1"
## [106] "darkseagreen4"
## [109] "darkslategray1"
## [112] "darkslategray4"
## [115] "darkviolet"
## [118] "deeppink2"
## [121] "deepskyblue"
## [124] "deepskyblue3"
## [127] "dimgrey"
## [130] "dodgerblue2"
## [133] "firebrick"
## [136] "firebrick3"
## [139] "forestgreen"
## [142] "gold"
## [145] "gold3"
## [148] "goldenrod1"
## [151] "goldenrod4"
## [154] "gray1"
## [157] "gray4"
## [160] "gray7"
## [163] "gray10"
## [166] "gray13"
## [169] "gray16"
## [172] "gray19"
## [175] "gray22"
## [178] "gray25"
## [181] "gray28"
## [184] "gray31"
## [187] "gray34"
## [190] "gray37"
## [193] "gray40"
## [196] "gray43"
## [199] "gray46"
## [202] "gray49"
## [205] "gray52"
## [208] "gray55"
## [211] "gray58"
## [214] "gray61"
"chocolate4"
"coral2"
"cornflowerblue"
"cornsilk2"
"cyan"
"cyan3"
"darkcyan"
"darkgoldenrod2"
"darkgray"
"darkkhaki"
"darkolivegreen1"
"darkolivegreen4"
"darkorange2"
"darkorchid"
"darkorchid3"
"darkred"
"darksalmon"
"darkseagreen2"
"darkslateblue"
"darkslategray1"
"darkslategray2"
"darkslategray4"
"darkviolet"
"deeppink"
"deeppink3"
"deepskyblue1"
"deepskyblue4"
"dimgrey"
"dodgerblue"
"dodgerblue3"
"firebrick1"
"firebrick4"
"gainsboro"
"gold1"
"gold4"
"goldenrod2"
"gray"
"gray2"
"gray5"
"gray7"
"gray10"
"gray13"
"gray16"
"gray19"
"gray22"
"gray25"
"gray28"
"gray31"
"gray34"
"gray37"
"gray40"
"gray43"
"gray46"
"gray49"
"gray52"
"gray55"
"gray58"
"gray61"
"coral"
"coral3"
"cornsilk"
"cornsilk3"
"cyan1"
"cyan4"
"darkgoldenrod"
"darkgoldenrod3"
"darkgreen"
"darkmagenta"
"darkolivegreen2"
"darkorange"
"darkorange3"
"darkorchid1"
"darkorchid4"
"darkseagreen"
"darkseagreen3"
"darkslategray"
"darkslategray3"
"darkturquoise"
"deeppink1"
"deeppink4"
"deepskyblue2"
"dimgray"
"dodgerblue1"
"dodgerblue4"
"firebrick2"
"floralwhite"
"ghostwhite"
"gold2"
"goldenrod"
"goldenrod3"
"gray0"
"gray3"
"gray6"
"gray9"
"gray12"
"gray15"
"gray18"
"gray21"
"gray24"
"gray27"
"gray30"
"gray33"
"gray36"
"gray39"
"gray42"
"gray45"
"gray48"
"gray51"
"gray54"
"gray57"
"gray60"
"gray63"

```

```

## [217] "gray64"          "gray65"          "gray66"
## [220] "gray67"          "gray68"          "gray69"
## [223] "gray70"          "gray71"          "gray72"
## [226] "gray73"          "gray74"          "gray75"
## [229] "gray76"          "gray77"          "gray78"
## [232] "gray79"          "gray80"          "gray81"
## [235] "gray82"          "gray83"          "gray84"
## [238] "gray85"          "gray86"          "gray87"
## [241] "gray88"          "gray89"          "gray90"
## [244] "gray91"          "gray92"          "gray93"
## [247] "gray94"          "gray95"          "gray96"
## [250] "gray97"          "gray98"          "gray99"
## [253] "gray100"         "green"           "green1"
## [256] "green2"          "green3"          "green4"
## [259] "greenyellow"      "grey"            "grey0"
## [262] "grey1"            "grey2"           "grey3"
## [265] "grey4"            "grey5"           "grey6"
## [268] "grey7"            "grey8"           "grey9"
## [271] "grey10"           "grey11"          "grey12"
## [274] "grey13"           "grey14"          "grey15"
## [277] "grey16"           "grey17"          "grey18"
## [280] "grey19"           "grey20"          "grey21"
## [283] "grey22"           "grey23"          "grey24"
## [286] "grey25"           "grey26"          "grey27"
## [289] "grey28"           "grey29"          "grey30"
## [292] "grey31"           "grey32"          "grey33"
## [295] "grey34"           "grey35"          "grey36"
## [298] "grey37"           "grey38"          "grey39"
## [301] "grey40"           "grey41"          "grey42"
## [304] "grey43"           "grey44"          "grey45"
## [307] "grey46"           "grey47"          "grey48"
## [310] "grey49"           "grey50"          "grey51"
## [313] "grey52"           "grey53"          "grey54"
## [316] "grey55"           "grey56"          "grey57"
## [319] "grey58"           "grey59"          "grey60"
## [322] "grey61"           "grey62"          "grey63"
## [325] "grey64"           "grey65"          "grey66"
## [328] "grey67"           "grey68"          "grey69"
## [331] "grey70"           "grey71"          "grey72"
## [334] "grey73"           "grey74"          "grey75"
## [337] "grey76"           "grey77"          "grey78"
## [340] "grey79"           "grey80"          "grey81"
## [343] "grey82"           "grey83"          "grey84"
## [346] "grey85"           "grey86"          "grey87"
## [349] "grey88"           "grey89"          "grey90"
## [352] "grey91"           "grey92"          "grey93"
## [355] "grey94"           "grey95"          "grey96"
## [358] "grey97"           "grey98"          "grey99"
## [361] "grey100"          "honeydew"        "honeydew1"
## [364] "honeydew2"         "honeydew3"        "honeydew4"
## [367] "hotpink"          "hotpink1"        "hotpink2"
## [370] "hotpink3"         "hotpink4"        "indianred"
## [373] "indianred1"       "indianred2"      "indianred3"
## [376] "indianred4"       "ivory"           "ivory1"

```

```

## [379] "ivory2"           "ivory3"           "ivory4"
## [382] "khaki"             "khaki1"            "khaki2"
## [385] "khaki3"            "khaki4"            "lavender"
## [388] "lavenderblush"      "lavenderblush1"     "lavenderblush2"
## [391] "lavenderblush3"      "lavenderblush4"     "lawngreen"
## [394] "lemonchiffon"       "lemonchiffon1"     "lemonchiffon2"
## [397] "lemonchiffon3"       "lemonchiffon4"     "lightblue"
## [400] "lightblue1"          "lightblue2"          "lightblue3"
## [403] "lightblue4"          "lightcoral"         "lightcyan"
## [406] "lightcyan1"          "lightcyan2"         "lightcyan3"
## [409] "lightcyan4"          "lightgoldenrod"    "lightgoldenrod1"
## [412] "lightgoldenrod2"      "lightgoldenrod3"    "lightgoldenrod4"
## [415] "lightgoldenrodyellow" "lightgray"          "lightgreen"
## [418] "lightgrey"          "lightpink"          "lightpink1"
## [421] "lightpink2"          "lightpink3"          "lightpink4"
## [424] "lightsalmon"         "lightsalmon1"       "lightsalmon2"
## [427] "lightsalmon3"        "lightsalmon4"       "lightseagreen"
## [430] "lightskyblue"        "lightskyblue1"      "lightskyblue2"
## [433] "lightskyblue3"        "lightskyblue4"      "lightslateblue"
## [436] "lightslategray"       "lightslategrey"     "lightsteelblue"
## [439] "lightsteelblue1"      "lightsteelblue2"    "lightsteelblue3"
## [442] "lightsteelblue4"      "lightyellow"        "lightyellow1"
## [445] "lightyellow2"        "lightyellow3"       "lightyellow4"
## [448] "limegreen"          "linen"              "magenta"
## [451] "magenta1"            "magenta2"           "magenta3"
## [454] "magenta4"            "maroon"             "maroon1"
## [457] "maroon2"              "maroon3"            "maroon4"
## [460] "mediumaquamarine"    "mediumblue"         "mediumorchid"
## [463] "mediumorchid1"        "mediumorchid2"      "mediumorchid3"
## [466] "mediumorchid4"        "mediumpurple"       "mediumpurple1"
## [469] "mediumpurple2"        "mediumpurple3"      "mediumpurple4"
## [472] "mediumseagreen"       "mediumslateblue"    "mediumspringgreen"
## [475] "mediumturquoise"      "mediumvioletred"   "midnightblue"
## [478] "mintcream"            "mistyrose"          "mistyrose1"
## [481] "mistyrose2"            "mistyrose3"          "mistyrose4"
## [484] "moccasin"             "navajowhite"        "navajowhite1"
## [487] "navajowhite2"          "navajowhite3"       "navajowhite4"
## [490] "navy"                 "navyblue"           "oldlace"
## [493] "olivedrab"            "olivedrab1"         "olivedrab2"
## [496] "olivedrab3"            "olivedrab4"         "orange"
## [499] "orange1"              "orange2"             "orange3"
## [502] "orange4"              "orangered"          "orangered1"
## [505] "orangered2"            "orangered3"         "orangered4"
## [508] "orchid"                "orchid1"             "orchid2"
## [511] "orchid3"              "orchid4"             "palegoldenrod"
## [514] "palegreen"             "palegreen1"          "palegreen2"
## [517] "palegreen3"            "palegreen4"          "paleturquoise"
## [520] "paleturquoise1"        "paleturquoise2"     "paleturquoise3"
## [523] "paleturquoise4"        "palevioletred"      "palevioletred1"
## [526] "palevioletred2"        "palevioletred3"     "palevioletred4"
## [529] "papayawhip"            "peachpuff"          "peachpuff1"
## [532] "peachpuff2"            "peachpuff3"          "peachpuff4"
## [535] "peru"                  "pink"               "pink1"
## [538] "pink2"                  "pink3"               "pink4"

```

```

## [541] "plum"                      "plum1"                      "plum2"
## [544] "plum3"                      "plum4"                       "powderblue"
## [547] "purple"                     "purple1"                     "purple2"
## [550] "purple3"                    "purple4"                     "red"
## [553] "red1"                       "red2"                        "red3"
## [556] "red4"                       "rosybrown"                   "rosybrown1"
## [559] "rosybrown2"                 "rosybrown3"                  "rosybrown4"
## [562] "royalblue"                  "royalblue1"                  "royalblue2"
## [565] "royalblue3"                 "royalblue4"                  "saddlebrown"
## [568] "salmon"                     "salmon1"                     "salmon2"
## [571] "salmon3"                    "salmon4"                     "sandybrown"
## [574] "seagreen"                   "seagreen1"                   "seagreen2"
## [577] "seagreen3"                 "seagreen4"                   "seashell"
## [580] "seashell1"                 "seashell2"                   "seashell3"
## [583] "seashell4"                 "sienna"                      "sienna1"
## [586] "sienna2"                    "sienna3"                     "sienna4"
## [589] "skyblue"                    "skyblue1"                    "skyblue2"
## [592] "skyblue3"                   "skyblue4"                     "slateblue"
## [595] "slateblue1"                 "slateblue2"                  "slateblue3"
## [598] "slateblue4"                 "slategray"                   "slategray1"
## [601] "slategray2"                 "slategray3"                  "slategray4"
## [604] "slategrey"                  "snow"                         "snow1"
## [607] "snow2"                      "snow3"                        "snow4"
## [610] "springgreen"                "springgreen1"                "springgreen2"
## [613] "springgreen3"               "springgreen4"                "steelblue"
## [616] "steelblue1"                 "steelblue2"                  "steelblue3"
## [619] "steelblue4"                 "tan"                          "tan1"
## [622] "tan2"                       "tan3"                        "tan4"
## [625] "thistle"                   "thistle1"                    "thistle2"
## [628] "thistle3"                  "thistle4"                    "tomato"
## [631] "tomato1"                   "tomato2"                     "tomato3"
## [634] "tomato4"                   "turquoise"                   "turquoise1"
## [637] "turquoise2"                 "turquoise3"                  "turquoise4"
## [640] "violet"                     "violetred"                   "violetred1"
## [643] "violetred2"                "violetred3"                  "violetred4"
## [646] "wheat"                      "wheat1"                      "wheat2"
## [649] "wheat3"                     "wheat4"                      "whitesmoke"
## [652] "yellow"                     "yellow1"                     "yellow2"
## [655] "yellow3"                    "yellow4"                     "yellowgreen"

```

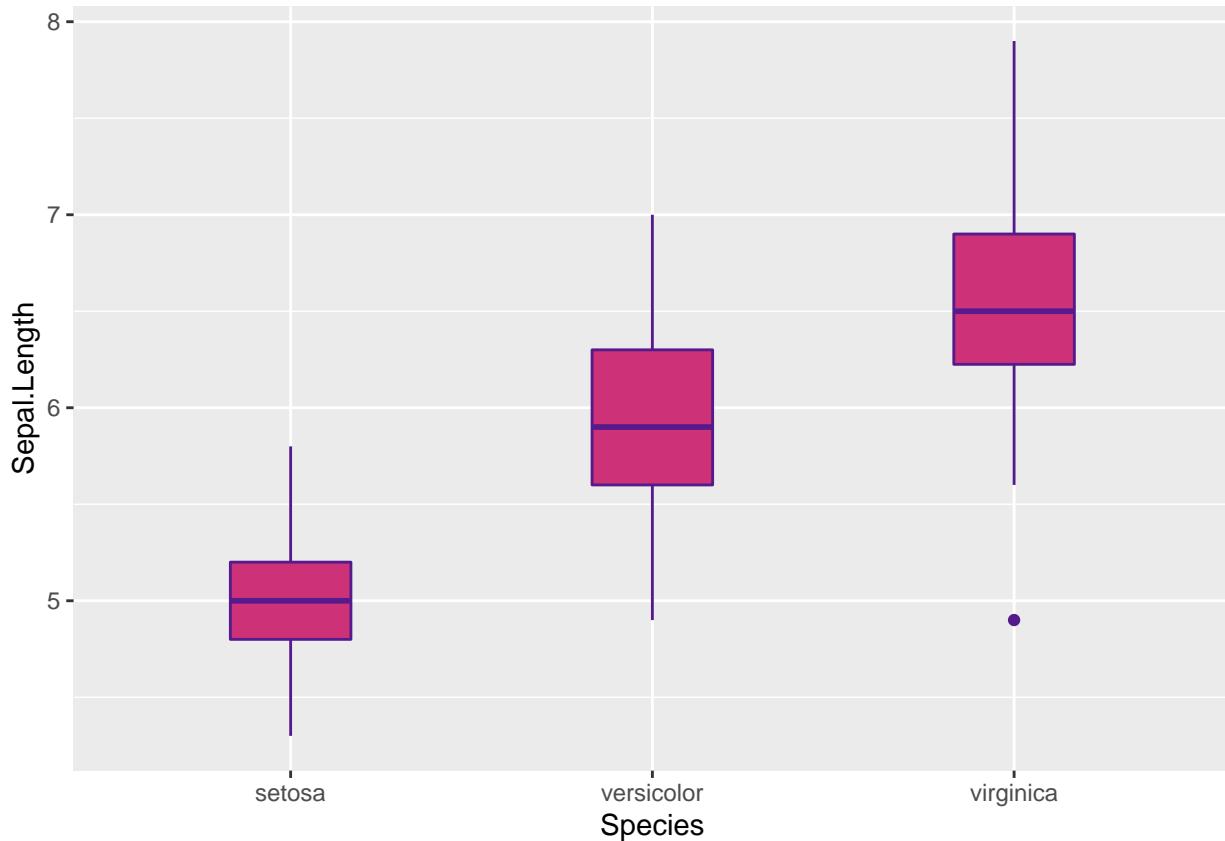
Los colores que corresponden a cada nombre pueden consultarse en este enlace o en el archivo Rcolor.pdf proporcionado junto con el manual.

Cuando se desee usar alguno de estos colores se deben escribir entre comillas en aquellos argumentos que acepten un color o un vector de colores:

```

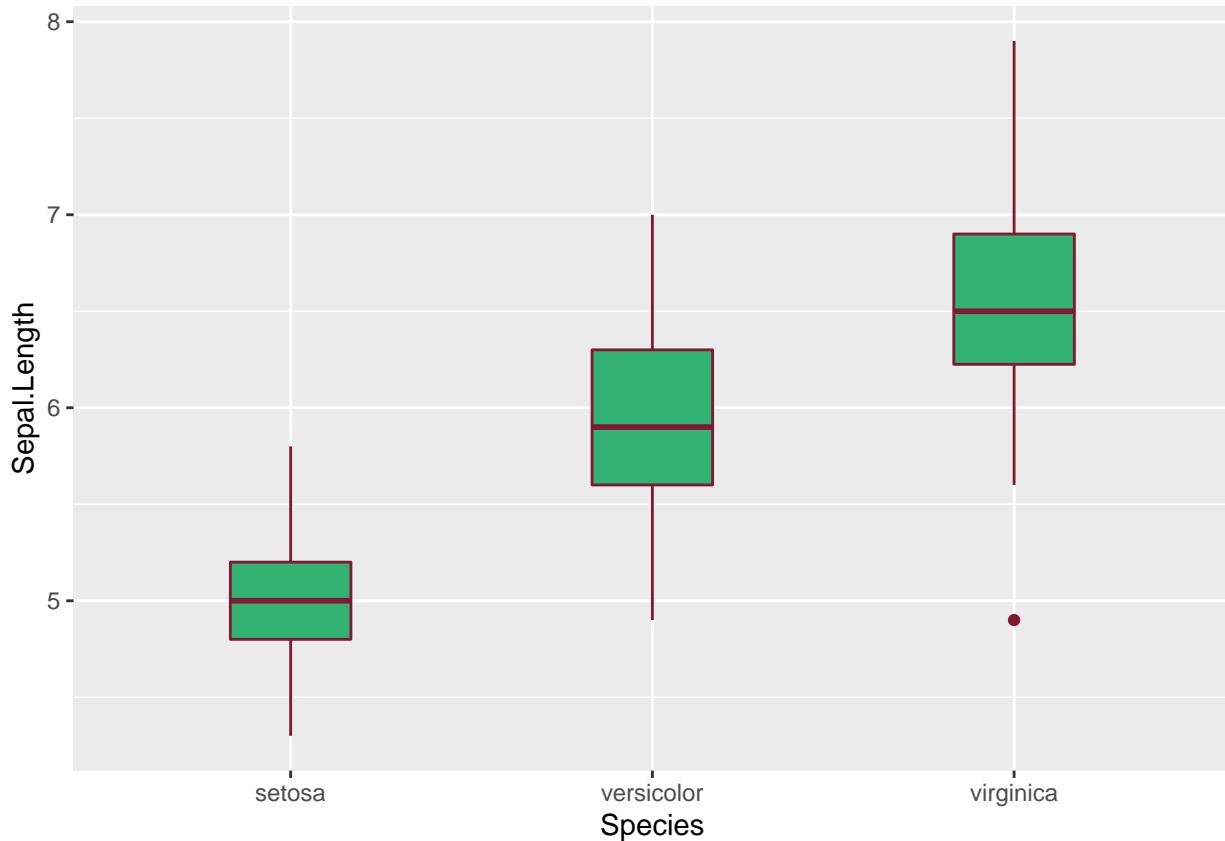
ggplot(data = iris, aes(x = Species, y = Sepal.Length)) +
  geom_boxplot(fill = "violetred3",
               color = "purple4",
               width = 1/3)

```



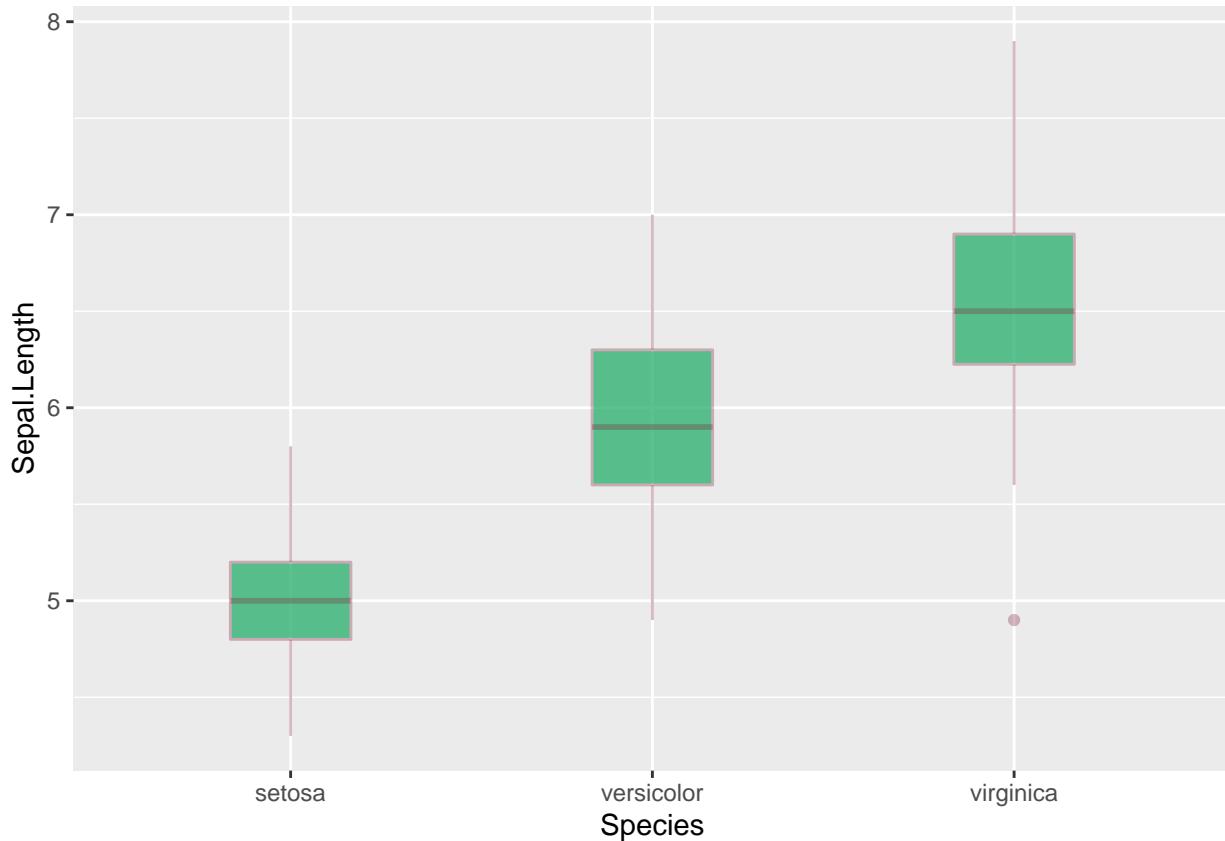
Los colores también se pueden especificar con mayor precisión usando funciones como `rgb()` o `hsv()`. Con `rgb()` se especifica la intensidad de rojo, verde y azul con valores de 0 a 1:

```
ggplot(data = iris, aes(x = Species, y = Sepal.Length)) +
  geom_boxplot(fill = rgb(red = .2, blue = .45, green = .7),
               col = rgb(red = .5, blue = .2, green = .1),
               width = 1/3)
```



La opacidad de cada color también puede ser indicada con el argumento `alpha`, siendo 1 la máxima opacidad y 0 la máxima transparencia:

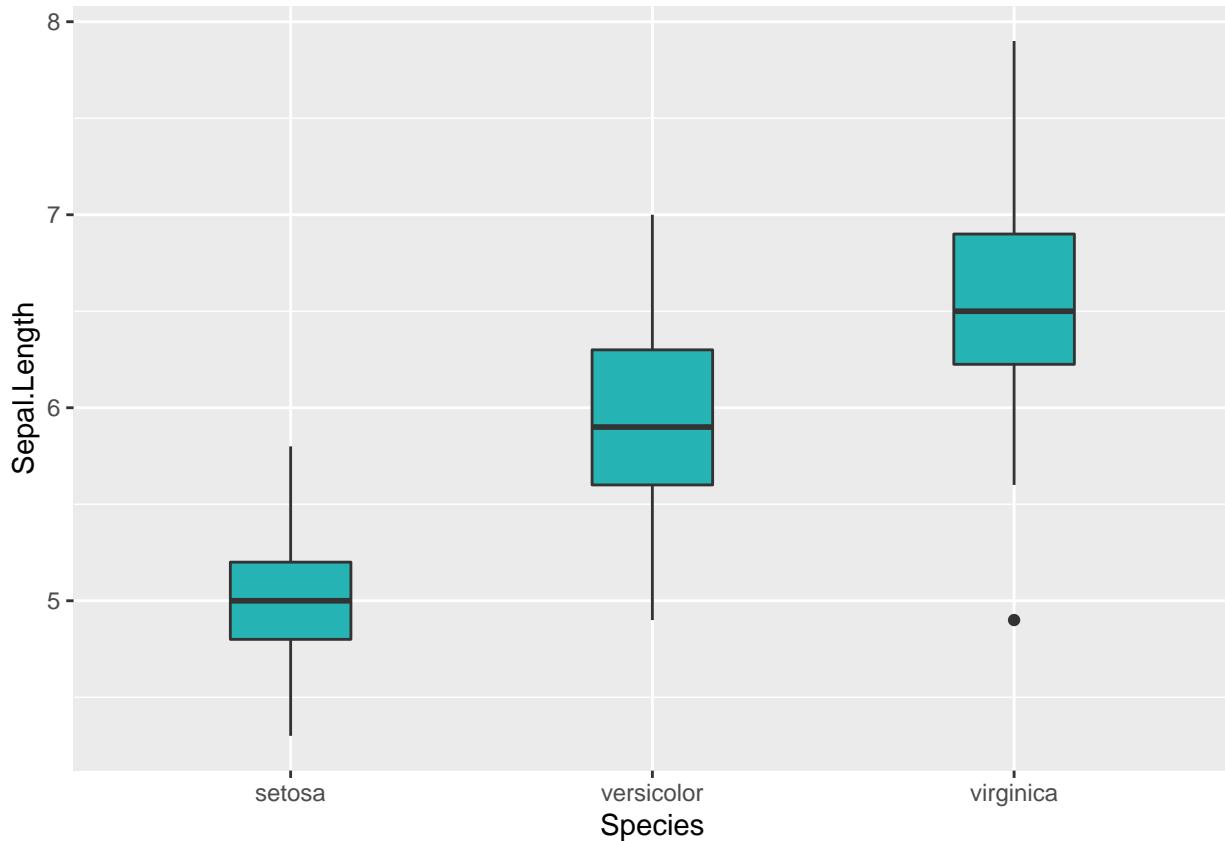
```
ggplot(data = iris, aes(x = Species, y = Sepal.Length)) +
  geom_boxplot(fill = rgb(red = .2, blue = .45, green = .7, alpha = .8),
               col = rgb(red = .5, blue = .2, green = .1, alpha = .3),
               width = 1/3)
```



Cuando se utiliza `rgb()` fuera de cualquier otra función, se obtiene el color representado con un número hexadecimal, el cual también puede ser usado en los argumentos que permiten introducir color:

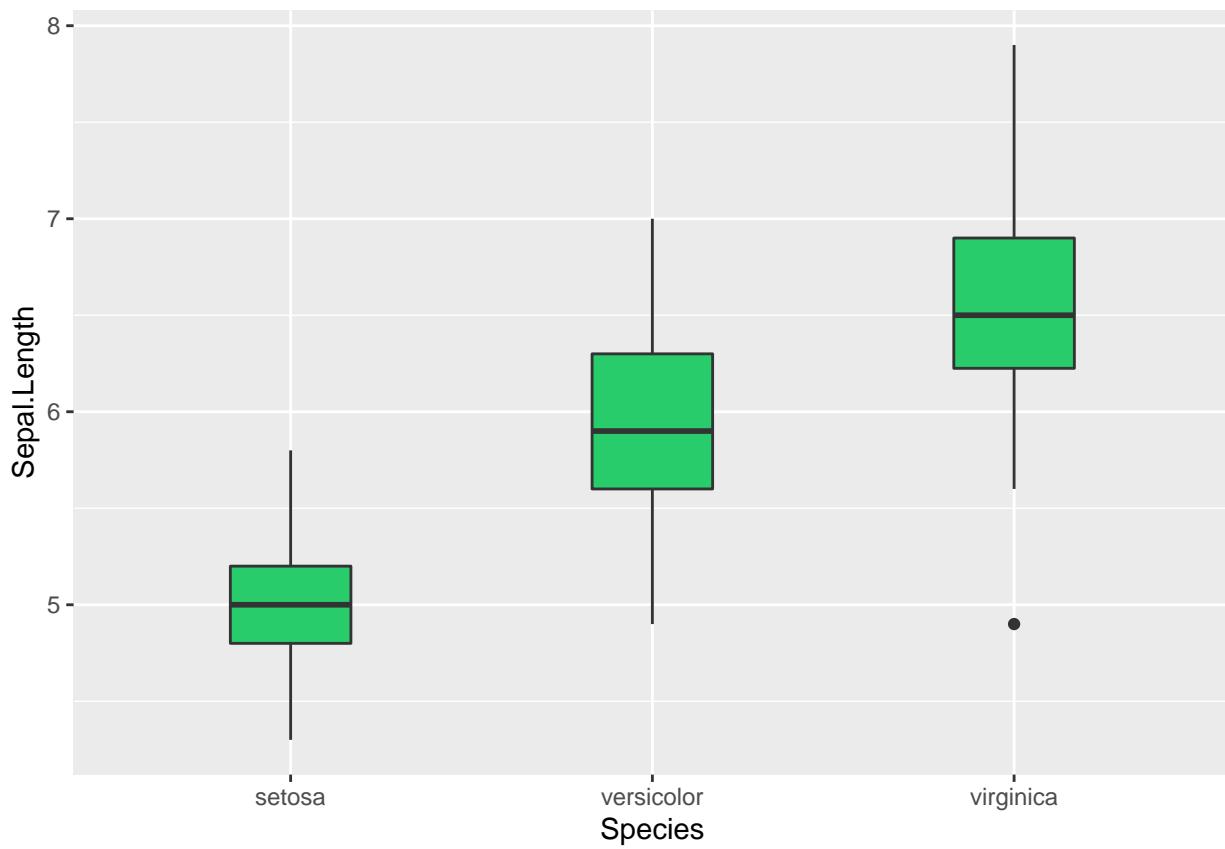
```
rgb(red = .15, green = .7, blue = .7)
## [1] "#26B3B3"

ggplot(data = iris, aes(x = Species, y = Sepal.Length)) +
  geom_boxplot(fill = "#26B3B3",
               width = 1/3)
```



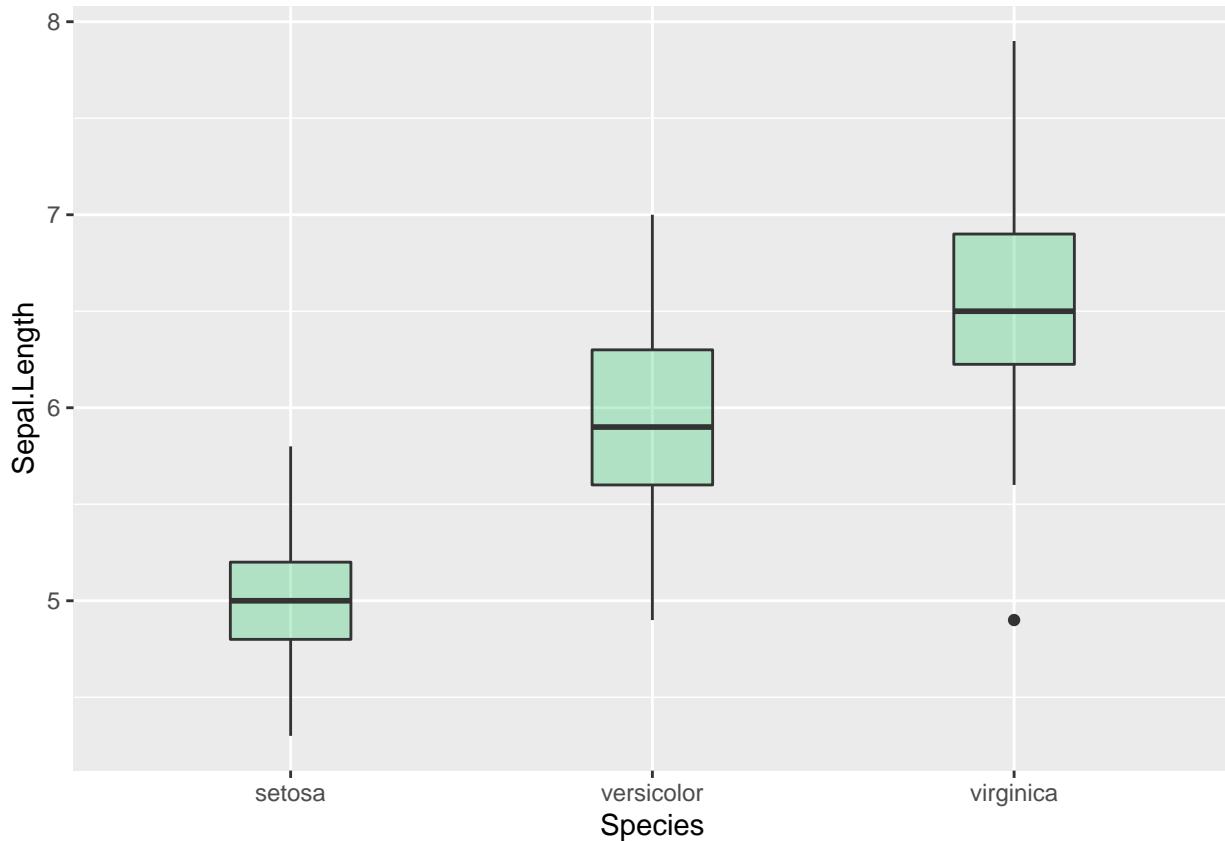
Los colores creados con `hsv()` son resultado de especificar el matiz (hue), saturación (saturation) y valor (value). Los argumentos `h`, `s` y `v` toman valores de 0 a 1:

```
ggplot(data = iris, aes(x = Species, y = Sepal.Length)) +
  geom_boxplot(fill = hsv(h = .4, s = .8, v = .8),
               width = 1/3)
```



De igual manera se puede obtener un valor hexadecimal y especificar la opacidad con `alpha`:

```
ggplot(data = iris, aes(x = Species, y = Sepal.Length)) +  
  geom_boxplot(fill = hsv(h = .4, s = .8, v = .8, alpha = .3),  
               width = 1/3)
```



5.5.1 Paletas

Para realizar gráficas visualmente atractivas se pueden crear paletas con las funciones previamente mencionadas o utilizar paletas incluidas en otros paquetes a través de la librería `paletteer`, la cual agrupa múltiples paletas que se pueden ser consultadas aquí.

La manera en que se usan con `ggplot()` es mediante funciones que indican si la paleta se va a usar para el relleno, (fill) o los contornos (color). También se debe tomar en cuenta que algunas paletas solo funcionan para variables continuas y otras para variables discretas. De esta manera, las capas disponibles son:

- `scale_fill_paletteer_c()`: relleno para variables continuas.
- `scale_fill_paletteer_d()`: relleno para variables discretas.
- `scale_color_paletteer_c()`: contorno para variables continuas.
- `scale_color_paletteer_d()`: contorno para variables discretas.

Cada una de estas capas tiene como primer argumento `palette`, en el cual se deben poner el paquete y el nombre de la paleta entre comillas de la siguiente manera: "`paquete::paleta`". Las paletas disponibles y sus paquetes correspondientes pueden consultarse con: `View(palettes_c_names)` y `View(palettes_d_names)`. Para usar las paletas en la lista deben instalarse los paquetes correspondientes, si es que no están instalados al momento de usarlas en una gráfica, la consola mostrará un mensaje que permite instalar el paquete al escribir 1 y presionando Enter.

Cabe destacar que estas funciones solo pueden ser utilizadas cuando, el relleno o el color de contorno están asociados a una variable que ha sido especificada dentro de `aes()`.

El data frame `ToothGrowth` contiene la longitud de odontoblastos de 60 conejillos de indias de acuerdo con tres dosis de vitamina C administrada por dos medios: jugo de naranja o ácido ascórbico. De este modo, el data frame tiene tres variables: longitud, tipo de suplemento y dosis.

```
str(ToothGrowth)
```

```

## 'data.frame':   60 obs. of  3 variables:
## $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
## $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 ...
## $ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...

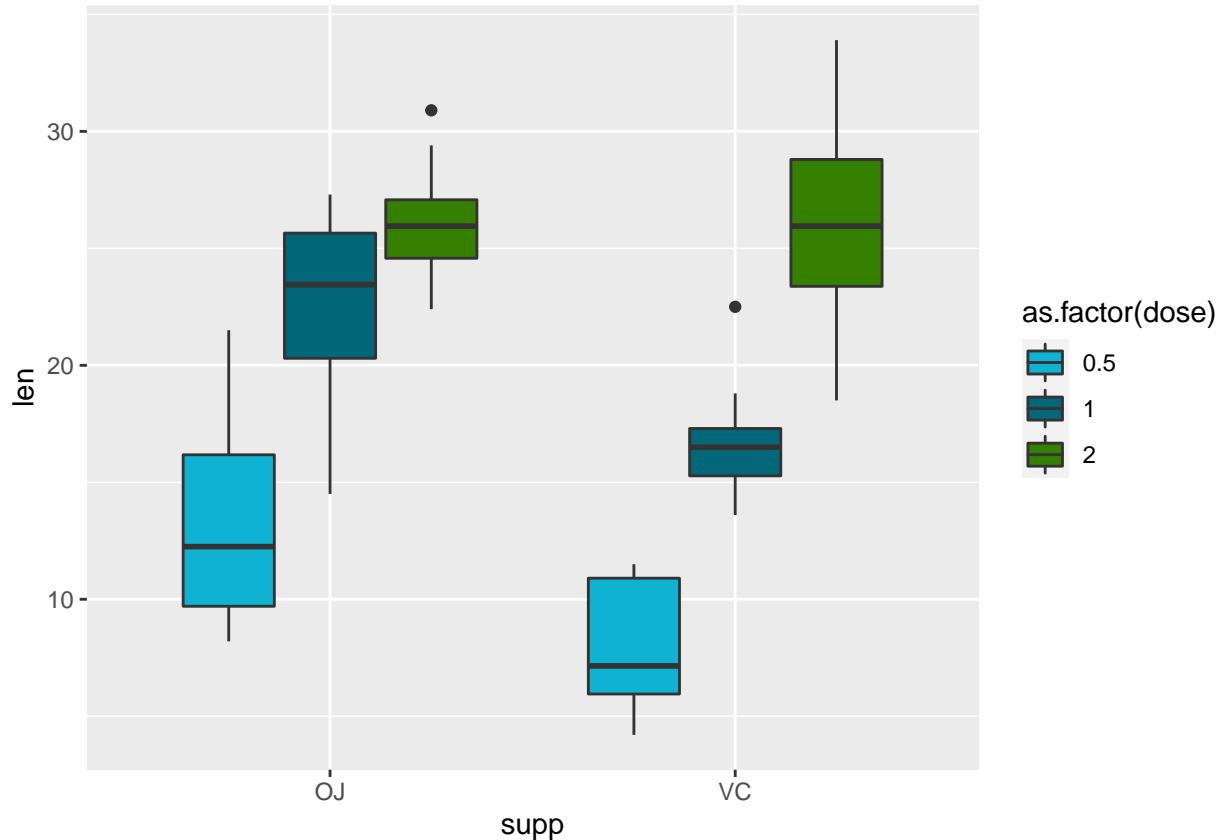
```

Las tres variables pueden representarse en una gráfica de cajas y bigotes de la siguiente manera:

```

ggplot(data = ToothGrowth, aes(x = supp, y = len, fill = as.factor(dose))) +
  geom_boxplot() +
  scale_fill_paleteer_d(palette = "calecopal::kelp2")

```

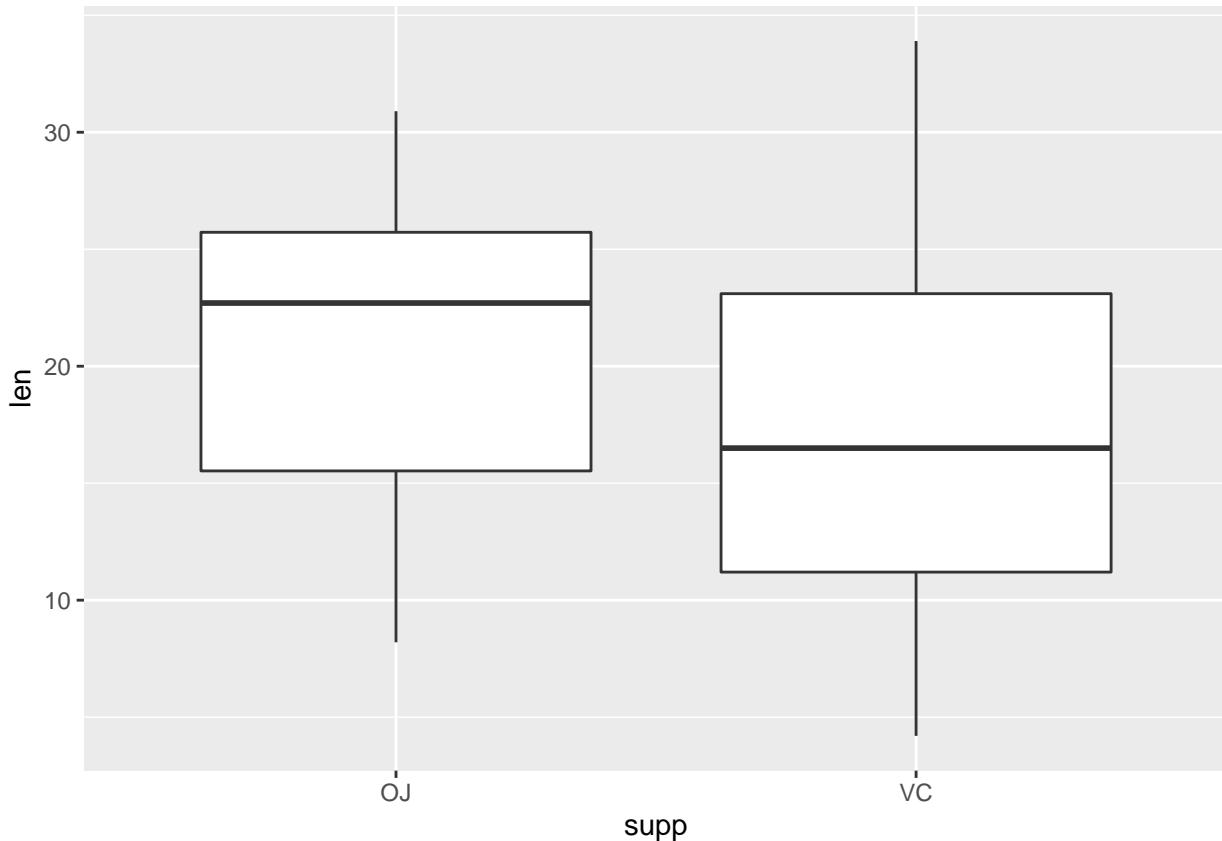


En el caso anterior, `dose` debe ser indicada como un factor debido a que en el data frame original es interpretado como una variable continua. De no hacerlo la gráfica no tiene sentido:

```

ggplot(data = ToothGrowth, aes(x = supp, y = len, fill = dose)) +
  geom_boxplot() +
  scale_fill_paleteer_d(palette = "calecopal::kelp2")

```

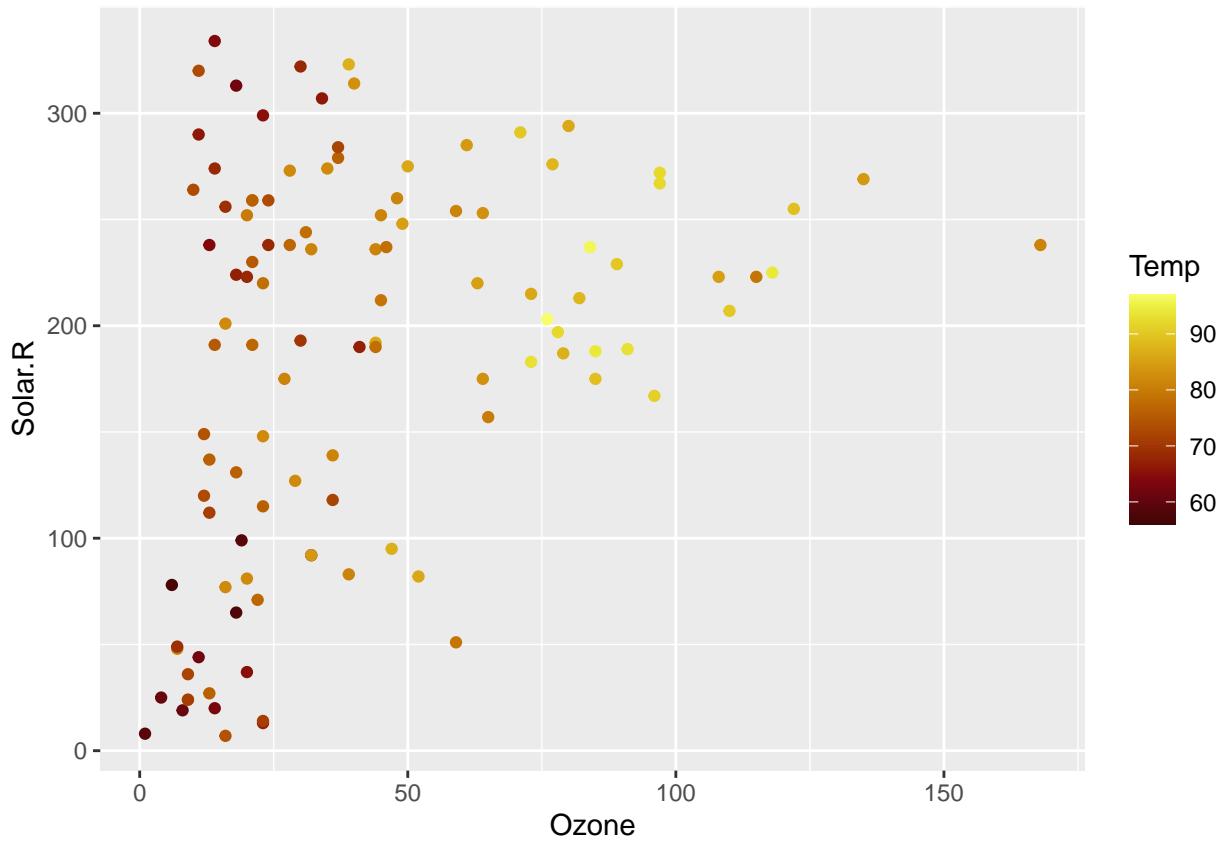


Un ejemplo en el que el color puede utilizarse para una variable continua es el data frame `airquality`, que contiene la calidad del aire de Nueva York, de mayo a septiembre de 1973. Las variables son promedio de ozono en partes por mil millones, radiación solar, promedio de velocidad de viento, temperatura máxima diaria, mes y día:

```
str(airquality)
## 'data.frame': 153 obs. of 6 variables:
## $ Ozone : int 41 36 12 18 NA 28 23 19 8 NA ...
## $ Solar.R: int 190 118 149 313 NA NA 299 99 19 194 ...
## $ Wind   : num 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
## $ Temp   : int 67 72 74 62 56 66 65 59 61 69 ...
## $ Month  : int 5 5 5 5 5 5 5 5 5 5 ...
## $ Day    : int 1 2 3 4 5 6 7 8 9 10 ...
```

para representar las tres variables continuas en la misma gráfica de dispersión:

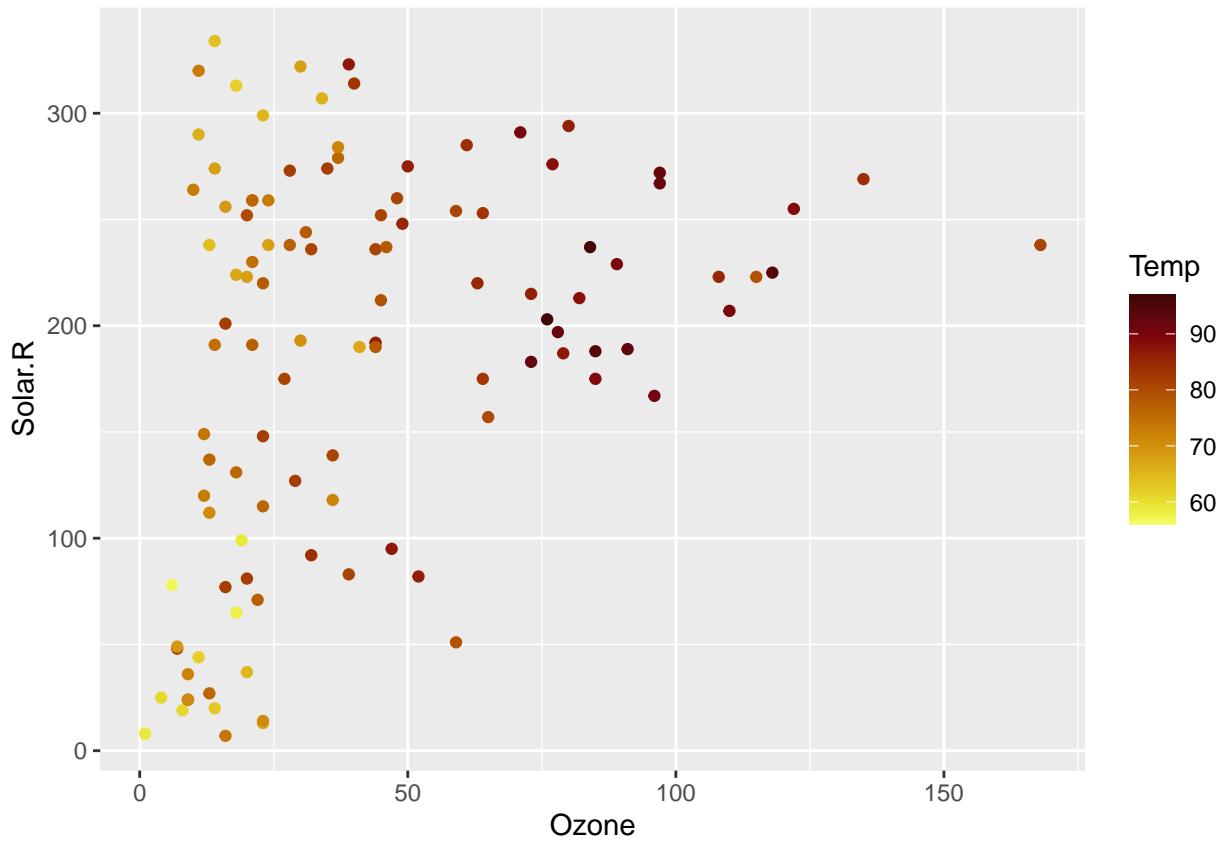
```
ggplot(data = airquality, aes(x = Ozone, y = Solar.R, color = Temp)) +
  geom_point() +
  scale_color_cooler(palette = "pals::ocean.oxy")
## Warning: Removed 42 rows containing missing values (geom_point).
```



Para este ejemplo podría ser conveniente que la escala estuviera invertida, de modo que a mayor temperatura el color sea más oscuro. Ello puede hacerse mediante el argumento `direction`, que por defecto es 1, debe cambiarse a -1:

```
ggplot(data = airquality, aes(x = Ozone, y = Solar.R, color = Temp)) +
  geom_point() +
  scale_color_palatteer_c(palette = "pals::ocean.oxy", direction = -1)

## Warning: Removed 42 rows containing missing values (geom_point).
```



Usando `paletteer_c()` y `paletteer_d()` pueden consultarse los colores que conforman cada paleta, los cuales se muestran en un vector con números hexadecimales. Para las paletas continuas se debe especificar la cantidad de colores que se desean.

```
paletteer_c(palette = "ggthemes::Classic_Orange", n = 15)
## <colors>
## #F0C294FF #F6B881FF #FBAE6DFF #FDA15AFF #FD9346FF #FB8433FF #F67524FF #F06511FF #E5570AFF #DB4903FF #
```

De lo contrario se obtendrá un error:

```
paletteer_c(palette = "grDevices::Mint")
## Error in gen_fun(name = palette[2], n = n): argument "n" is missing, with no default
paletteer_c(palette = "grDevices::Mint", n = 20)
## <colors>
## #005D67FF #00646CFF #006C71FF #007476FF #007C7BFF #218380FF #338B86FF #42938BFF #509B91FF #5DA397FF #
```

`paletteer_d(palette = "awtools::a_palette")`

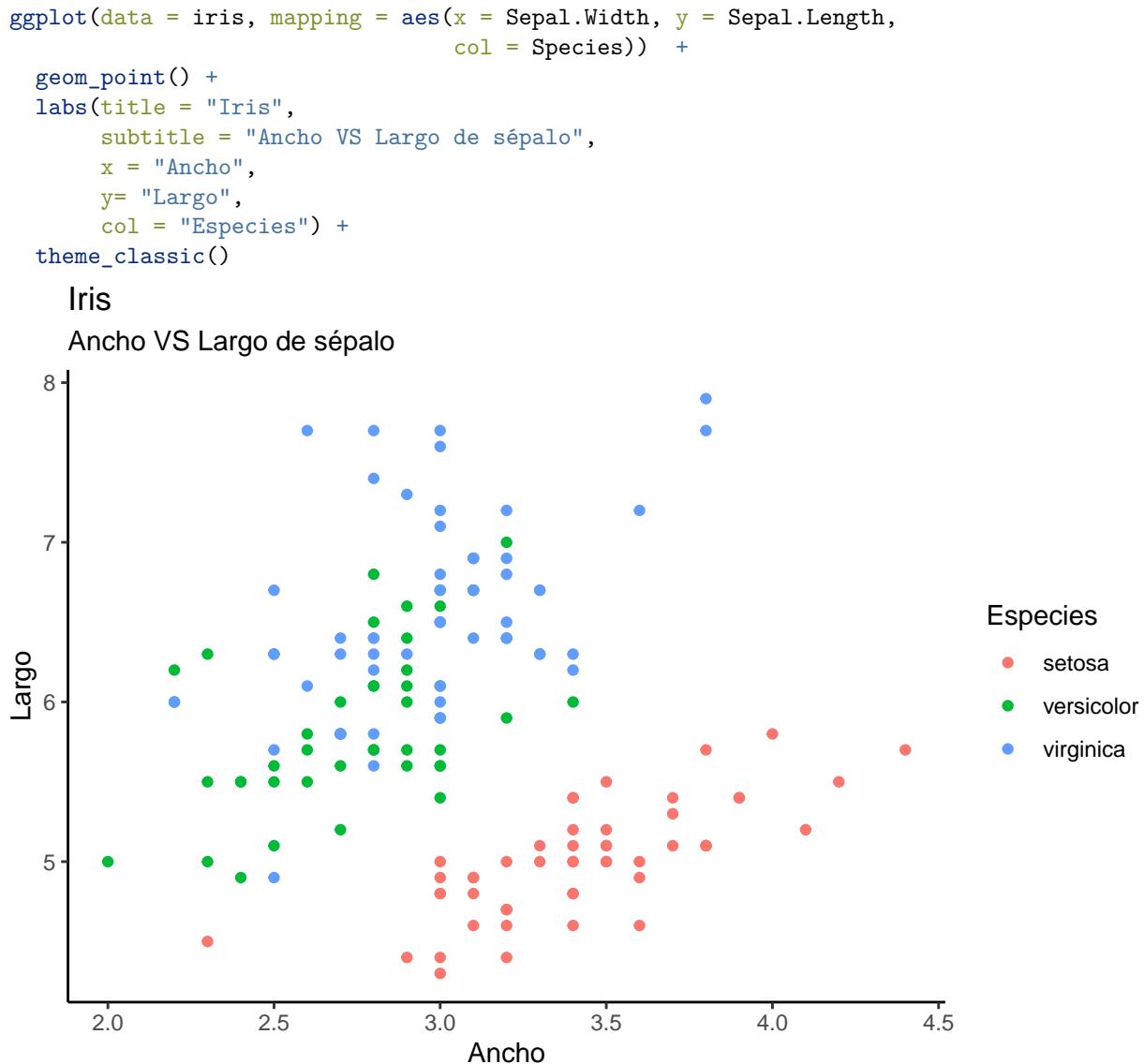
```
## <colors>
## #2A363BFF #019875FF #99B898FF #FECEA8FF #FF847CFF #E84A5FFF #C0392BFF #96281BFF
```

Si se obtiene un error indicando que la paleta no existe, puede ser que se haya introducido una paleta continua o discreta en la función incorrecta:

```
paletteer_c(palette = "awtools::a_palette")
## Error in `check_palette()`:
## ! Palette not found. Make sure both package and palette name are spelled correct in the format "packa
```

5.6 Temas

Permite cambiar la apariencia general de la gráfica de acuerdo con parámetros preestablecidos. Solo se tienen que agregar como una función más, por ejemplo `theme_classic()`:

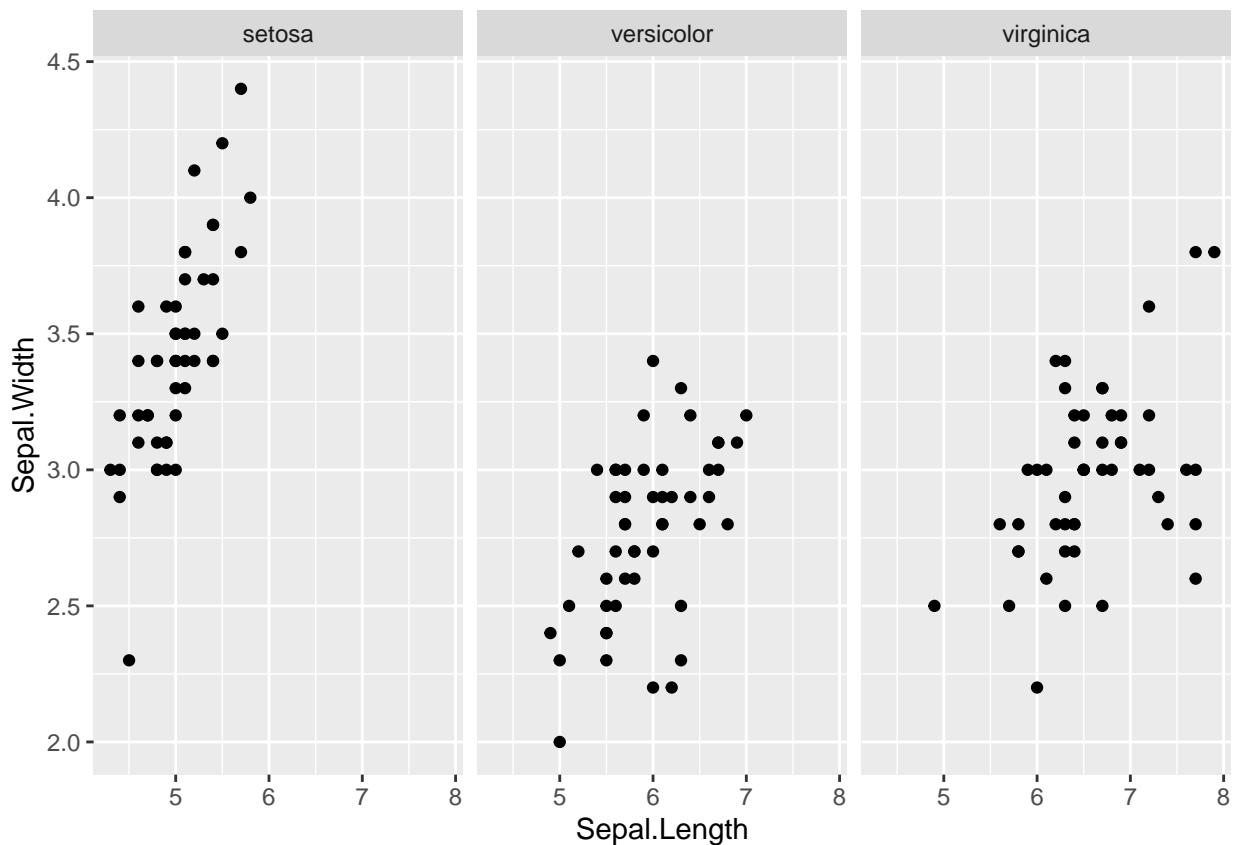


La lista completa de temas puede consultarse con `?theme_classic()` o con el nombre de cualquier otro tema en la lista. Con este manual se provee un ejemplo de cada uno de los temas disponibles.

5.7 Facetting

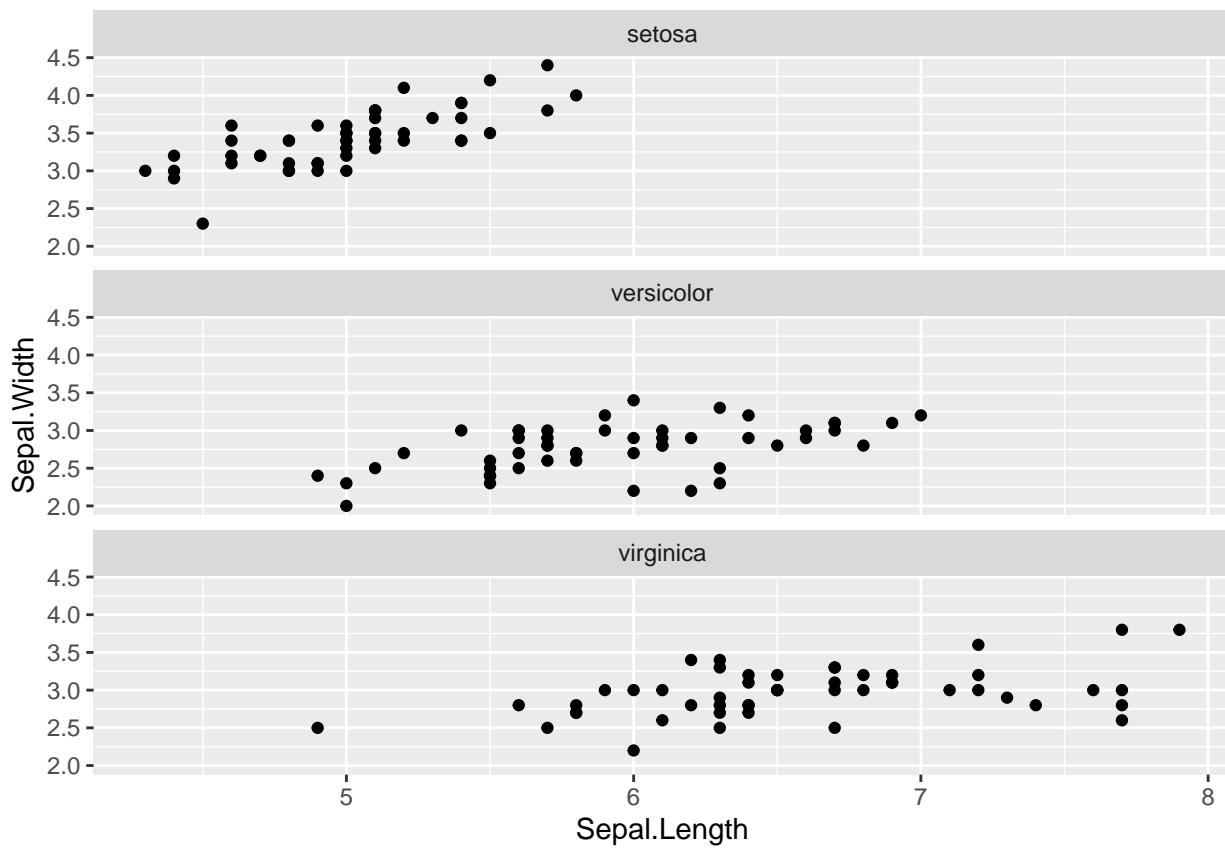
Otra manera de mostrar variables en una gráfica es separando la información en paneles diferentes, de acuerdo con los niveles de una variable categórica. En el caso de `iris`, se pueden separar los datos de cada una de las especies a través de la capa `facet_wrap()`, en la cual se debe colocar un `~` como operador, seguido de la variable que se quiere utilizar para separar los datos:

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point() +
  facet_wrap(~ Species)
```



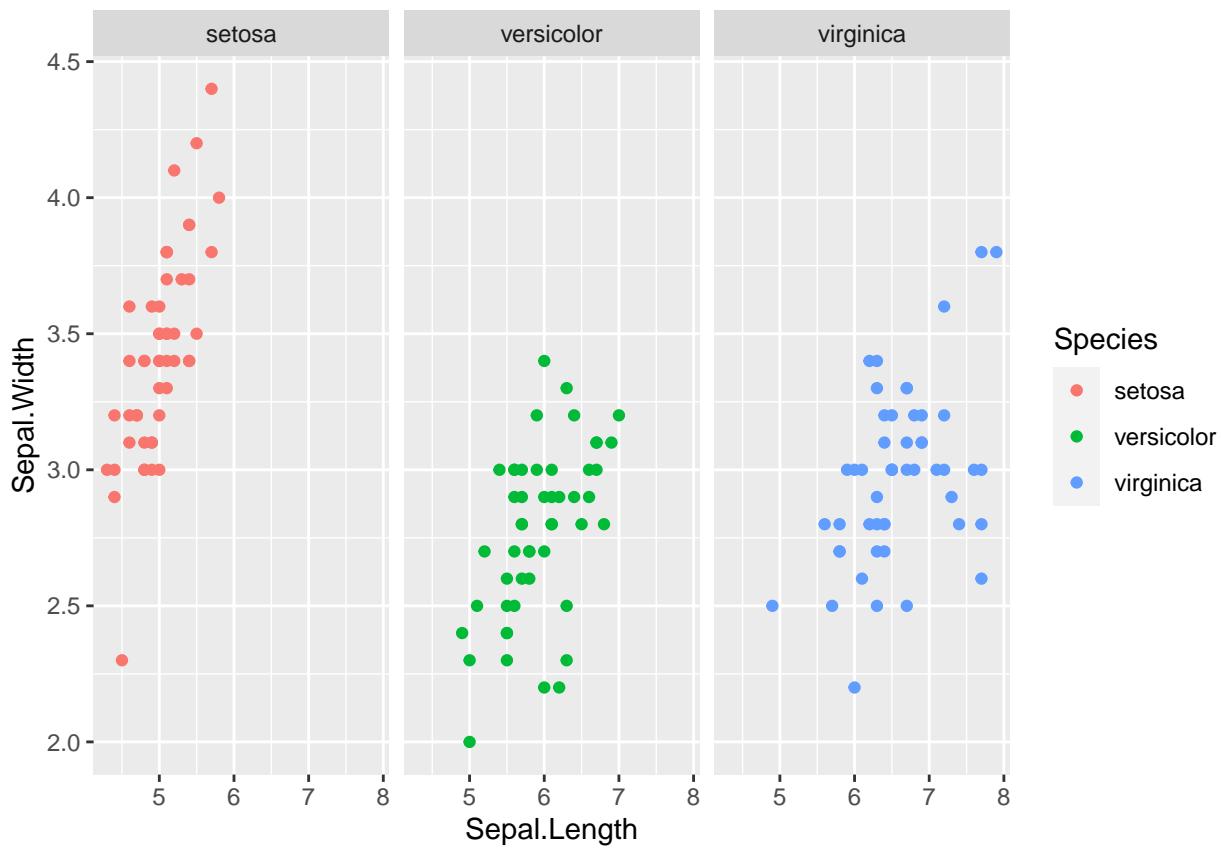
Cada una de los paneles tiene los mismos ejes y el espacio en el que se grafican los datos tiene las mismas dimensiones, lo cual facilita su comparación. En este caso fueron acomodados en una sola fila y tres columnas, este arreglo puede cambiarse en `facet_wrap()`, estableciendo el número de columnas deseadas con `ncol`, o bien, el número de filas con `nrow`:

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point() +
  facet_wrap(~ Species, ncol = 1)
```



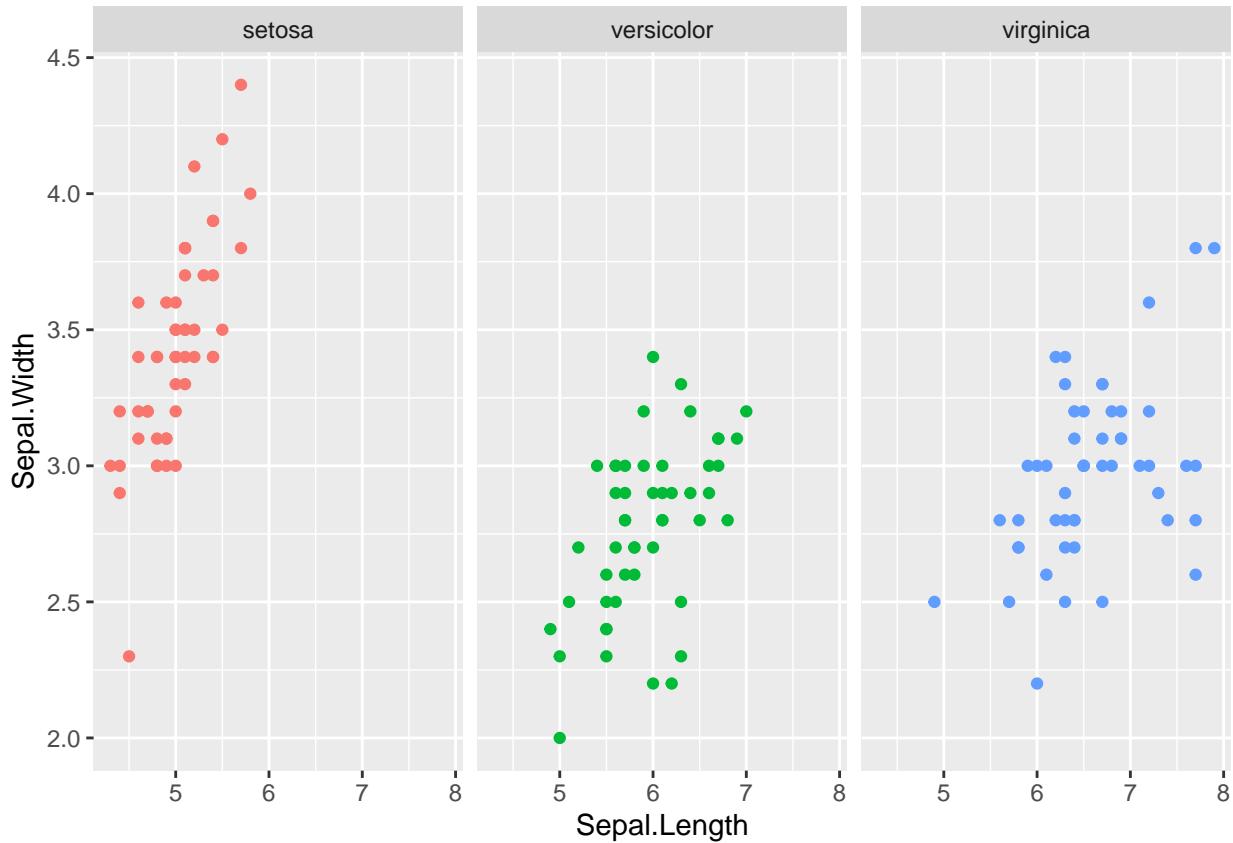
Los datos en cada una de los paneles pueden ser coloreados al colocar `Species` en `aes()`. A pesar de que esto resulta redundante debido a que la gráfica tiene dos separaciones para la variable `Species`, puede resultar útil para hacer más fácil o llamativa la visualización:

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +
  geom_point() +
  facet_wrap(~ Species)
```



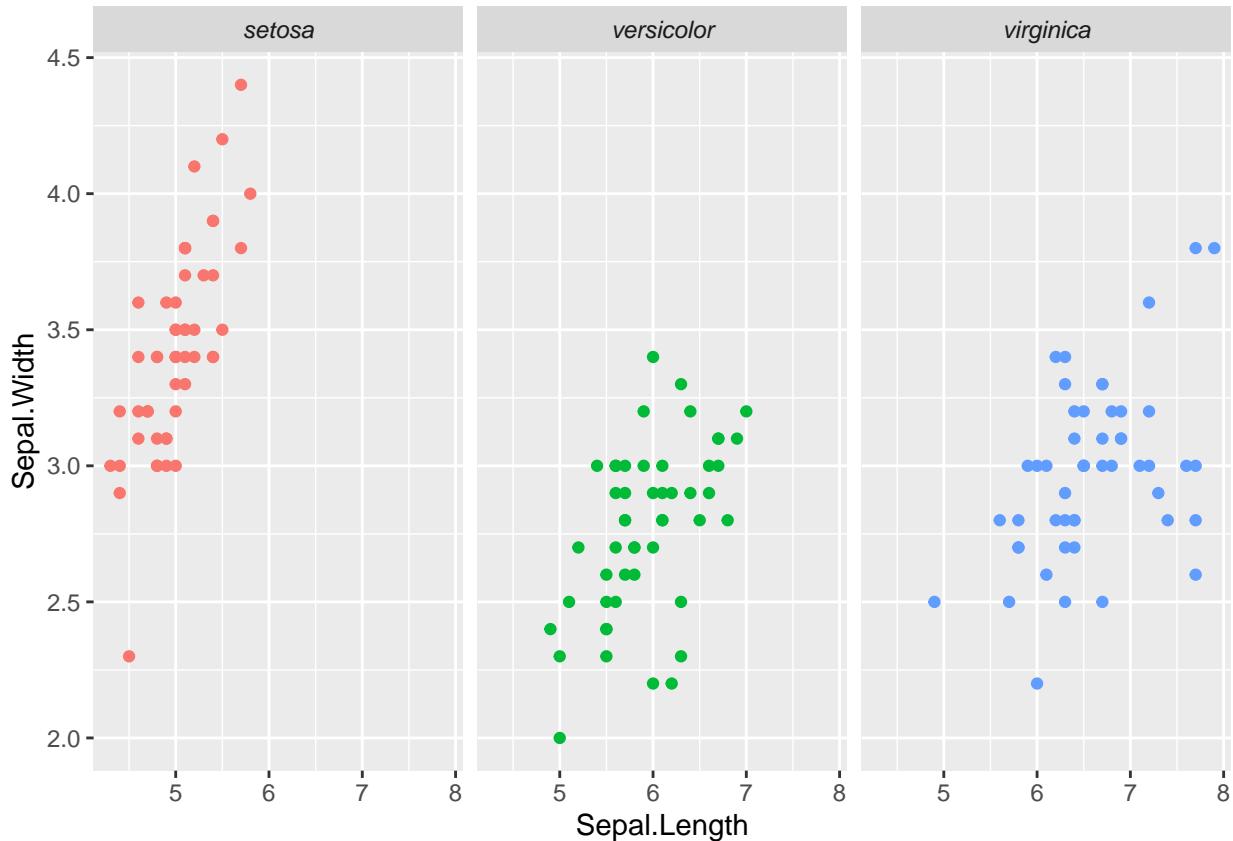
Esta redundancia se presenta en la leyenda, la cual no es necesaria puesto que cada panel indica a que especie pertenecen los datos. Eso se soluciona sencillamente eliminando la leyenda de la gráfica con `theme()`:

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +
  geom_point() +
  facet_wrap(~ Species) +
  theme(legend.position = "none")
```



Si se desean modificar características del texto que identifica cada panel, esta opción se encuentra disponible en `theme()`, mediante la opción `strip.text`, la cual requiere usar `element_text()`:

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width, col = Species)) +
  geom_point() +
  facet_wrap(~ Species) +
  theme(legend.position = "none",
        strip.text = element_text(face = "italic"))
```



Usando paneles es posible incorporar una segunda variable, de tal manera que una de ellas se muestre las filas y otra en las columnas, y la intersección de ambas corresponde al conjunto de datos que coincide en ambas variables. Esto puede ejemplificarse con el data frame `storms`.

Primero es necesario crear un subconjunto con los nombres de algunas tormentas cuyos datos serán utilizados para la gráfica:

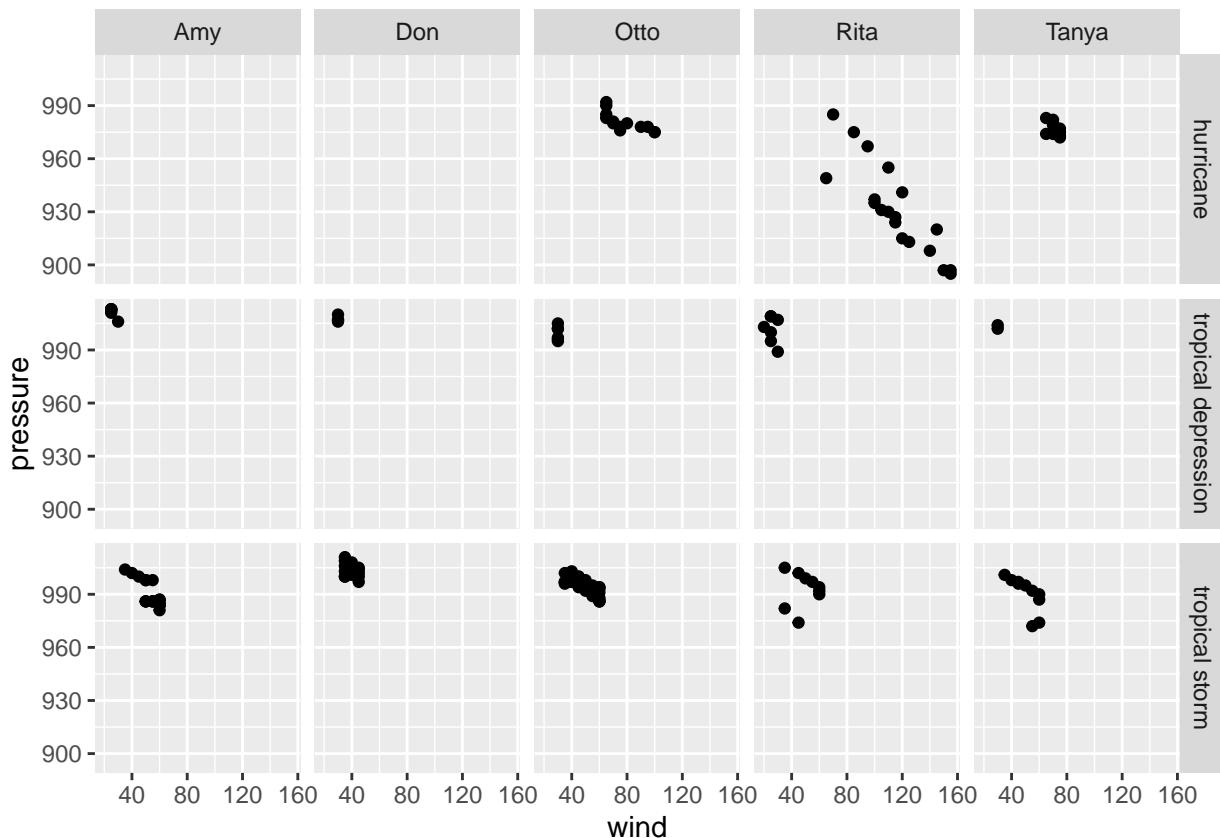
```
tormentas <- storms %>%
  filter(name %in% c("Amy", "Tanya", "Otto", "Rita", "Don"))
```

```
tormentas

## # A tibble: 155 x 13
##   name    year month day hour lat long status      categ~1 wind press~2
##   <chr> <dbl> <dbl> <int> <dbl> <dbl> <chr>      <ord> <int> <int>
## 1 Amy     1975     6     27     0 27.5 -79  tropical dep~-1      25 1013
## 2 Amy     1975     6     27     6 28.5 -79  tropical dep~-1      25 1013
## 3 Amy     1975     6     27    12 29.5 -79  tropical dep~-1      25 1013
## 4 Amy     1975     6     27    18 30.5 -79  tropical dep~-1      25 1013
## 5 Amy     1975     6     28     0 31.5 -78.8 tropical dep~-1      25 1012
## 6 Amy     1975     6     28     6 32.4 -78.7 tropical dep~-1      25 1012
## 7 Amy     1975     6     28    12 33.3 -78  tropical dep~-1      25 1011
## 8 Amy     1975     6     28    18 34    -77  tropical dep~-1      30 1006
## 9 Amy     1975     6     29     0 34.4 -75.8 tropical sto~0      35 1004
## 10 Amy    1975     6     29     6 34    -74.8 tropical sto~0      40 1002
## # ... with 145 more rows, 2 more variables: tropicalstorm_force_diameter <int>,
## #   hurricane_force_diameter <int>, and abbreviated variable names 1: category,
## #   2: pressure
```

La velocidad de viento y presión se colocan en los ejes x y y, mientras que el nombre de la tormenta y su clasificación se separan en paneles con `facet_grid()` usando el operador `~`, de la siguiente manera: `variable_1 ~ variable_2`, el orden es arbitrario, solo indica cuál será organizada en columnas y cual en filas (`filas ~ columnas`):

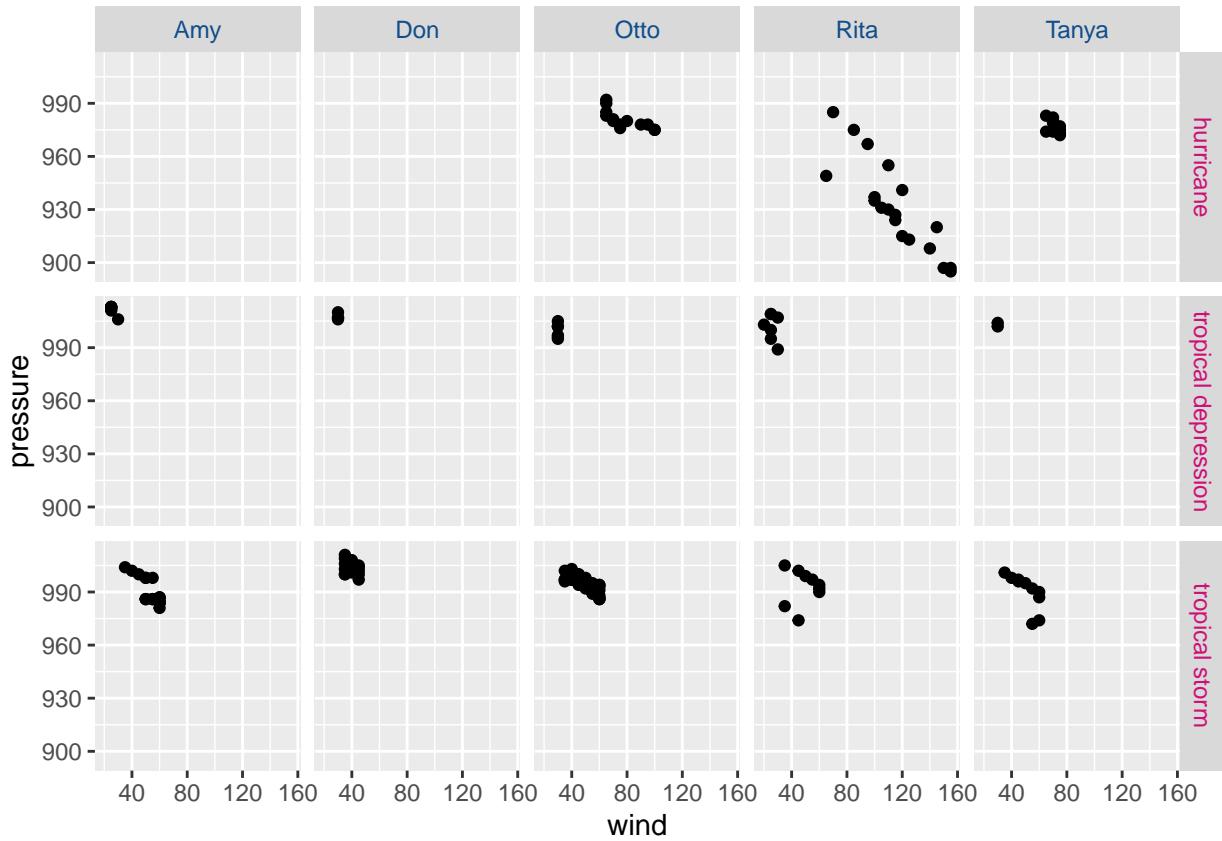
```
ggplot(data = tormentas, aes(x = wind, y = pressure)) +
  geom_point() +
  facet_grid(status ~ name)
```



De esta manera es posible separar los datos no solo de cada tormenta, sino de acuerdo con cada una de sus etapas y es sencillo observar que Amy y Don no se convirtieron en huracanes.

Para modificar el texto que identifica cada panel, aquí es posible separar entre ambos ejes x y y:

```
ggplot(data = tormentas, aes(x = wind, y = pressure)) +
  geom_point() +
  facet_grid(status ~ name) +
  theme(strip.text.y = element_text(color = "deeppink3"),
        strip.text.x = element_text(color = "dodgerblue4"))
```



Usando estas características es solo es posible modificar los aspectos visuales, si se desea cambiar el texto es necesario primero crear un vector cuyos elementos estén nombrados. Los elementos del vector corresponden al nuevo texto y el nombre de cada uno debe ser el nombre original:

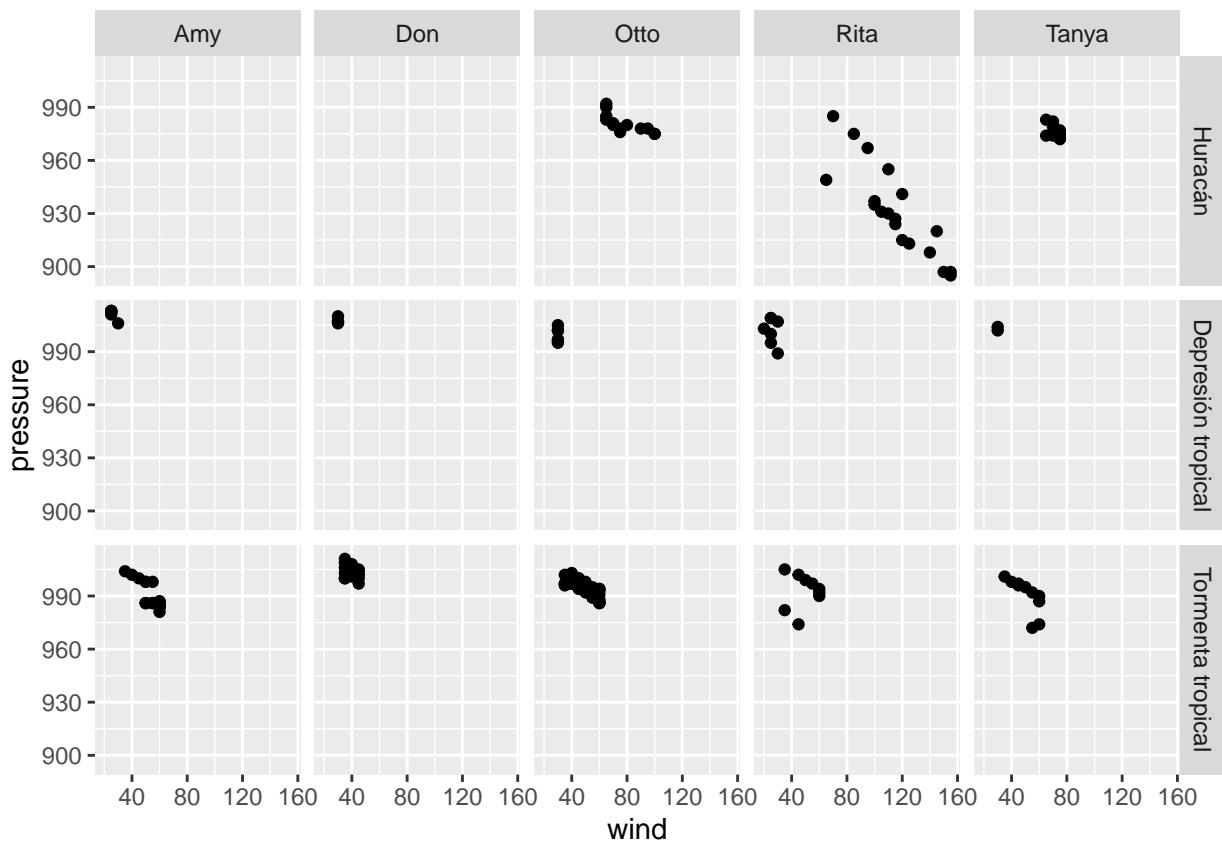
```
tipos_tormentas <- c("Huracán", "Depresión tropical", "Tormenta tropical")
names(tipos_tormentas) <- c("hurricane", "tropical depression", "tropical storm")
```

```
tipos_tormentas
```

```
##           hurricane  tropical depression      tropical storm
##           "Huracán"  "Depresión tropical"  "Tormenta tropical"
```

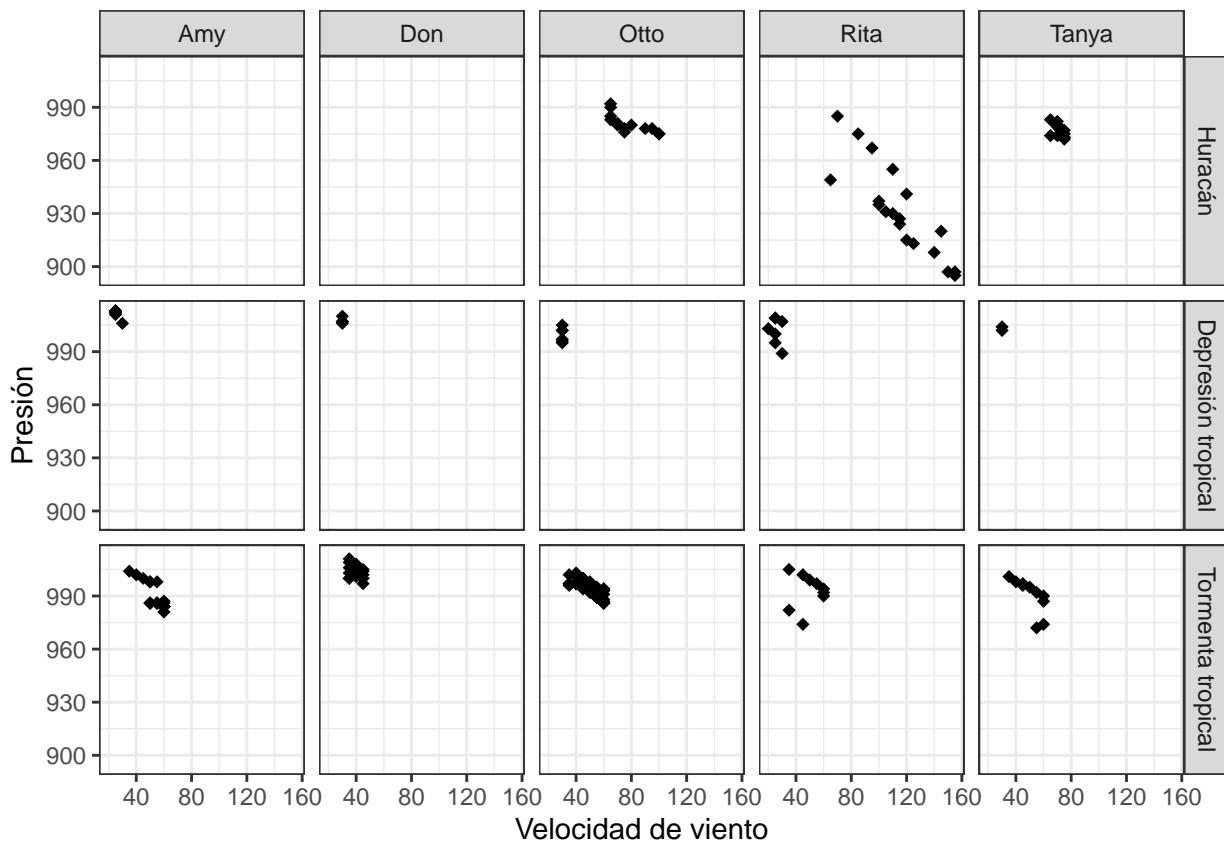
Con este vector se especifican los nuevos nombres dentro de `facet_grid()`, usando el argumento `labeller`, el cual a su vez requiere la función `labeller()`

```
ggplot(data = tormentas, aes(x = wind, y = pressure)) +
  geom_point() +
  facet_grid(status ~ name,
             labeller = labeller(status = tipos_tormentas))
```



Modificando otras características, la gráfica terminada puede verse de la siguiente manera:

```
ggplot(data = tormentas, aes(x = wind, y = pressure)) +
  geom_point(pch = 18, size = 2) +
  facet_grid(status ~ name,
             labeller = labeller(status = tipos_tormentas)) +
  labs(
    x = "Velocidad de viento",
    y = "Presión"
  ) +
  theme_bw()
```



5.8 Exportar gráficas

Una que la gráfica se encuentra tal y como la queremos, es necesario guardar la gráfica como un archivo fuera de R. Se pueden copiar en el portapapeles desde la pestaña Plots en el panel inferior derecho, pero usualmente la resolución no es muy buena. Para solucionar el problema se usa `ggsave()`, en donde se indica el lugar y el nombre con el que se desea guardar la gráfica con el argumento `filename`. En `plot` se coloca el nombre del objeto que se desea exportar y las dimensiones del archivo resultante se especifican con `width` y `height`, de acuerdo con las unidades especificadas en `units`. Finalmente, la resolución se indica con `dpi` (dots per inch).

Primero se debe guardar la gráfica como un objeto en el ambiente global y luego se especifican los argumentos de `ggsave()`:

```
grafica_iris <- ggplot(data = iris, mapping = aes(x = Sepal.Width,
                                                 y = Sepal.Length,
                                                 col = Species)) +
  geom_point() +
  labs(title = "Iris",
       subtitle = "Ancho VS Largo de sépalo",
       x = "Ancho",
       y= "Largo",
       col = "Especies") +
  theme_classic()
```

Si esta gráfica fuera a ser utilizada en una hoja carta, se usan las unidades que se ajusten a ese ancho en centímetros:

```
ggsave(filename = "graficas/iris.png",
       plot = grafica_iris,
```

```
width = 15, height = 10, units = "cm",
dpi = 300)
```

En filename se indicó que la gráfica debe guardarse en la carpeta `graficas`, la cual está en el presente directorio de trabajo. De este mismo modo se pueden indicar otras carpetas dentro de esta, si es que las hubiera. El nombre del archivo debe contar con la extensión deseada, en este caso `.png`. Otros tipos de archivo aceptados son `.eps`, `.ps`, `.tex`, `.pdf`, `.jpeg`, `.tiff`, `.bmp` y `.svg`.

Las gráficas que se desean exportar no necesariamente tiene que ser un objeto en el ambiente global. Se puede simplemente usar `last_plot()` para guardar la última gráfica que se creó:

```
ggsave(filename = "graficas/iris_2.png",
       plot = last_plot(),
       width = 15, height = 10, units = "cm",
       dpi = 300)
```

En general, si se busca que la imagen tenga una buena resolución par impresión la resolución debe ser de 300 DPI. Para un medio digital 72 DPI es adecuado:

```
ggsave(filename = "graficas/iris_3.png",
       plot = last_plot(),
       width = 15, height = 10, units = "cm",
       dpi = 92)
```

En mi opinión, es mejor siempre obtener una imagen con alta resolución para que todos los detalles sean suficientemente claros, por lo que mi resolución preferida es 300 DPI, ya que se ajusta a imágenes de varios tamaños. Sin embargo, esta sugerencia no podría ser ideal para todas las situaciones.

Las imágenes que son exportadas usualmente son muy diferentes a como se muestran en la ventana de R. Seguramente será necesario modificar algunos aspectos hasta que los diferentes atributos de la gráfica se ajusten adecuadamente a la imagen exportada.

5.9 Gráfica de cajas y bigotes

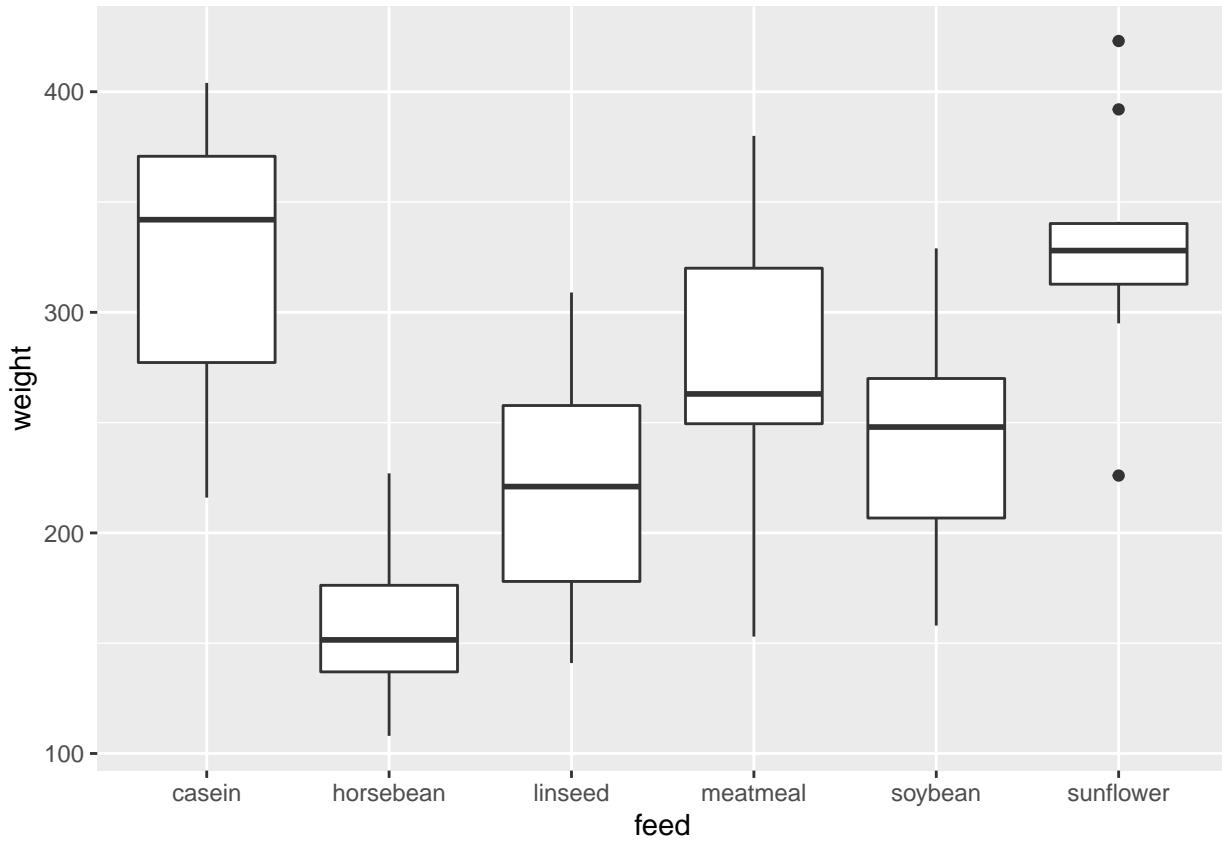
Este tipo de gráfica se emplea para mostrar datos numéricos (eje y) en función de datos categóricos (eje x). Por ejemplo, el data frame `chickwts`:

```
str(chickwts)

## 'data.frame':    71 obs. of  2 variables:
## $ weight: num  179 160 136 227 217 168 108 124 143 140 ...
## $ feed   : Factor w/ 6 levels "casein","horsebean",...: 2 2 2 2 2 2 2 2 2 2 ...
```

Se puede colocar el peso de los pollitos en función del tipo de alimento. La función necesaria es `geom_boxplot()`:

```
ggplot(data = chickwts, aes(x = feed, y = weight)) +
  geom_boxplot()
```



Cada una de las cajas muestra el rango de los datos asociados con cada categoría. En cada una de las categorías se muestra el, mínimo y máximo de cada subconjunto de datos, así como el primer cuartil, la mediana y el tercer cuartil:

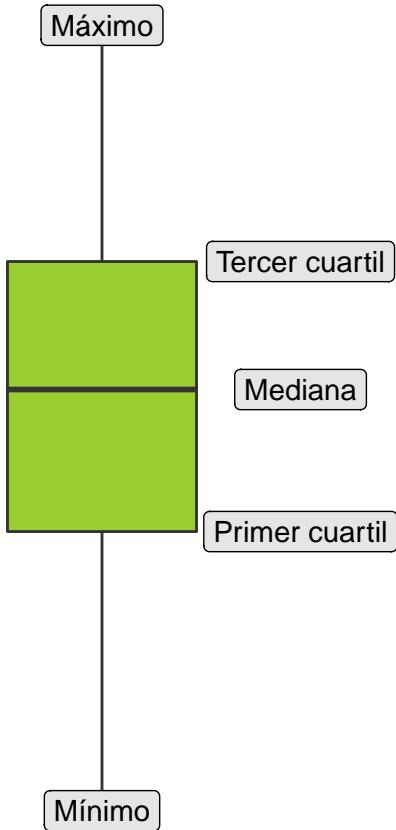
```

y <- rnorm(n = 100, mean = 10, sd = .1)
cajas_y_bigotes <- tibble(x = rep(x = 2, times = 100),
                           y = y)

cuartiles <- tibble(y = quantile(x = y, probs = c(.25, .5, .75)),
                      nombre = c("Primer cuartil", "Mediana", "Tercer cuartil"))
rango <- tibble(y = c(min(y), max(y)),
                  nombre = c("Mínimo", "Máximo"))

ggplot() +
  geom_boxplot(data = cajas_y_bigotes, aes(x = x, y = y),
               width = 1/3, fill = "olivedrab3") +
  xlim(1, 3) +
  geom_label(data = cuartiles, aes(x = 2.35, y = y,
                                    label = nombre),
             fill = "gray90") +
  geom_label(data = rango, aes(x = 2, y = y,
                               label = nombre),
             fill = "gray90") +
  theme_void()

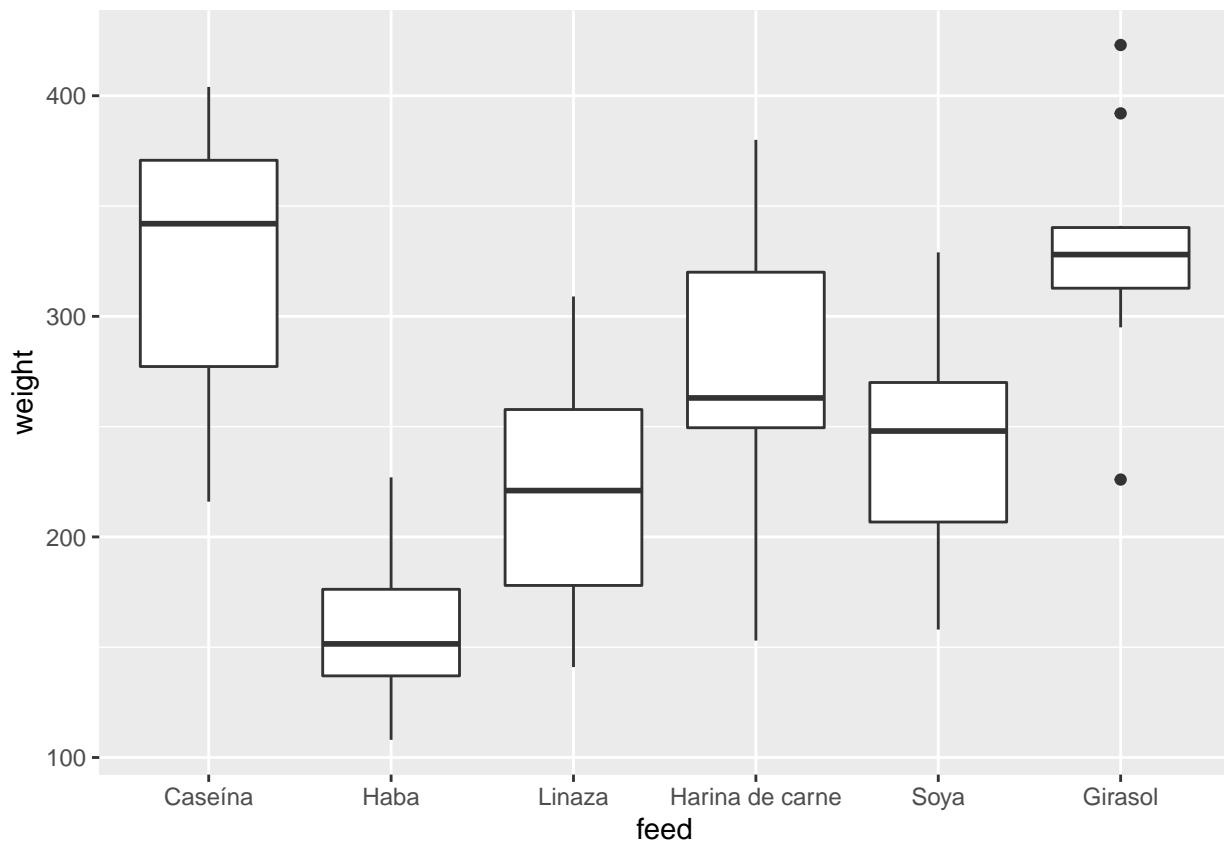
```



El espacio comprendido por al primer y tercer cuartil (la caja en color verde) es el intercuartil, en donde se encuentran la mitad de las observaciones y la otra mitad está distribuida en ambos extremos por encima y por debajo del intercuartil. En los datos correspondientes a `sunflower` se observan puntos fuera de los bigotes debido a que son valores extremos. La extensión máxima de los bigotes es de 1.5 veces el rango intercuartil por debajo del primer cuartil o por encima del tercero, de modo que aquellos datos que excedan este valor serán clasificados como datos extremos por `geom_boxplot()`.

Uno de los aspectos que pueden modificarse de la gráfica anterior es el nombre de las categorías para que se muestren en español en lugar de tomar las categorías del data frame. Se debe usar un vector con los nuevos nombres, los cuales deben ser escritos en el mismo orden en el que aparecen en la gráfica anterior. Este vector se introduce `scale_x_discrete()` en el argumento `labels` :

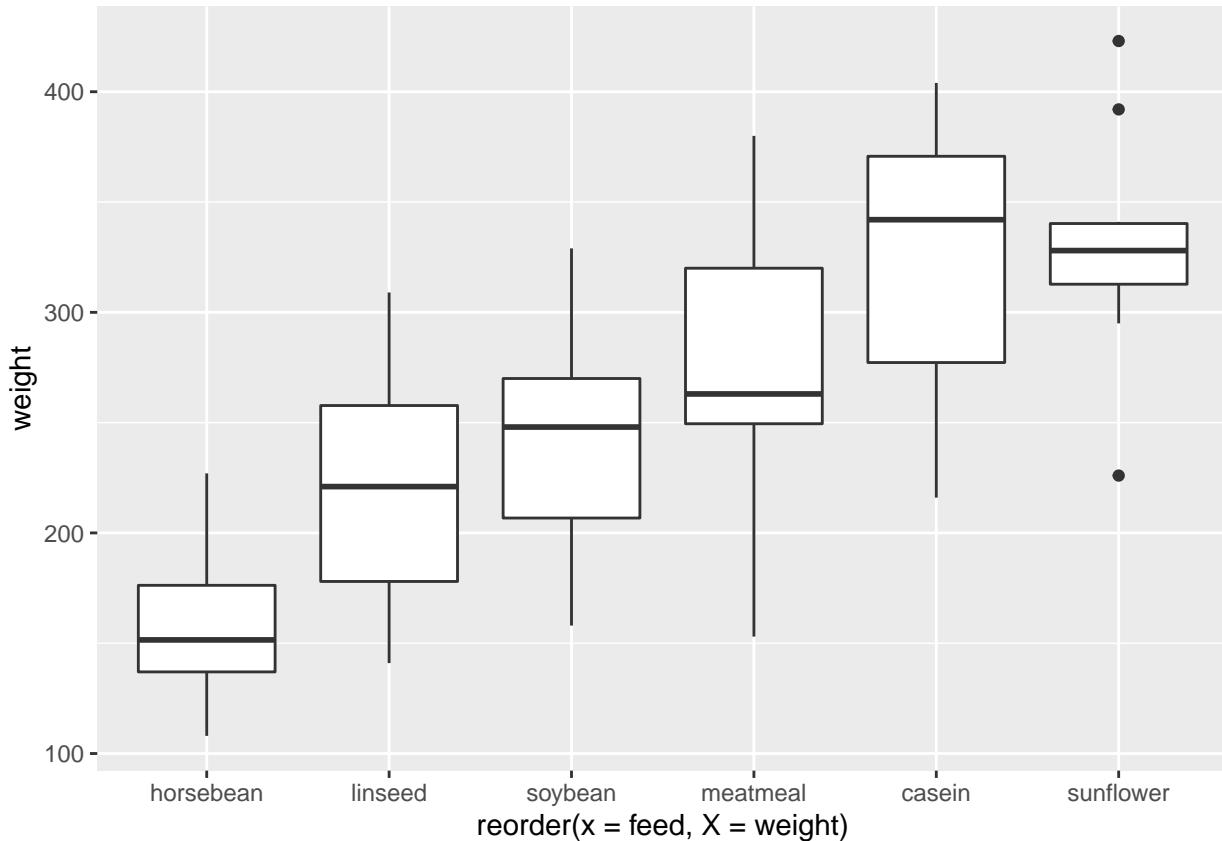
```
ggplot(data = chickwts, aes(x = feed, y = weight)) +
  geom_boxplot() +
  scale_x_discrete(labels = c("Caseína", "Haba",
                             "Linaza", "Harina de carne",
                             "Soya", "Girasol"))
```



Es importante respetar el orden de la gráfica inicial debido a que los nuevos nombres son arbitrarios y no están asociados con los datos originales.

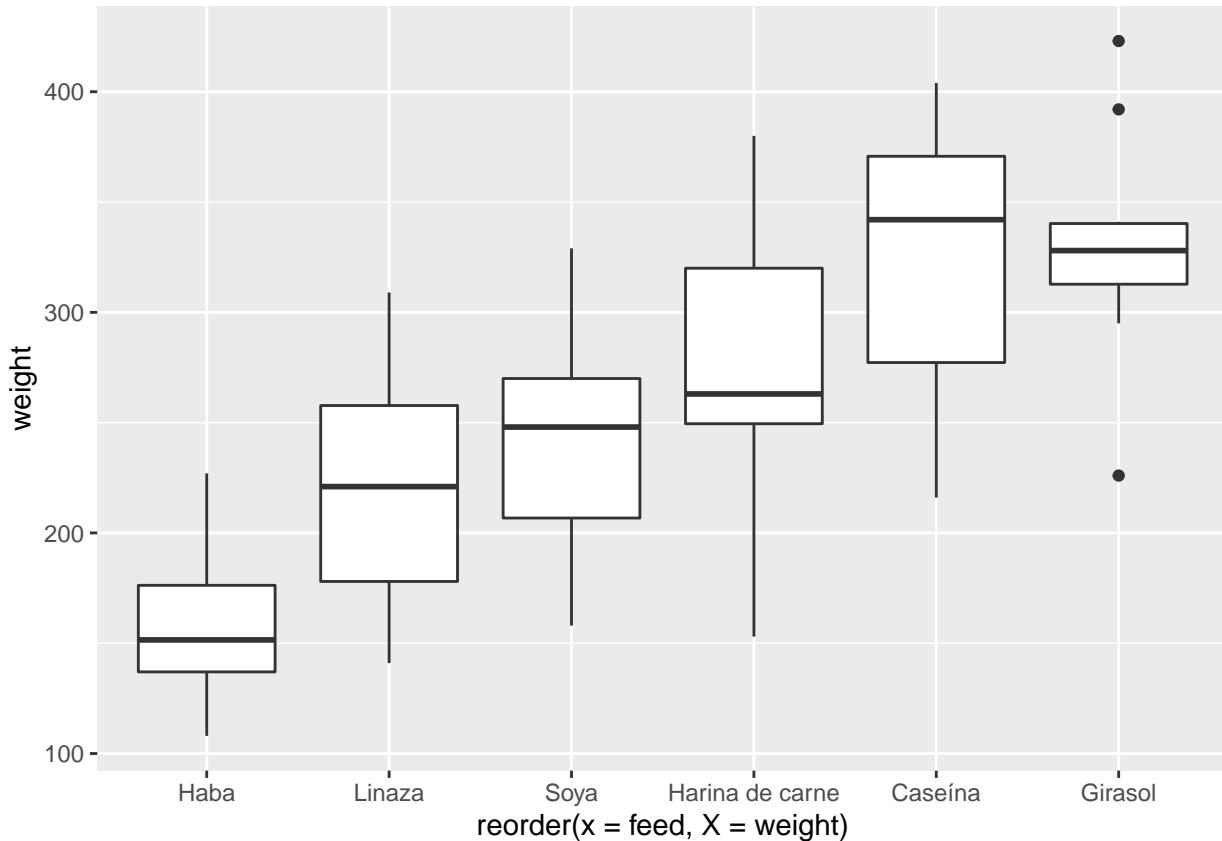
Si se desea representar las categorías en orden creciente debe de modificarse la manera en que se especifica la variable categórica en `aes()` con `reorder()`. Esta función requiere colocar las categorías en el argumento `x` y los valores numéricos en `X` (mayúscula), para adecuadamente indicar que la variable numérica debe ordenarse para cada categoría del factor:

```
ggplot(data = chickwts, aes(x = reorder(x = feed, X = weight),
                             y = weight)) +
  geom_boxplot()
```



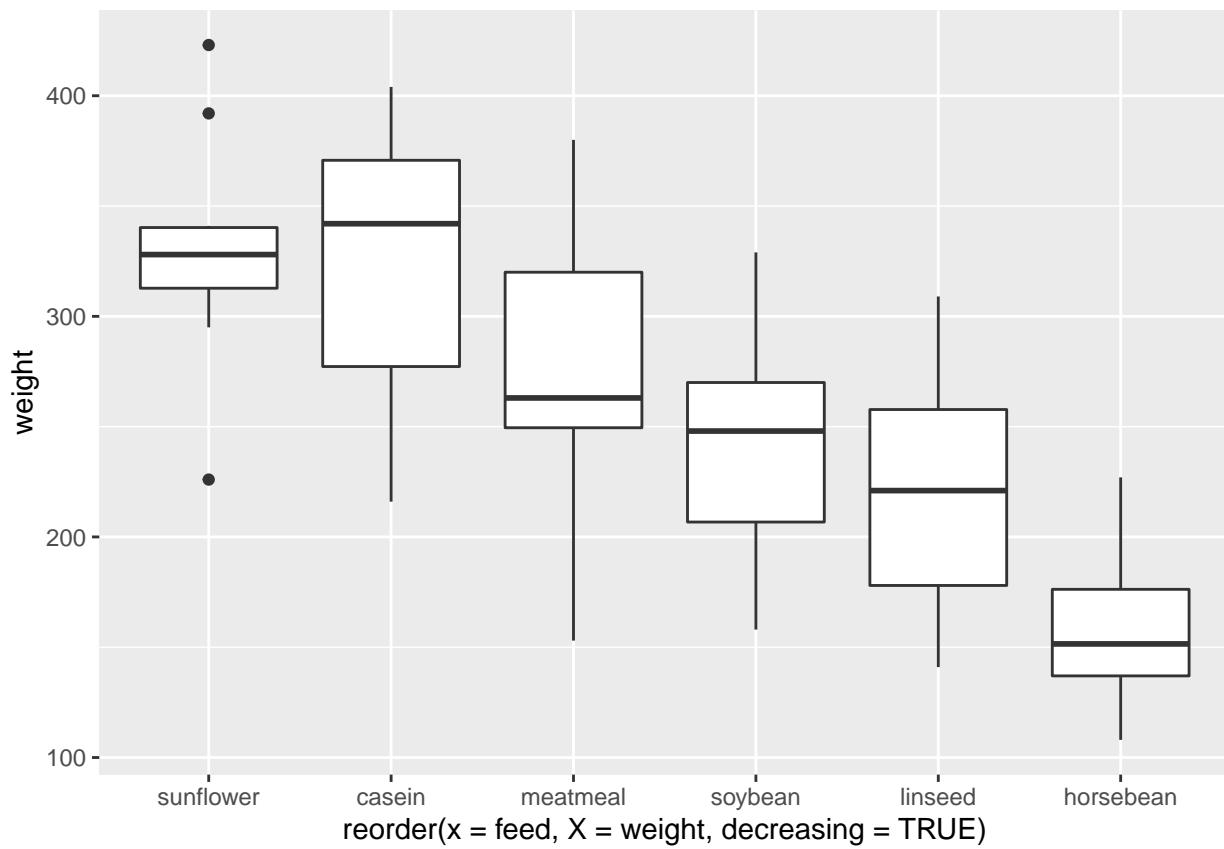
Con este nuevo orden se debe cambiar también el vector usado en `scale_x_discrete()`:

```
ggplot(data = chickwts, aes(x = reorder(x = feed, X = weight),
                             y = weight)) +
  geom_boxplot() +
  scale_x_discrete(labels = c("Haba", "Linaza", "Soya",
                             "Harina de carne", "Caseína",
                             "Girasol"))
```



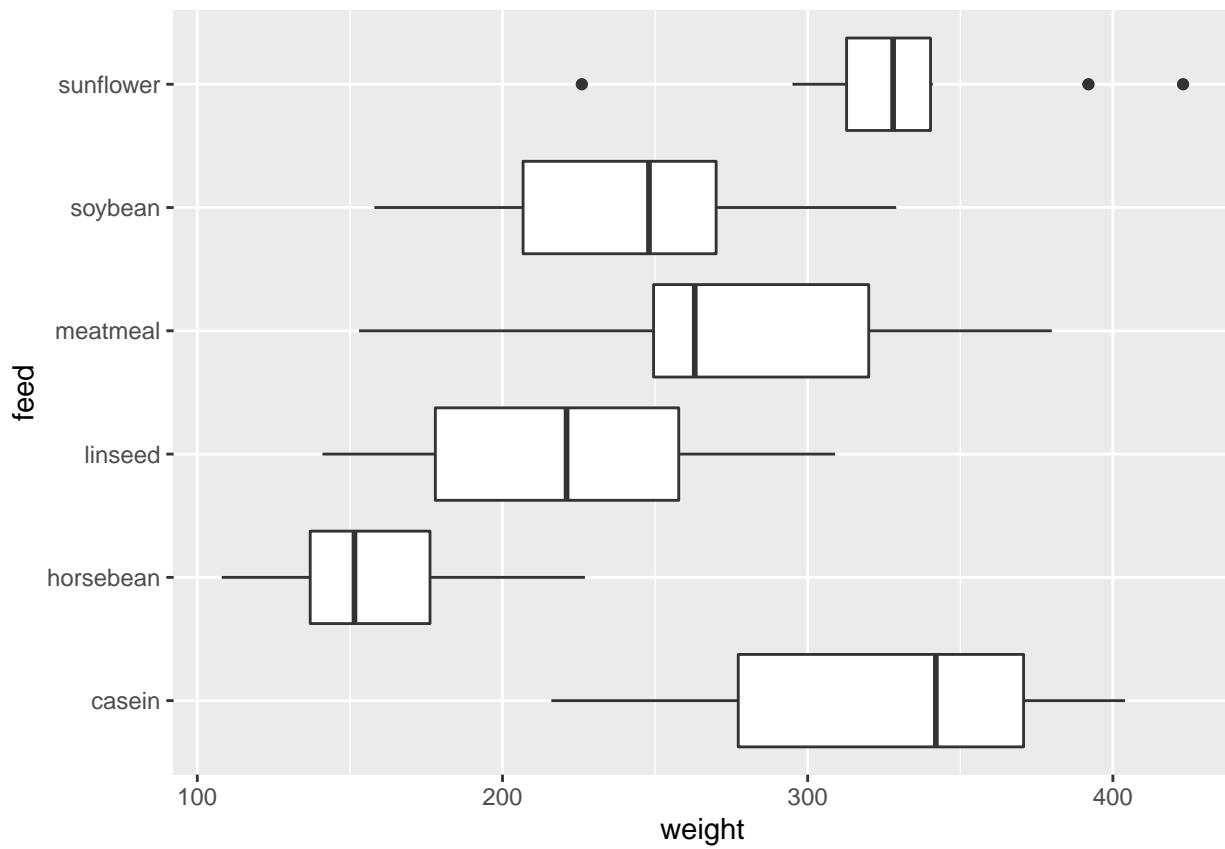
Si se mantuviera el vector usado en la primer gráfica, las categorías tendrían estarían mal nombradas ya que `scale_x_discrete()` no se ajusta a los cambios realizados en `aes()`. En caso de que el orden deseado sea decreciente se cambia el valor de `decreasing`, que por defecto es falso:

```
ggplot(data = chickwts, aes(x = reorder(x = feed, X = weight,
                                         decreasing = TRUE),
                                         y = weight)) +
  geom_boxplot()
```



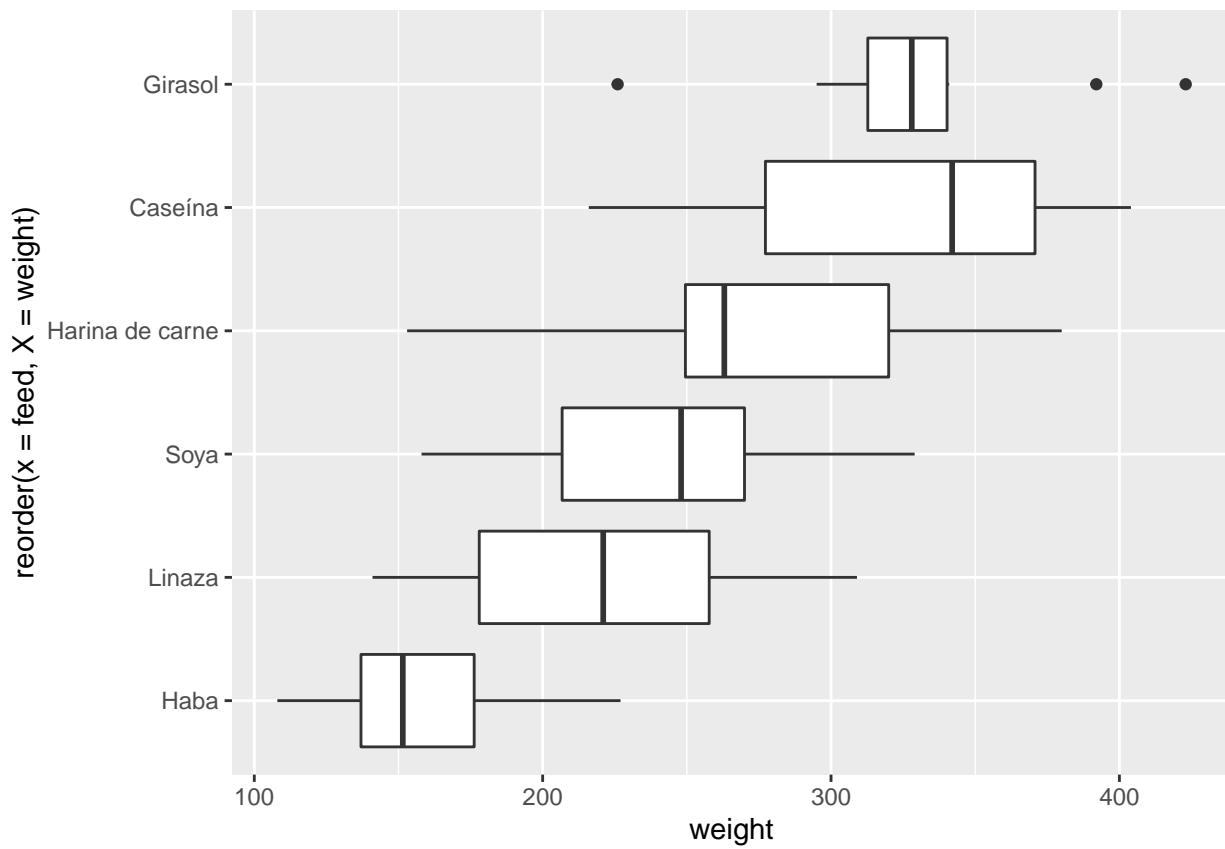
La gráfica también puede mostrarse horizontalmente al agregar la función `coord_flip()`:

```
ggplot(data = chickwts, aes(x = feed, y = weight)) +
  geom_boxplot() +
  coord_flip()
```



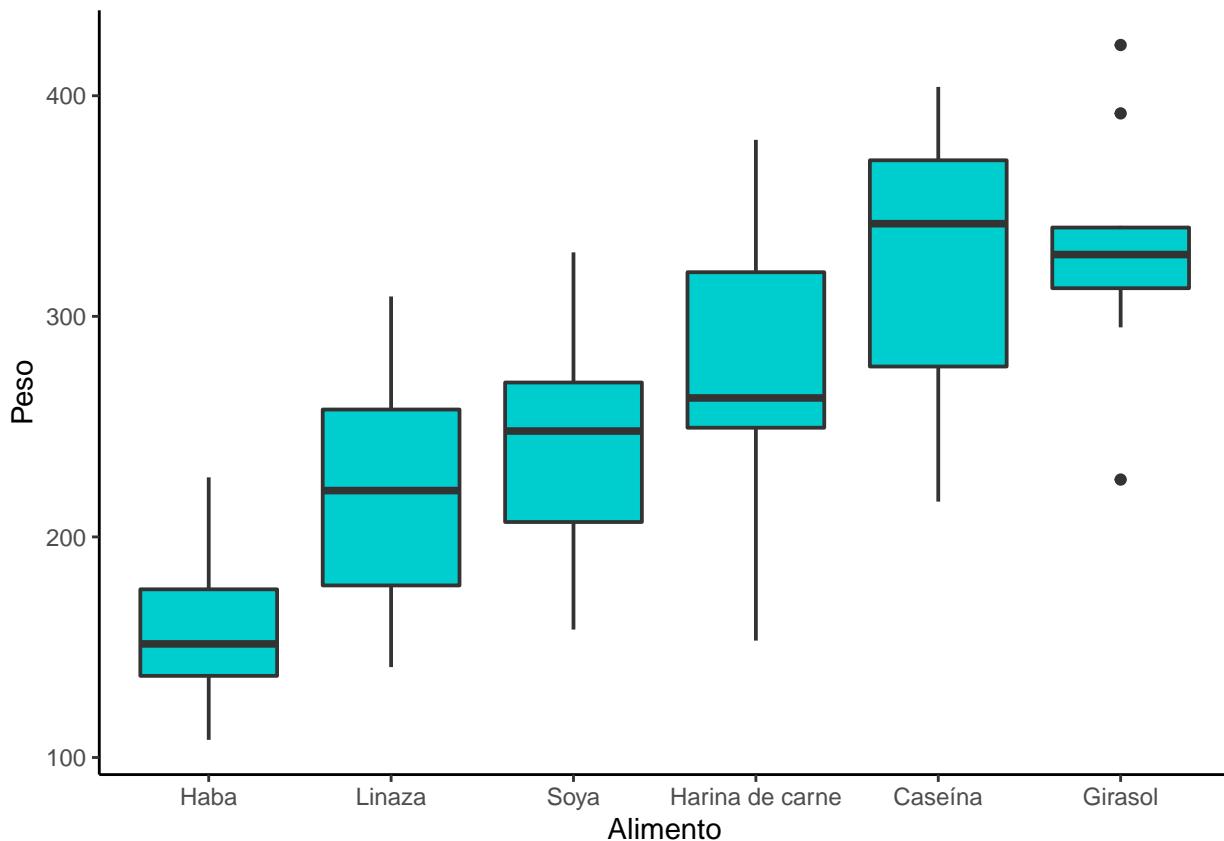
La posición horizontal puede combinarse con las categorías reorganizadas:

```
ggplot(data = chickwts, aes(x = reorder(x = feed, X = weight),
                             y = weight)) +
  geom_boxplot() +
  scale_x_discrete(labels = c("Haba", "Linaza", "Soya",
                             "Harina de carne", "Caseína",
                             "Girasol")) +
  coord_flip()
```



Una gráfica terminada podría verse de la siguiente manera:

```
ggplot(data = chickwts, aes(x = reorder(x = feed, X = weight),
                             y = weight)) +
  geom_boxplot(fill = "cyan3", size = 2/3) +
  scale_x_discrete(labels = c("Haba", "Linaza", "Soya",
                             "Harina de carne", "Caseína",
                             "Girasol")) +
  labs(x = "Alimento",
       y = "Peso") +
  theme_classic()
```



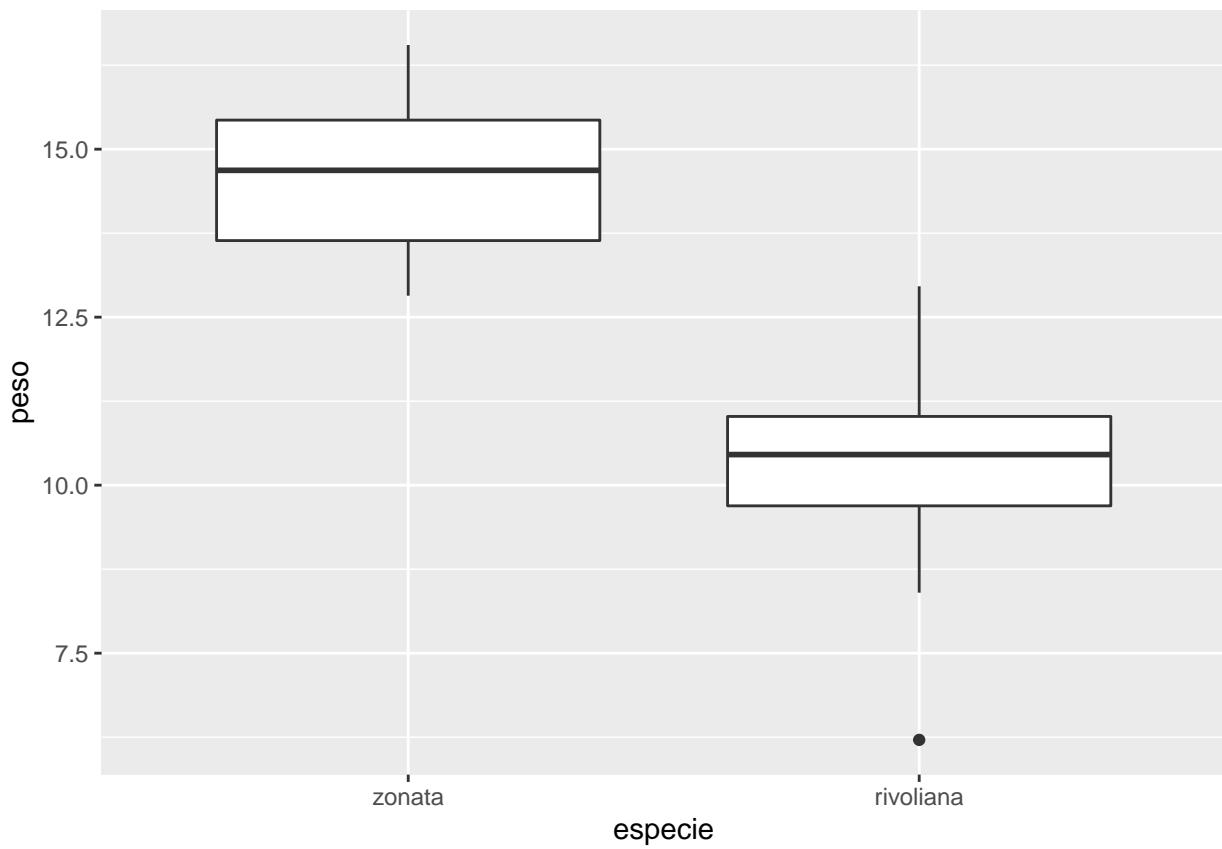
Los datos que se busca representar mediante este tipo de gráficas pueden estar asociados con más de una variable categórica. La manera de incluir variables adicionales es a través del uso de estéticas, principalmente color. El siguiente conjunto de datos contiene pesos de peces del género *Seriola* clasificados de acuerdo con su sexo y especie:

```
pesos_seriola <- read_csv(file = "datos_manual/pesos_seriola.csv",
                           col_types = "nff")
str(pesos_seriola)

## # spec_tbl_df [80 x 3] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
## $ peso    : num [1:80] 15.1 15.4 15.8 15.4 16 ...
## $ especie: Factor w/ 2 levels "zonata","rivoliana": 1 1 1 1 1 1 1 1 1 ...
## $ sexo    : Factor w/ 2 levels "masculino","femenino": 1 1 1 1 1 1 1 1 1 ...
## - attr(*, "spec")=
##   .. cols(
##     ..   peso = col_number(),
##     ..   especie = col_factor(levels = NULL, ordered = FALSE, include_na = FALSE),
##     ..   sexo = col_factor(levels = NULL, ordered = FALSE, include_na = FALSE)
##     .. )
## - attr(*, "problems")=<externalptr>
```

Graficando únicamente el peso para los individuos de cada especie muestra la siguiente gráfica:

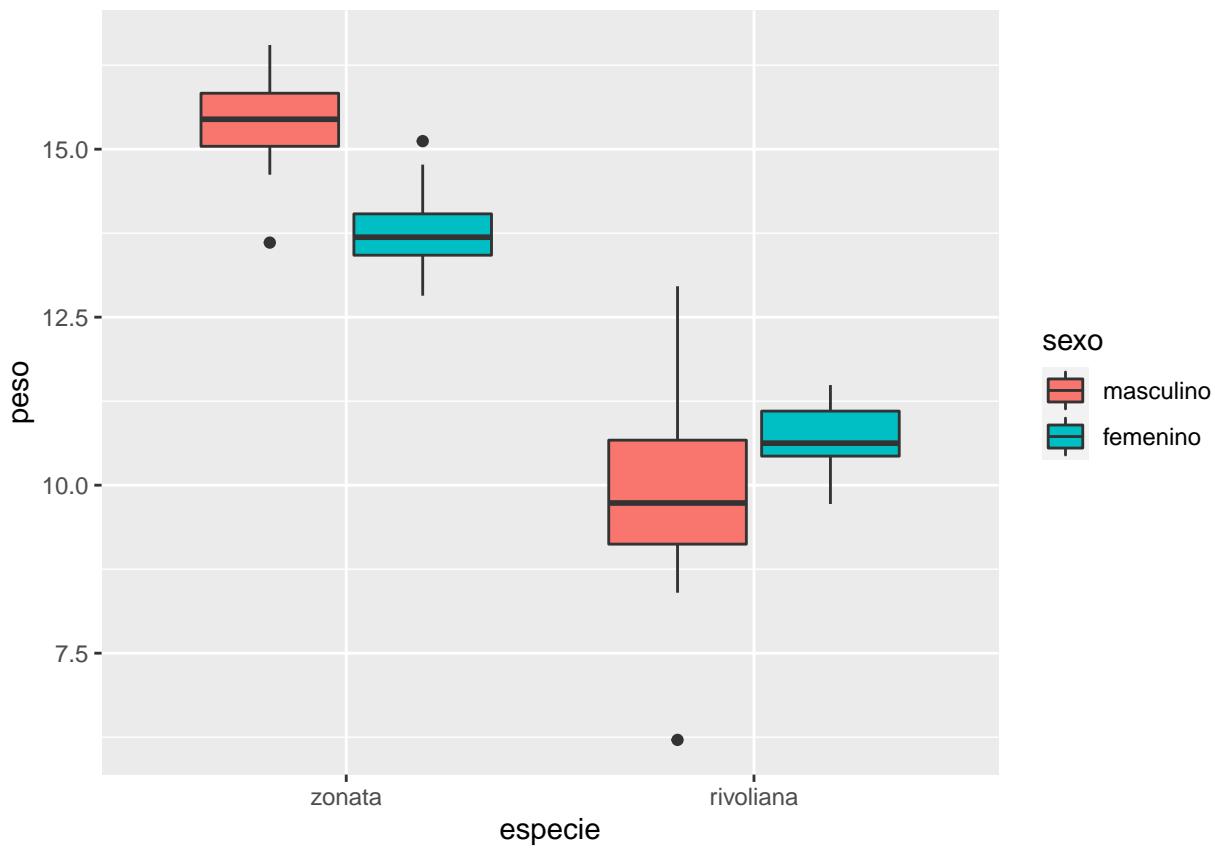
```
ggplot(data = pesos_seriola, aes(x = especie, y = peso)) +
  geom_boxplot()
```



Aquí se puede ver que los individuos de *S. zonata* pesan más que los de *S. rivoliana*. El objeto `pesos_seriola` también cuenta con información sobre el sexo de cada uno de los individuos pesados.

Para ver los datos de cada especie de acuerdo con el sexo, se debe indicar dentro de `aes()` que `fill` se usará para indicar el sexo:

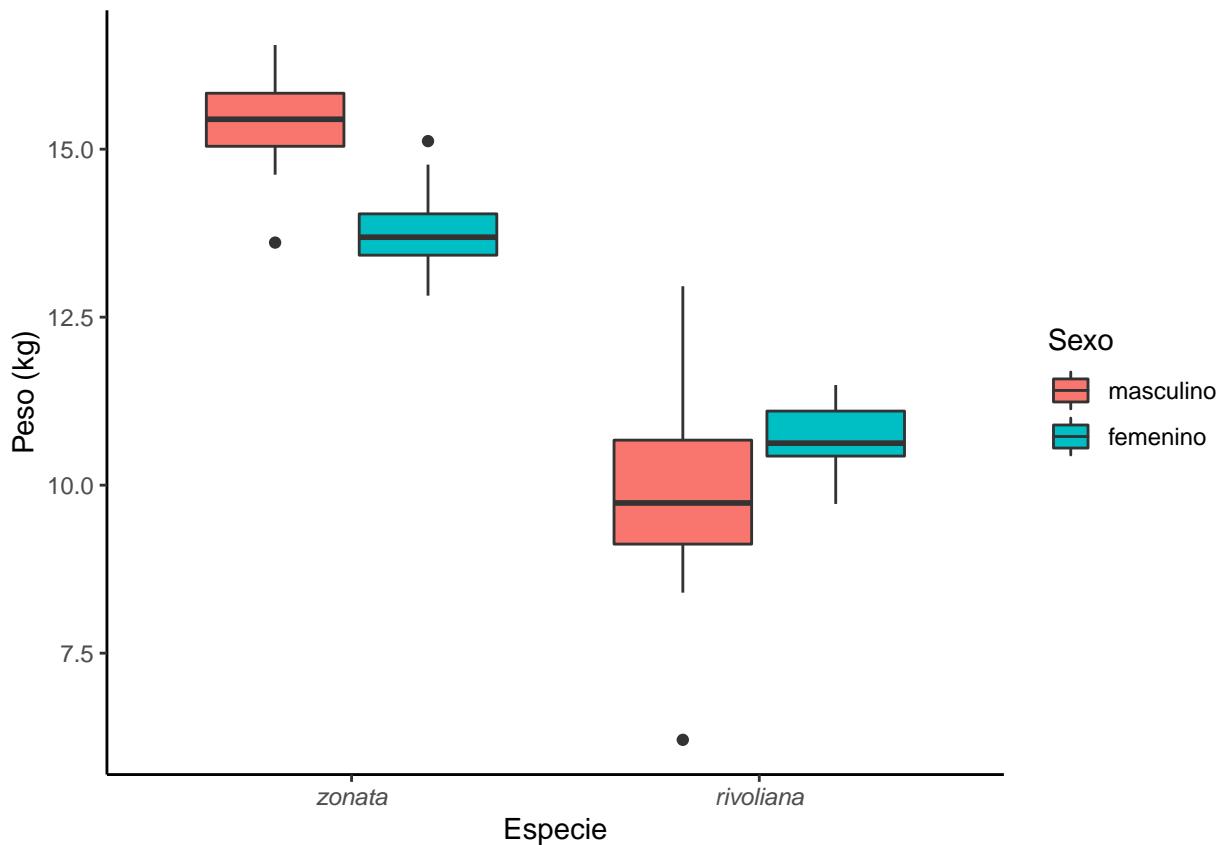
```
ggplot(data = pesos_seriola, aes(x = especie, y = peso,
                                    fill = sexo)) +
  geom_boxplot()
```



Ahora es posible observar que los machos de *S. zonata* pesan más que las hembras, pero es el caso contrario en *S. rivoliana*. Debido a que el rango de las hembras se encuentra por encima de la mediana de los machos, podemos ver a partir de la gráfica que las hembras pesan más que la mitad de los machos que fueron pesados.

La gráfica terminada:

```
ggplot(data = pesos_seriola, aes(x = especie, y = peso,
                                    fill = sexo)) +
  geom_boxplot() +
  labs(x = "Especie",
       y = "Peso (kg)",
       fill = "Sexo") +
  theme_classic() +
  theme(axis.text.x = element_text(face = "italic"))
```



Debido a que las gráficas de cajas y bigotes no muestran observaciones individuales, no es posible usar el tamaño o el tipo de carácter para incluir más variables en la gráfica. En cambio, se pueden usar paneles para separar la información de acuerdo con las categorías de un factor.

El siguiente conjunto de datos muestra el peso seco de plantas de acuerdo con diferentes sitios y tratamientos para cada sitio. Además, los datos están clasificados dependiendo si el peso registrado corresponde a la parte aérea o subterránea de la planta:

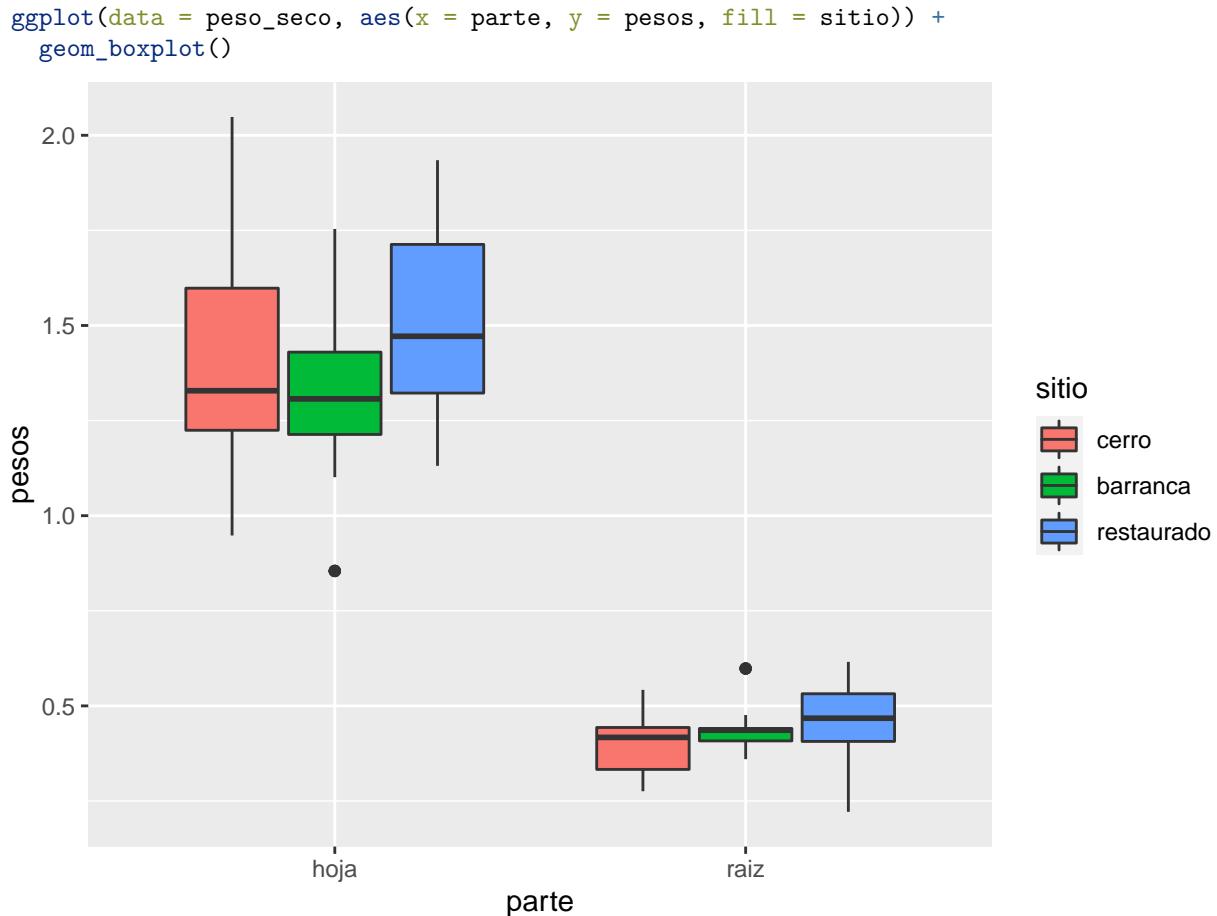
```

peso_seco <- read_csv(file = "datos_manual/peso_seco.csv",
                      col_types = "nfff")
str(peso_seco)

## # spec_tbl_df [288 x 4] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
## # $ pesos      : num [1:288] 1.54 1.34 1.13 1.27 1.16 ...
## # $ parte      : Factor w/ 2 levels "hoja","raiz": 1 1 1 1 1 1 1 1 1 ...
## # $ sitio       : Factor w/ 3 levels "cerro","barranca",...: 1 2 3 1 2 3 1 2 3 1 ...
## # $ tratamiento: Factor w/ 3 levels "A","B","C": 1 1 1 1 2 2 2 3 3 ...
## # - attr(*, "spec")=
## #   .. cols(
## #     ..   pesos = col_number(),
## #     ..   parte = col_factor(levels = NULL, ordered = FALSE, include_na = FALSE),
## #     ..   sitio = col_factor(levels = NULL, ordered = FALSE, include_na = FALSE),
## #     ..   tratamiento = col_factor(levels = NULL, ordered = FALSE, include_na = FALSE)
## #     .. )
## # - attr(*, "problems")=<externalptr>

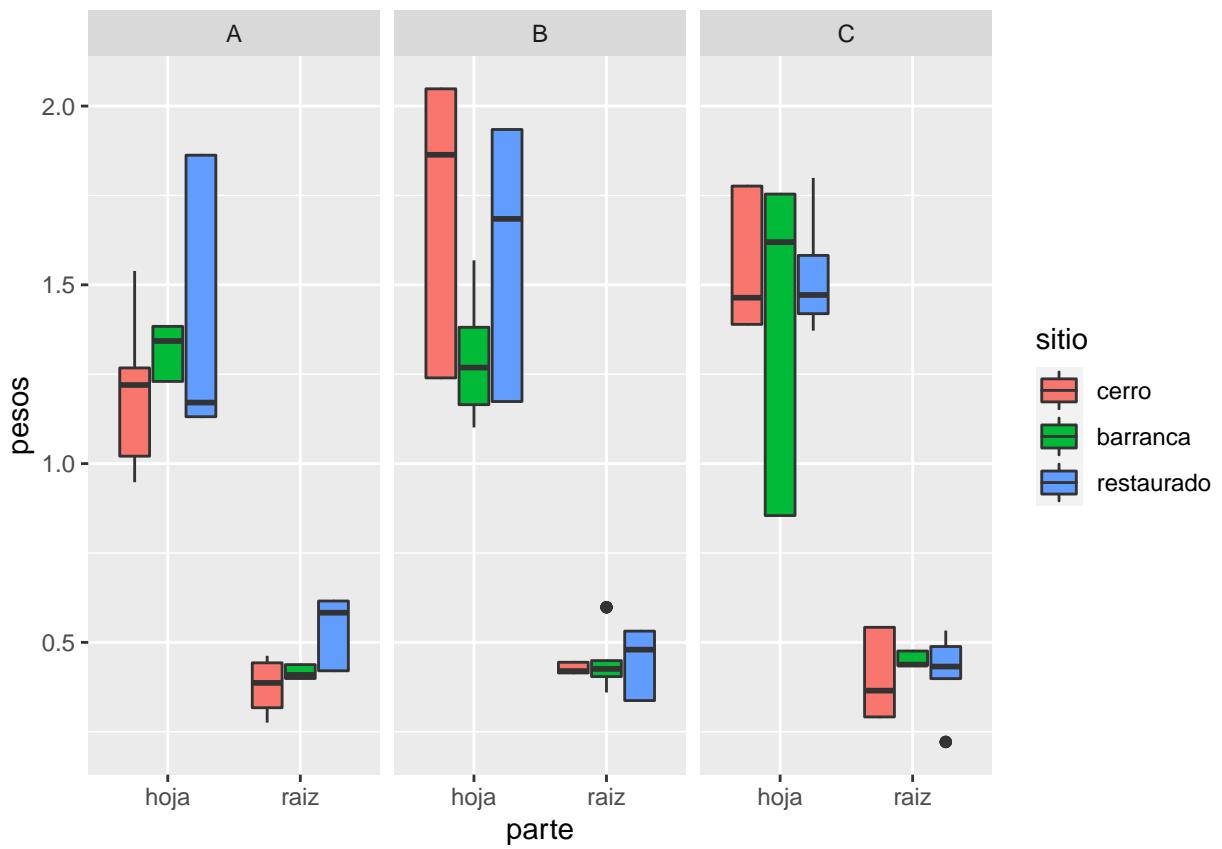
```

En este conjunto de datos, la parte de la planta y el sitio se muestran en el eje x y a través del color de relleno:



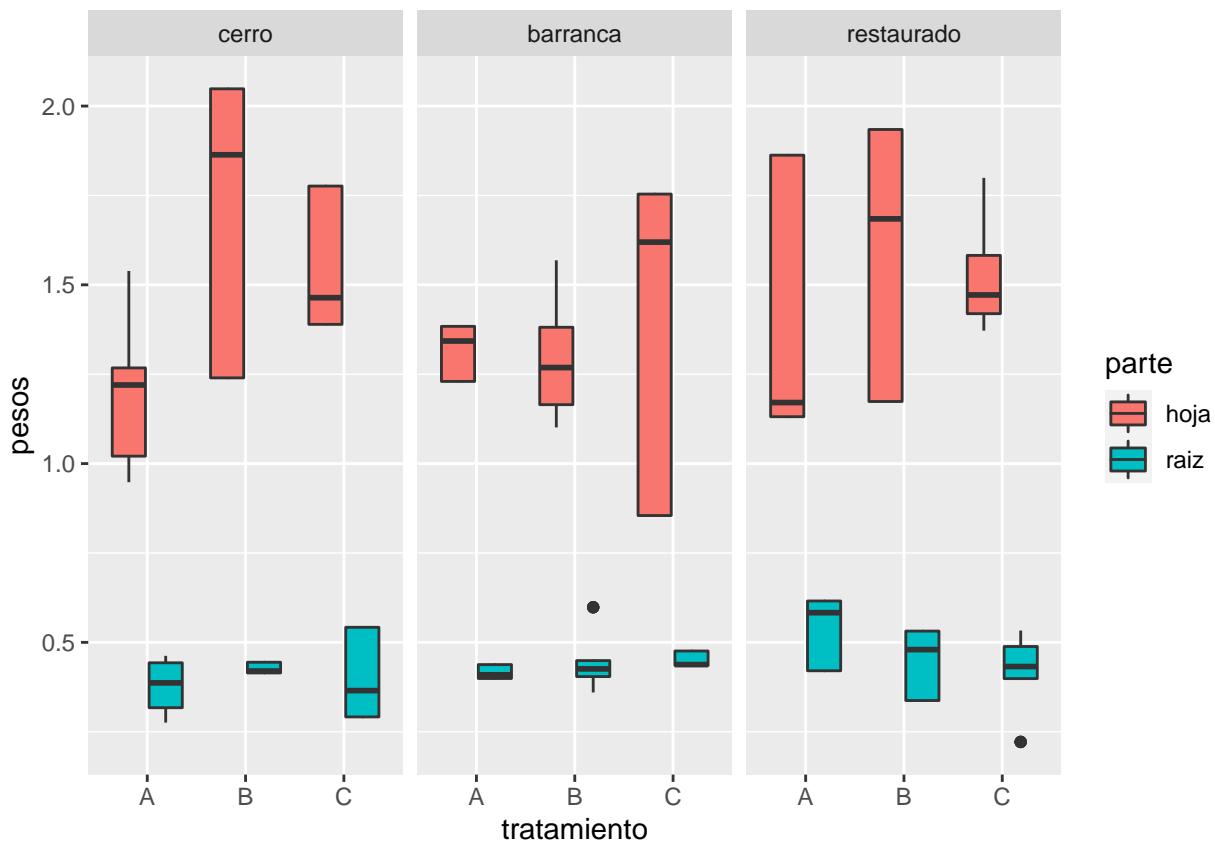
Para separar los datos en función del tratamiento se usa la capa `facet_wrap()`, en la cual se coloca el operador `~` y después la variable que se usará para separar los datos:

```
ggplot(data = peso_seco, aes(x = parte, y = pesos, fill = sitio)) +
  geom_boxplot() +
  facet_wrap(~ tratamiento)
```



Las variables pueden ser intercambiadas para separar los datos de acuerdo con diferentes variables, por ejemplo, colocando el tratamiento en el eje x, la parte de la planta con el color y separando en paneles de acuerdo con el sitio:

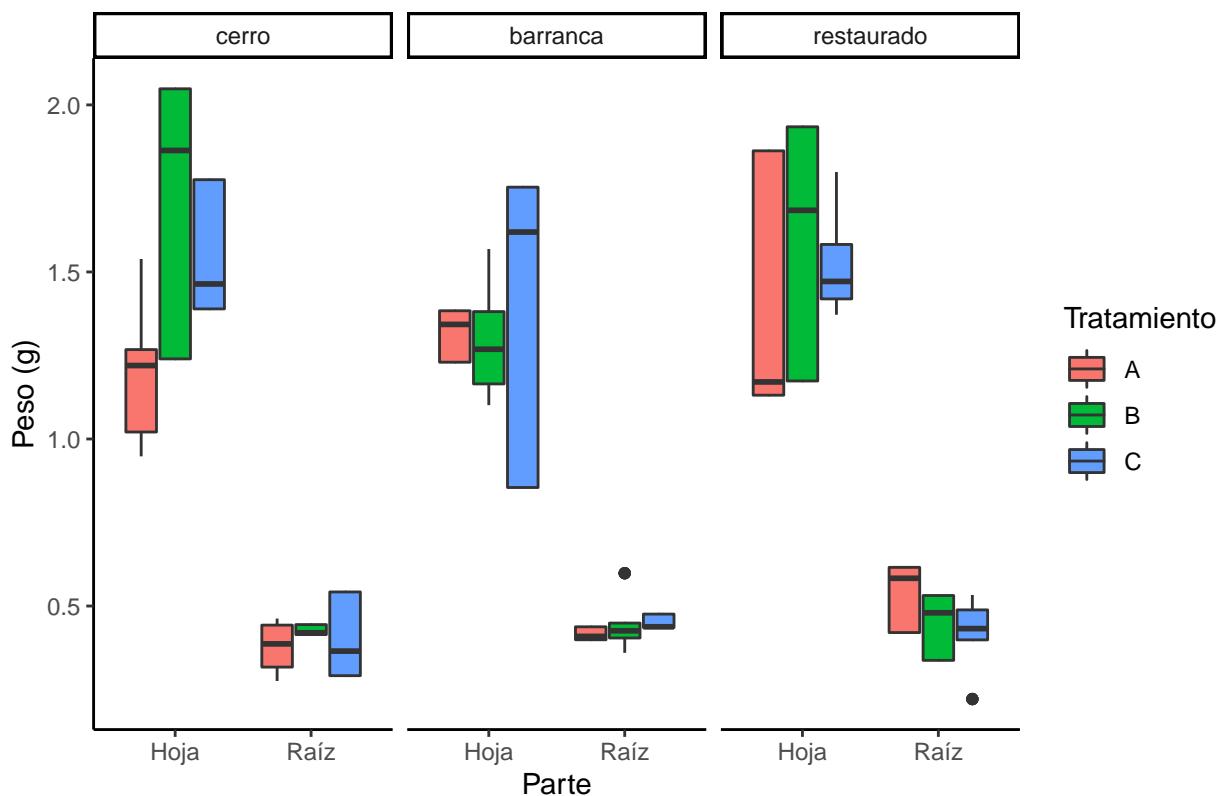
```
ggplot(data = peso_seco, aes(x = tratamiento, y = pesos, fill = parte)) +
  geom_boxplot() +
  facet_wrap(~ sitio)
```



La gráfica terminada:

```
ggplot(data = peso_seco, aes(x = parte, y = pesos, fill = tratamiento)) +
  geom_boxplot() +
  facet_wrap(~ sitio) +
  labs(title = "Peso seco de raíz y hoja en cada sitio",
       x = "Parte",
       y = "Peso (g)",
       fill = "Tratamiento") +
  theme_classic() +
  scale_x_discrete(labels = c("Hoja", "Raíz"))
```

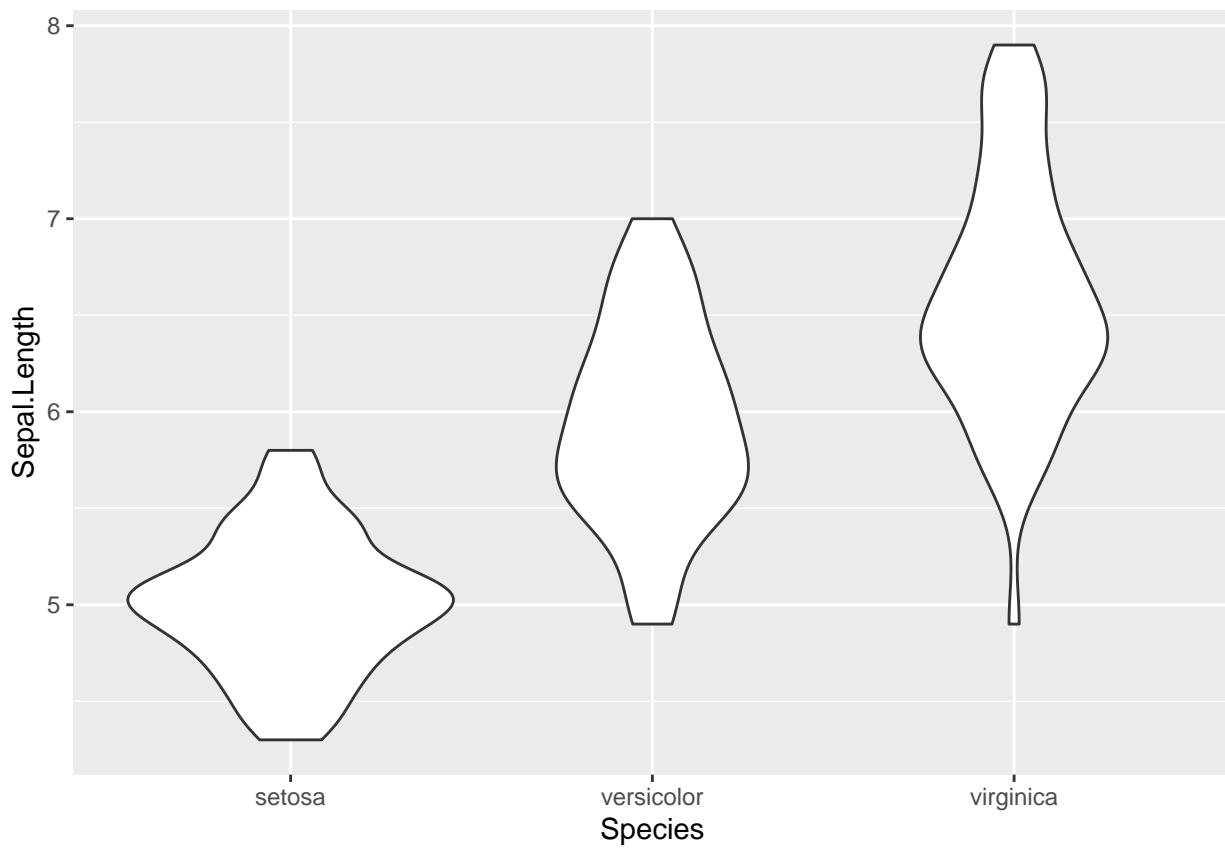
Peso seco de raíz y hoja en cada sitio



5.9.1 Gráfica de violín

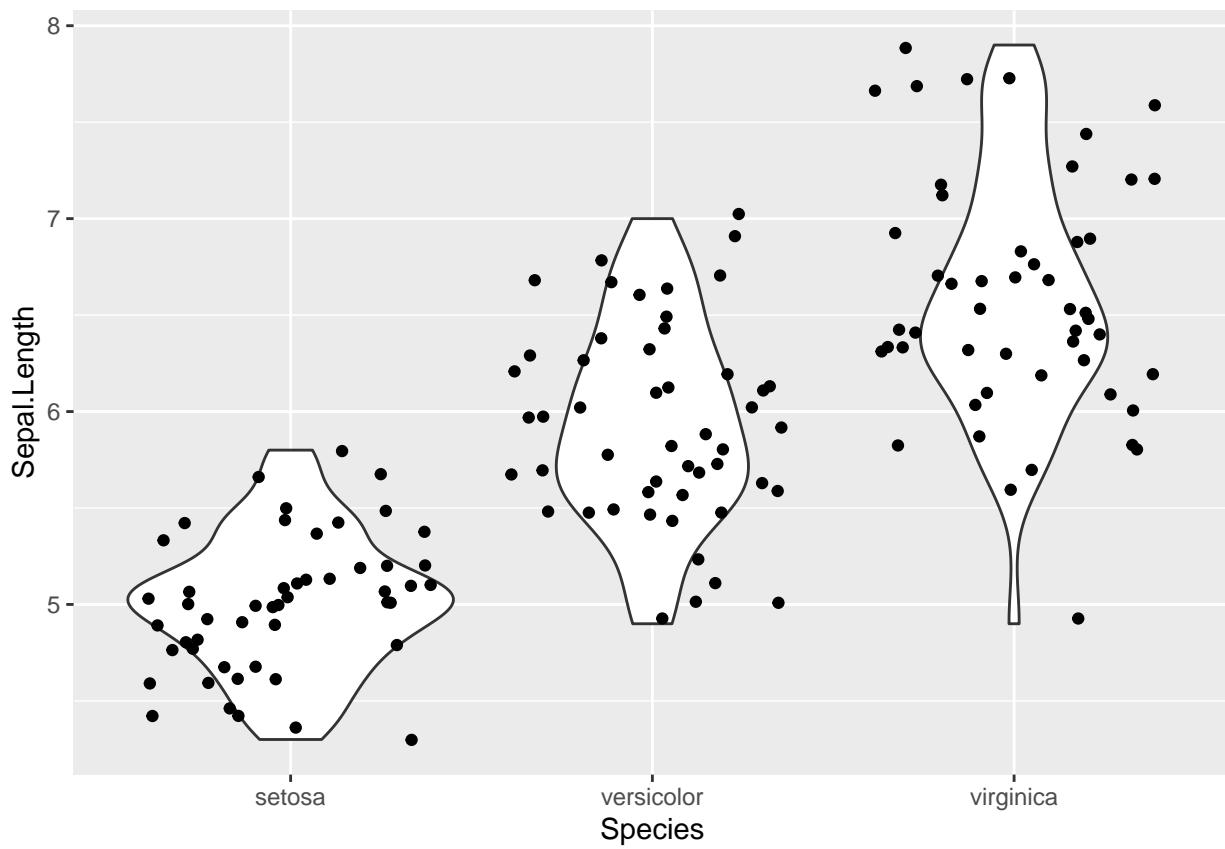
Las gráficas de cajas y bigotes permiten observar el rango, pero no dan una idea precisa acerca de la distribución de los datos a lo largo del rango. Esto se puede solucionar usando `geom_violin()`, la cual está organizada como una gráfica de cajas y bigotes:

```
ggplot(data = iris, aes(x = Species, y = Sepal.Length)) +
  geom_violin()
```



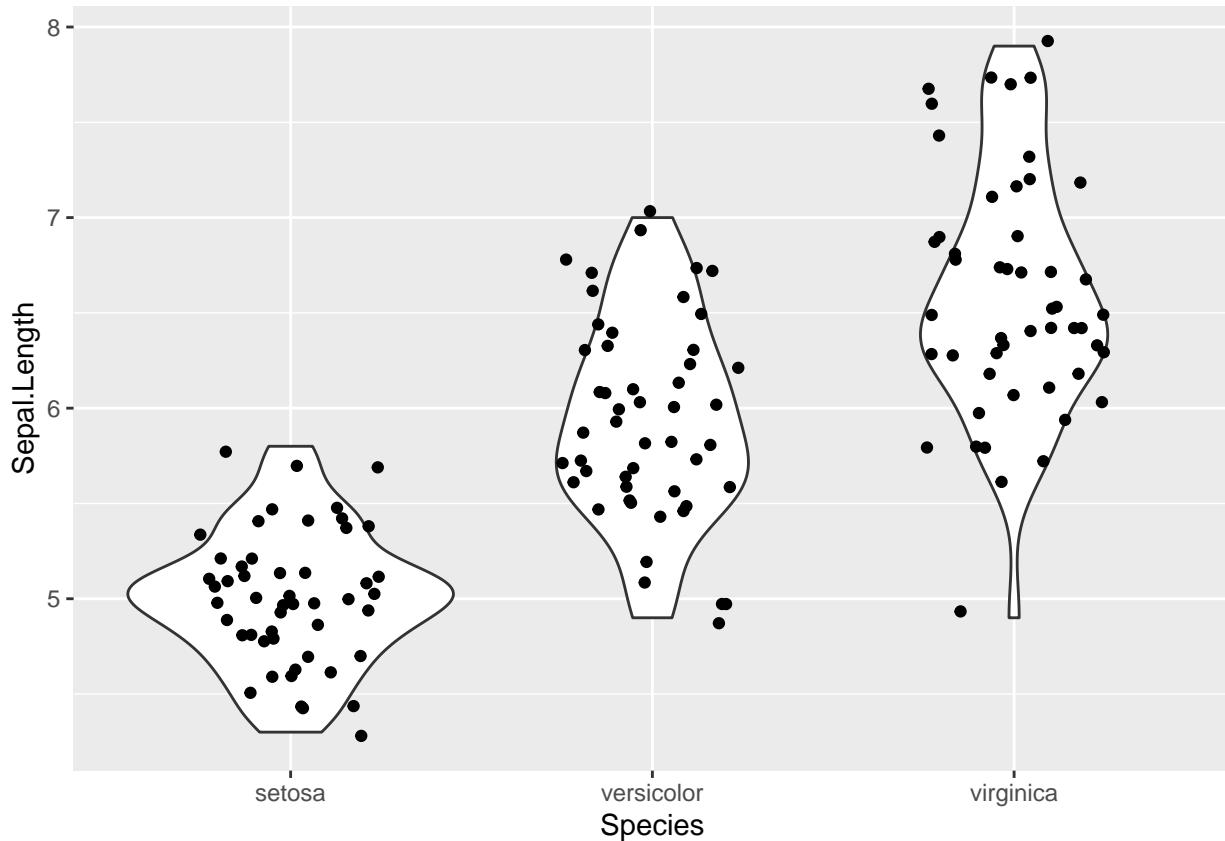
En este caso, las porciones más anchas indican el sitio o sitios en donde se acumula la mayor cantidad de datos dentro del rango de cada categoría. Es posible mostrar cada una de las observaciones usando `geom_jitter()`:

```
ggplot(data = iris, aes(x = Species, y = Sepal.Length)) +  
  geom_violin() +  
  geom_jitter()
```



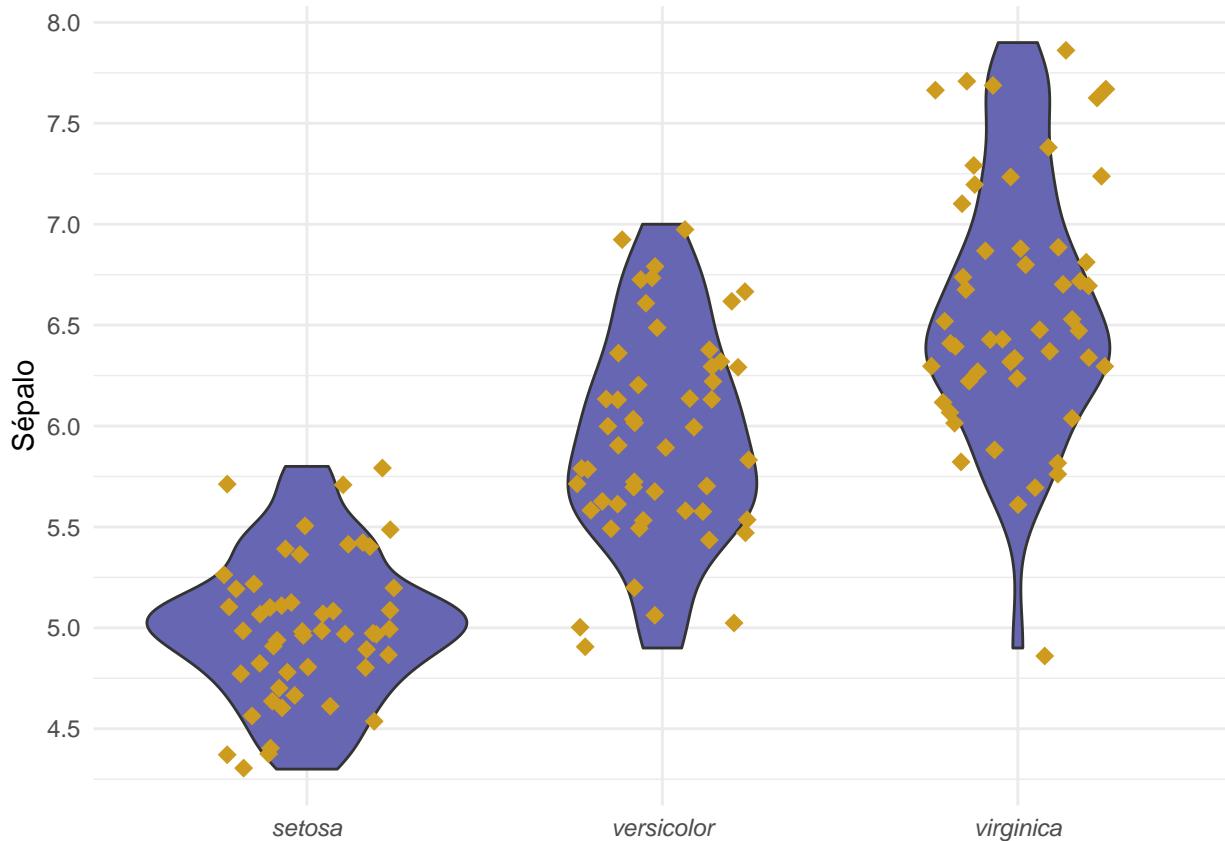
`geom_jitter()` es similar a `geom_point()`, pero `geom_jitter()` desplaza cada observación horizontalmente en el mismo valor de y dentro de la categoría correspondiente para evitar la superposición de los datos. El ancho del espacio horizontal que ocupan los datos de cada categoría puede cambiarse dentro de `geom_jitter()` con `width`:

```
ggplot(data = iris, aes(x = Species, y = Sepal.Length)) +
  geom_violin() +
  geom_jitter(width = 1/4)
```



Esta combinación es un poco redundante, y es más sencillo observar la moda a través de `geom_violin()`, pero `geom_jitter()` da una idea general de la cantidad de observaciones en cada categoría.

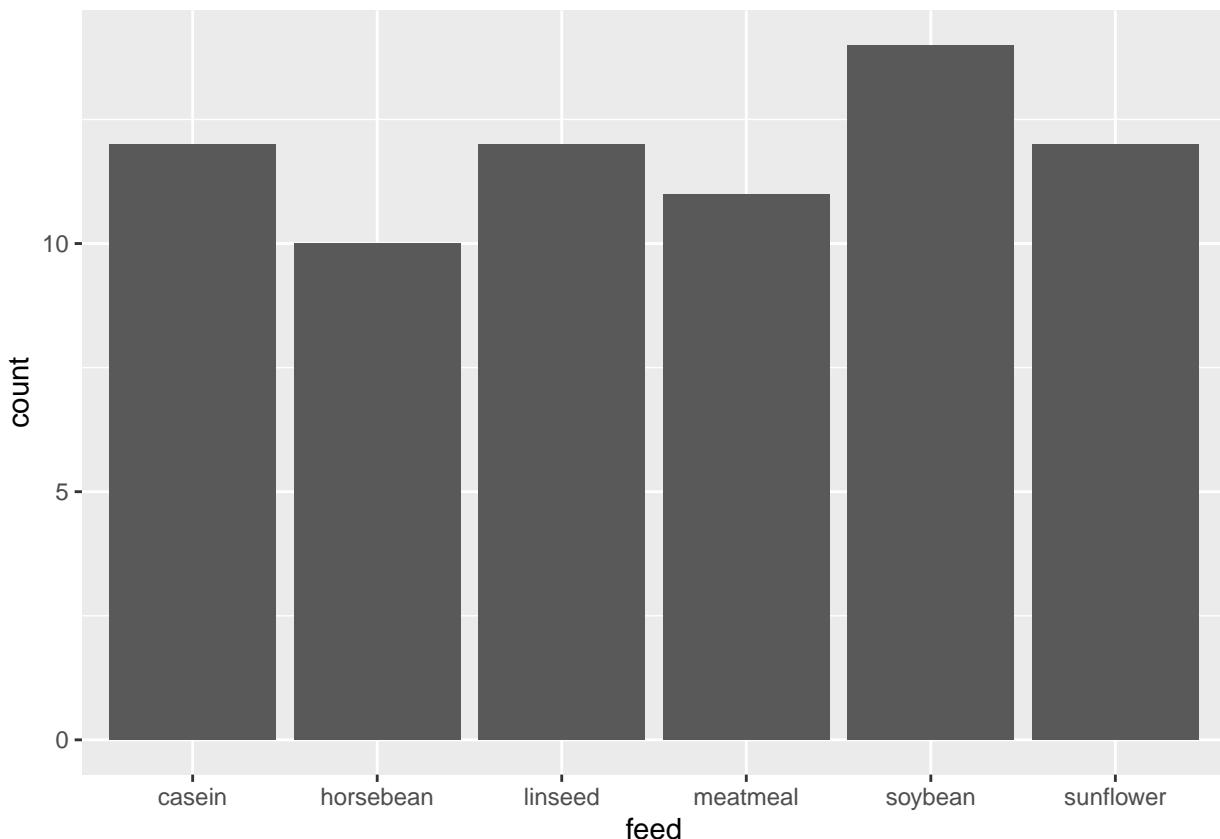
```
ggplot(data = iris, aes(x = Species, y = Sepal.Length)) +
  geom_violin(fill = rgb(red = .4,
                        green = .4,
                        blue = .7)) +
  geom_jitter(width = 1/4, pch = 18, size = 3, color = "goldenrod3") +
  labs(x = NULL,
       y = "Sépalo") +
  scale_y_continuous(breaks = seq(from = 0, to = 8, by = 0.5)) +
  theme_minimal() +
  theme(axis.text.x = element_text(face = "italic"))
```



5.10 Gráficas de barras

Se realizan con `geom_bar()`, pero su uso puede resultar en errores si no se conoce cómo funciona, particularmente el argumento `stat` (transformación estadística), que por defecto es "count", y especifica que la función debe realizar un conteo del número de observaciones para las categorías del factor especificado en `x`. Por ejemplo, en el data.frame `chickwts` se pudo contar el número de observaciones para cada tipo de alimentación:

```
ggplot(data = chickwts, aes(x = feed)) +
  geom_bar()
```



Debido a que el eje y corresponde a un conteo, no se especifica ninguna variable en `aes()`, el valor del eje será creado por `geom_bar()`.

Las gráficas de barras permiten representar el valor de una variable continua en función de una variable categórica, lo cual no siempre es un conteo. Por ejemplo, se pueden graficar los promedios de una variable, los cuales se colocan en el eje y . Estos valores deben estar previamente organizados en una columna de acuerdo con cada categoría. Volviendo a `chickwts`, es necesario primero calcular el promedio para cada categoría usando `group_by()` y `summarise`:

```
pesos_pollitos <- chickwts %>%
  group_by(feed) %>%
  summarise(peso_promedio = mean(weight))

pesos_pollitos

## # A tibble: 6 x 2
##   feed      peso_promedio
##   <fct>        <dbl>
## 1 casein       324.
## 2 horsebean    160.
## 3 linseed      219.
## 4 meatmeal     277.
## 5 soybean      246.
## 6 sunflower    329.
```

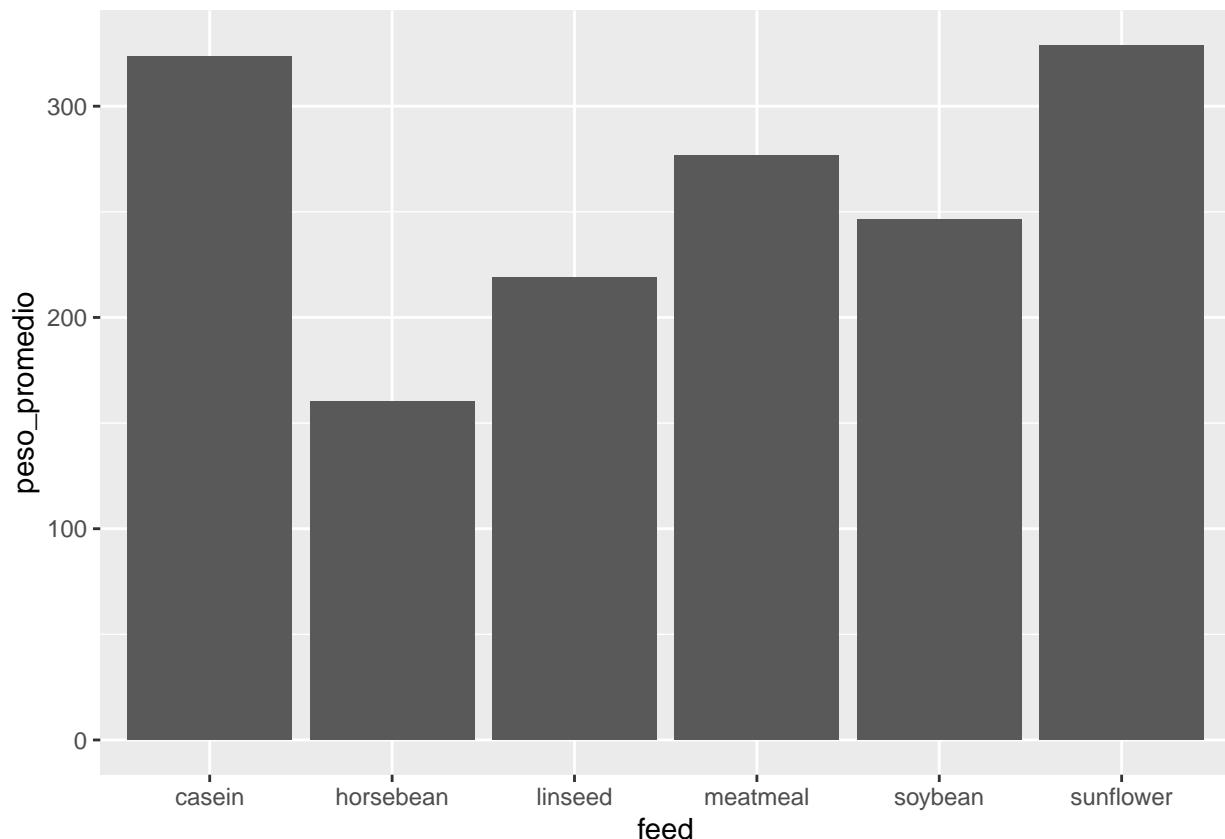
Usando este data frame no es posible usar `geom_bar()` porque marca un error relacionado con el conteo, el cual solo acepta una variable:

```
ggplot(data = pesos_pollitos, aes(x = feed, y = peso_promedio)) +
  geom_bar()
```

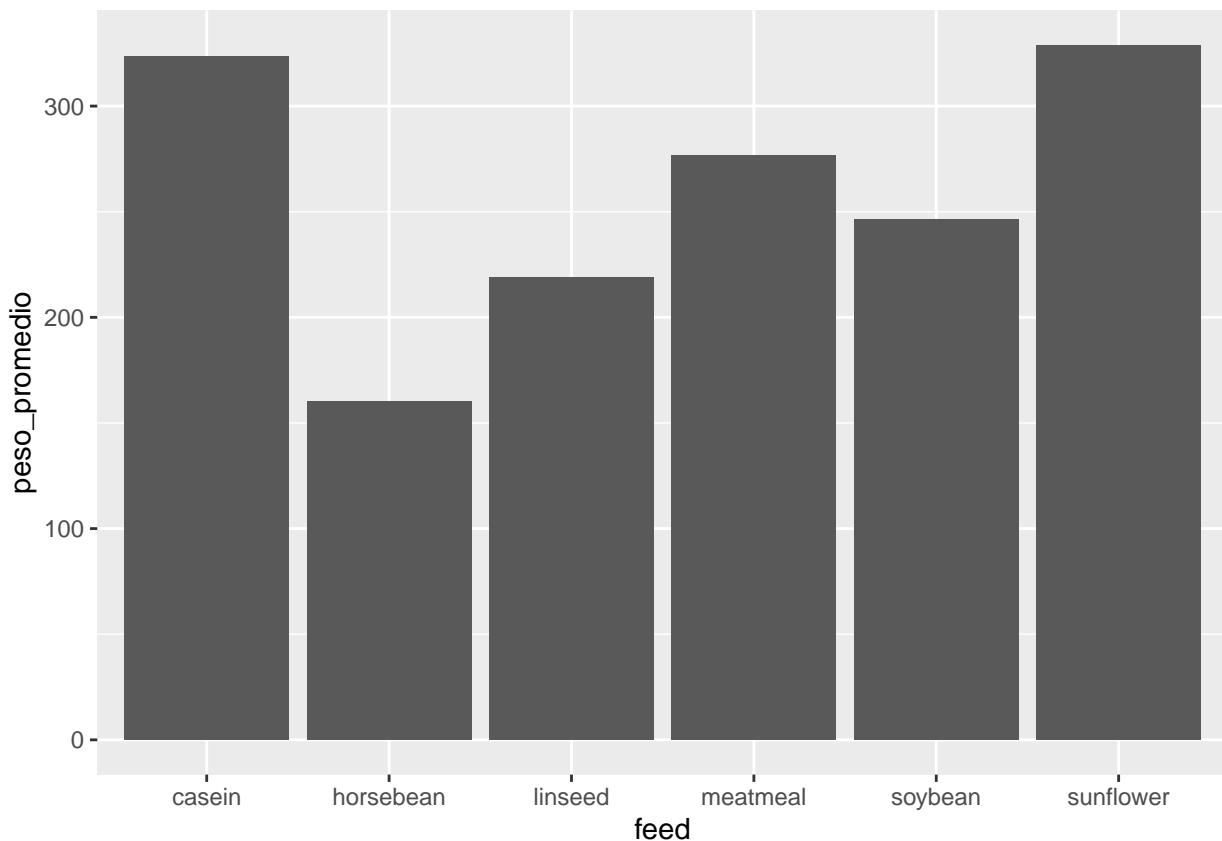
```
## Error in `f()`:  
## ! stat_count() can only have an x or y aesthetic.
```

Para graficar valores tomados de una columna de un data frame usando `geom_bar`, se debe especificar en el argumento `stat` el valor "identity" para indicar que no se debe realizar ninguna transformación a los datos (un conteo) y toma el valor de la columna indicada. Otra opción consiste en usar `geom_col()`, que utiliza por defecto `stat = "identity"`. De cualquier manera, ambos métodos tienen el mismo resultado.

```
ggplot(data = pesos_pollitos, aes(x = feed, y = peso_promedio)) +  
  geom_bar(stat = "identity")
```

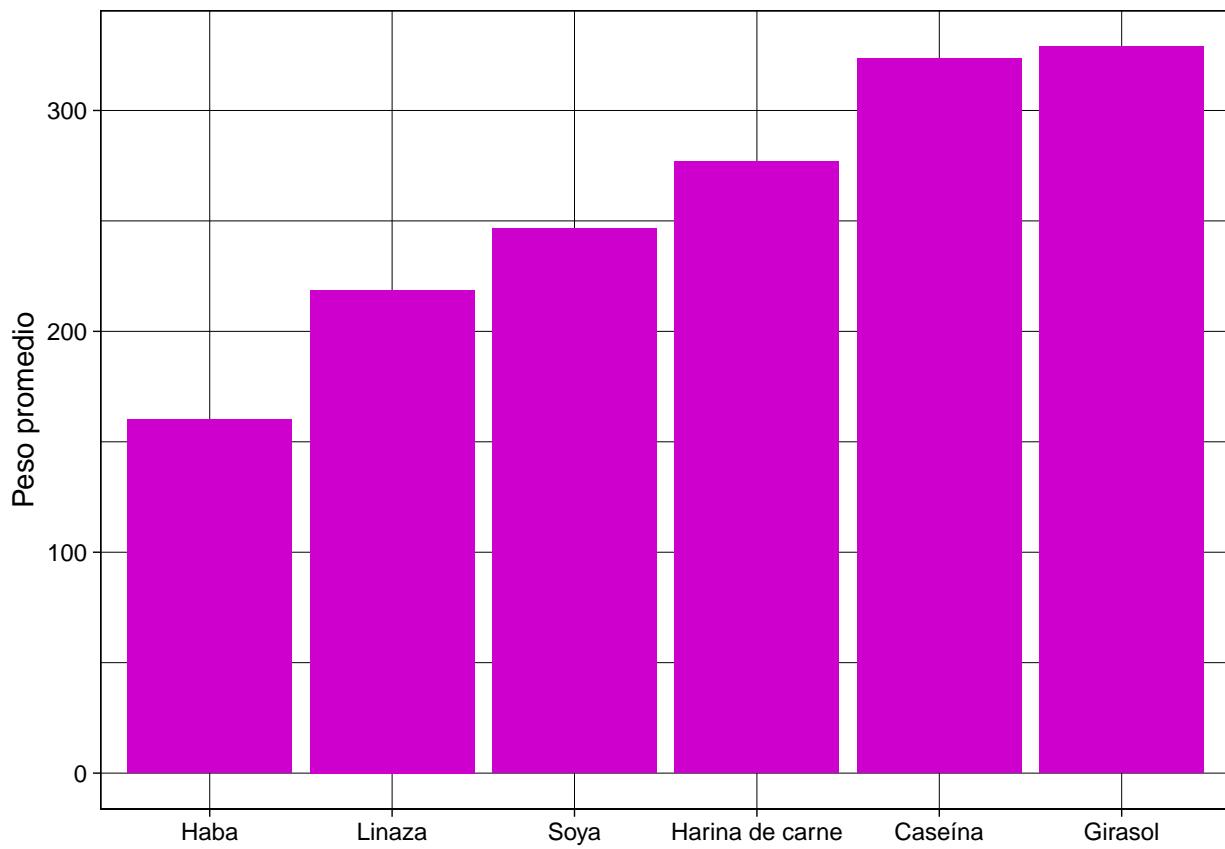


```
ggplot(data = pesos_pollitos, aes(x = feed, y = peso_promedio)) +  
  geom_col()
```



La gráfica terminada:

```
ggplot(data = pesos_pollitos, aes(x = reorder(x = feed, X = peso_promedio),
                                    y = peso_promedio)) +
  geom_col(fill = "magenta3") +
  theme_linedraw() +
  labs(
    x = NULL,
    y = "Peso promedio"
  ) +
  scale_x_discrete(labels = c("Haba", "Linaza", "Soya",
                             "Harina de carne", "Caseína",
                             "Girasol"))
```



5.10.0.1 Barras de error Se añaden con `geom_errorbar()`. Antes de añadir la función a la gráfica es necesario calcular los valores de la desviación estándar, los cuales indican la longitud de la barra por encima y por debajo del promedio. La desviación estándar puede calcularse junto con el promedio usando `group_by()` y `summarise()`:

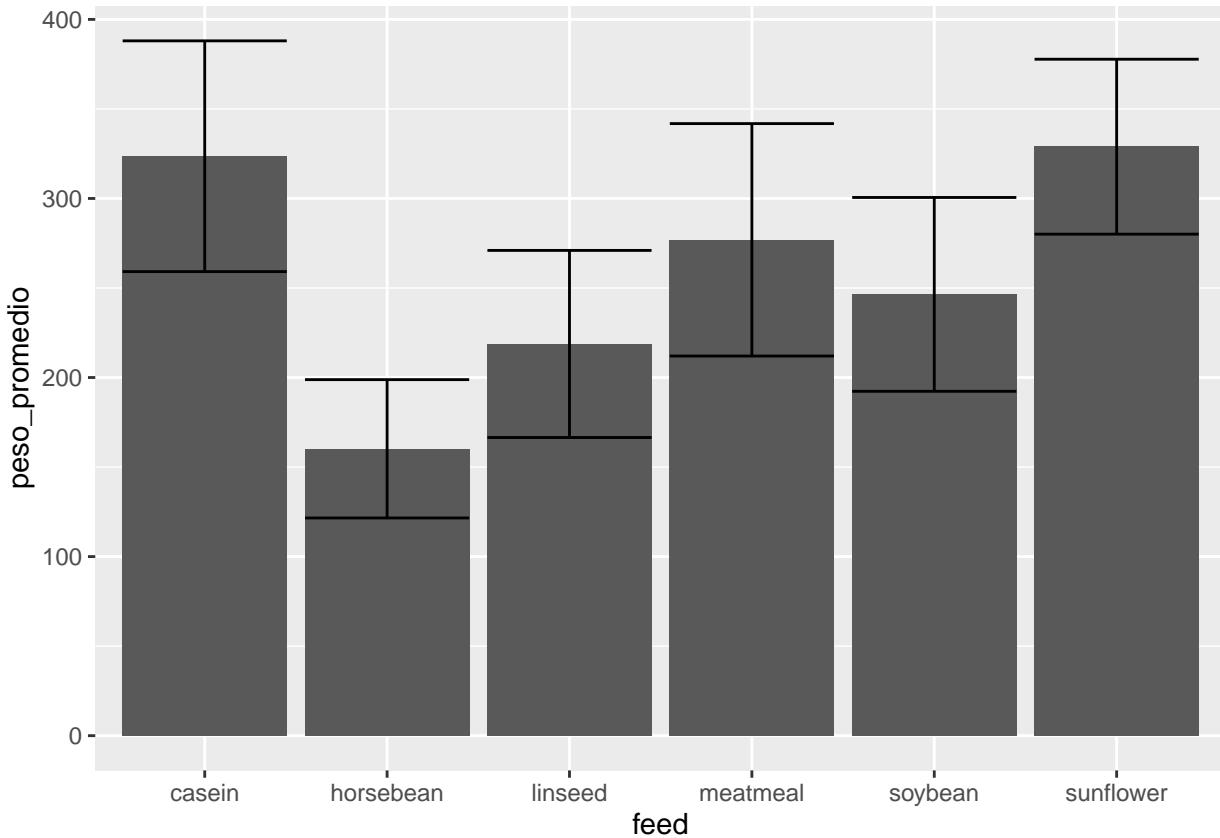
```
pesos_pollitos_error <- chickwts %>%
  group_by(feed) %>%
  summarise(peso_promedio = mean(weight),
            sd = sd(weight))

pesos_pollitos_error
## # A tibble: 6 x 3
##   feed      peso_promedio     sd
##   <fct>        <dbl> <dbl>
## 1 casein      324.   64.4
## 2 horsebean   160.   38.6
## 3 linseed     219.   52.2
## 4 meatmeal    277.   64.9
## 5 soybean     246.   54.1
## 6 sunflower   329.   48.8
```

Los valores de la columna `desv` se utilizan en `geom_errorbars()` dentro de `aes()`, en donde se deben colocar los valores mínimos y máximos para cada categoría en el eje y. Debido a que la distancia que debe cubrir la barra de error es una desviación estándar por encima y por debajo del promedio, las estéticas pueden especificarse como el promedio +- la desviación estándar:

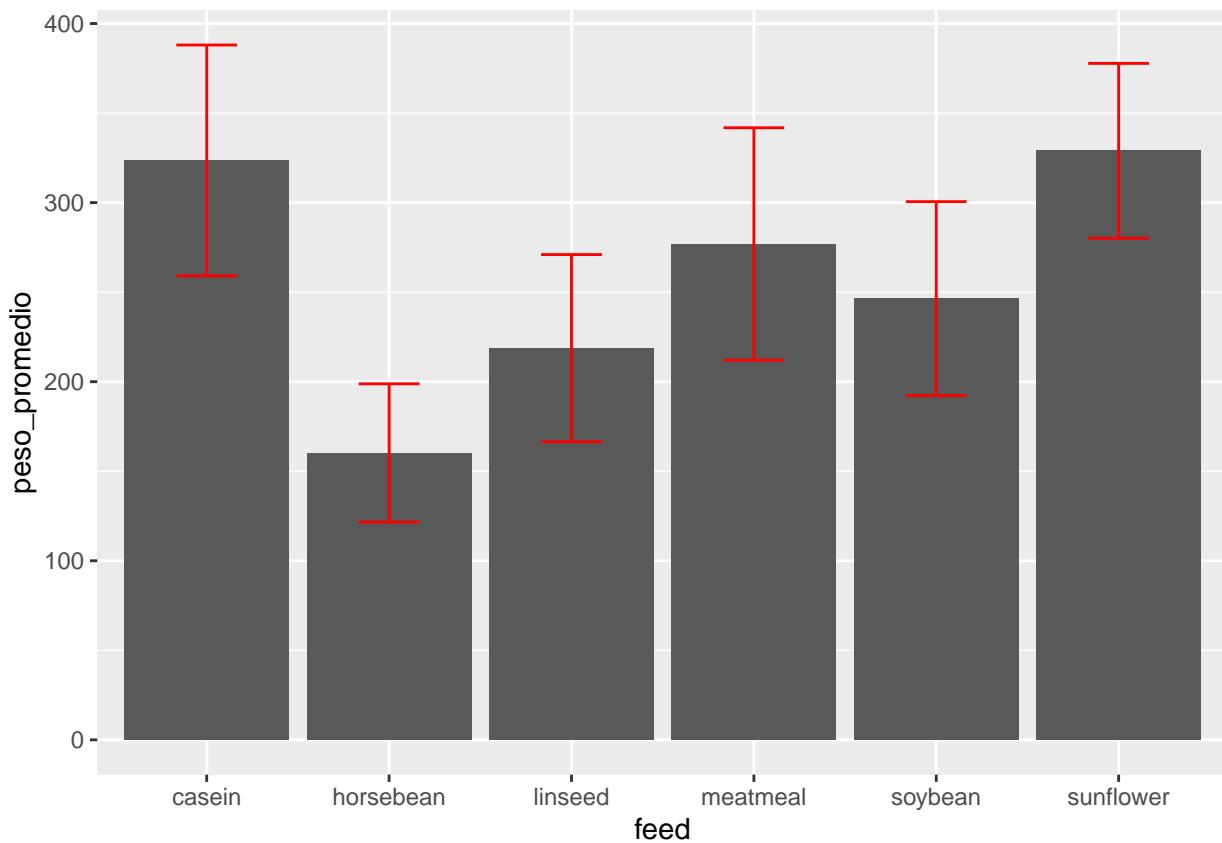
```
ggplot(data = pesos_pollitos_error, aes(x = feed, y = peso_promedio)) +
```

```
geom_col() +
geom_errorbar(aes(ymin = peso_promedio - sd,
                   ymax = peso_promedio + sd))
```



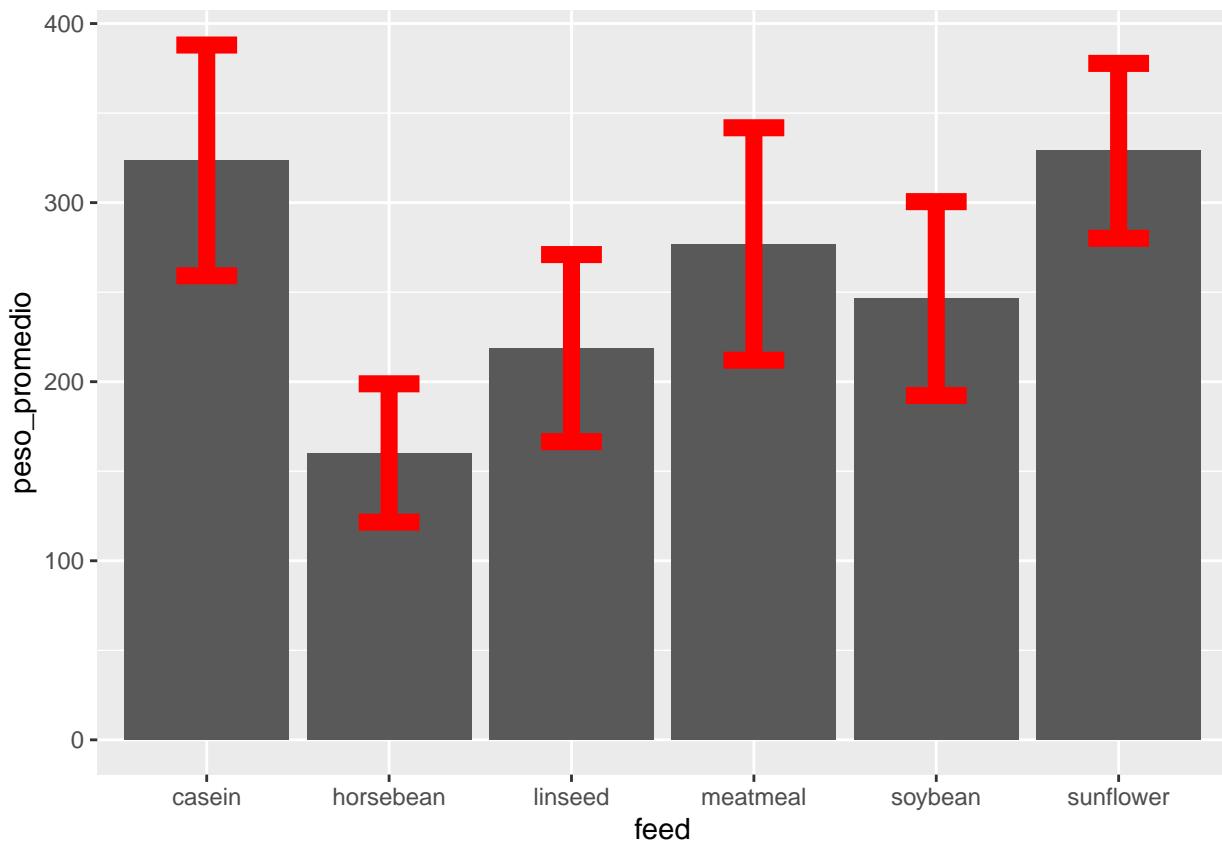
A esta gráfica inicial se le puede modificar el ancho de las barras y el color:

```
ggplot(data = pesos_pollitos_error, aes(x = feed, y = peso_promedio)) +
  geom_col() +
  geom_errorbar(aes(ymin = peso_promedio - sd,
                     ymax = peso_promedio + sd),
                width = 1/3, col = "red")
```



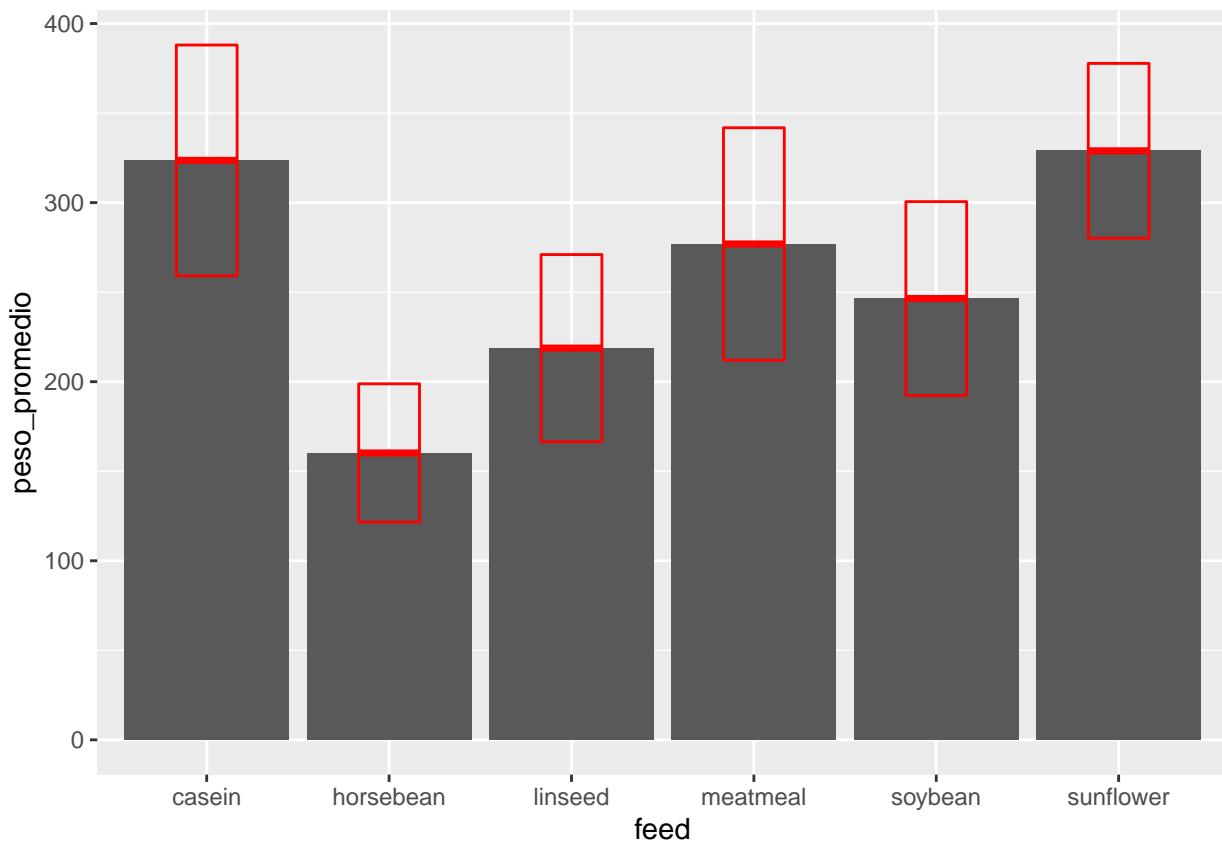
El ancho de las líneas puede ser cambiado con `size`:

```
ggplot(data = pesos_pollitos_error, aes(x = feed, y = peso_promedio)) +
  geom_col() +
  geom_errorbar(aes(ymin = peso_promedio - sd,
                     ymax = peso_promedio + sd),
                width = 1/3, col = "red", size = 3)
```

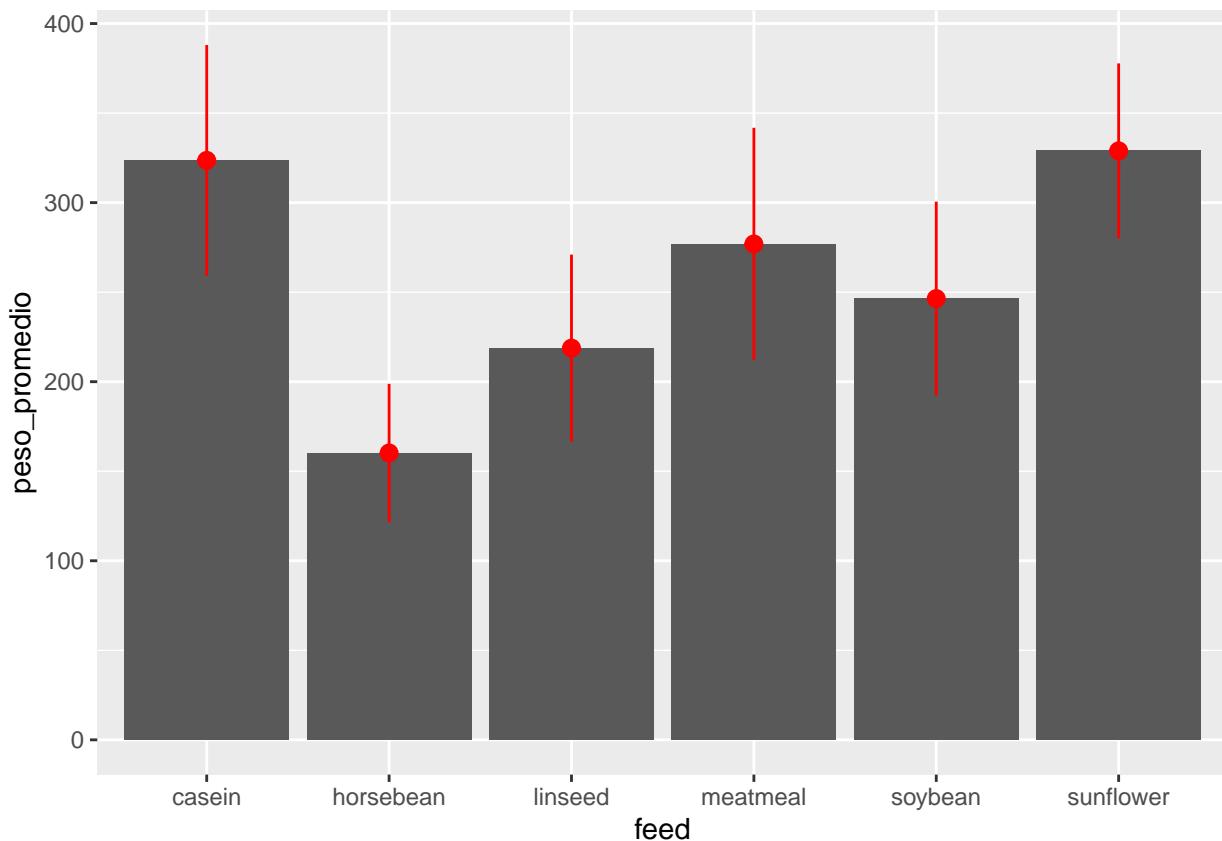


Existen otras alternativas para graficar barras de error:

```
ggplot(data = pesos_pollitos_error, aes(x = feed, y = peso_promedio)) +
  geom_col() +
  geom_crossbar(aes(ymin = peso_promedio - sd,
                     ymax = peso_promedio + sd),
                width = 1/3, col = "red")
```

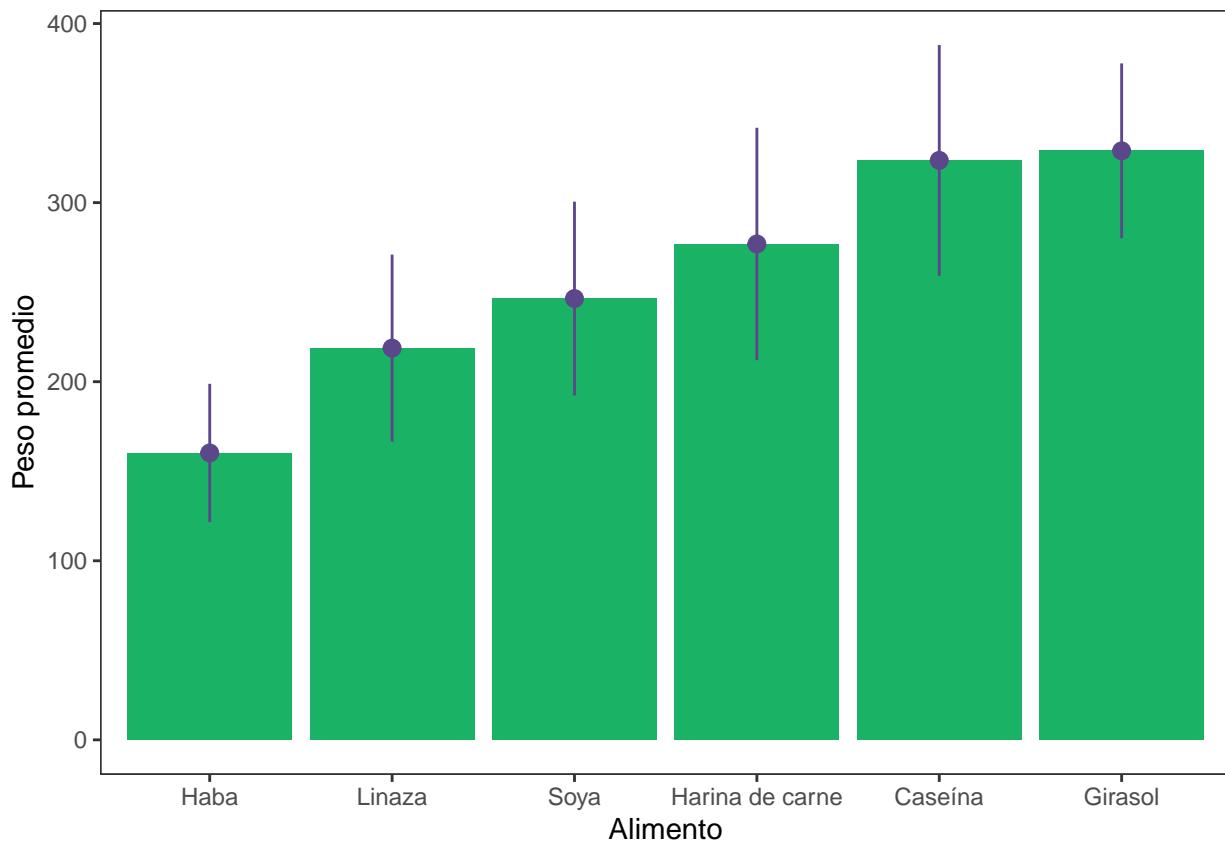


```
ggplot(data = pesos_pollitos_error, aes(x = feed, y = peso_promedio)) +  
  geom_col() +  
  geom_pointrange(aes(ymin = peso_promedio - sd,  
                      ymax = peso_promedio + sd),  
                  col = "red")
```



Una gráfica terminada puede verse así:

```
ggplot(data = pesos_pollitos_error, aes(x = reorder(x = feed, X = peso_promedio),
                                         y = peso_promedio)) +
  geom_col(fill = rgb(r = .1, g = .7, blue = .4)) +
  geom_pointrange(aes(ymin = peso_promedio - sd,
                       ymax = peso_promedio + sd),
                  col = "mediumpurple4") +
  theme_test() +
  labs(x = "Alimento",
       y = "Peso promedio",
       ) +
  scale_x_discrete(labels = c("Haba", "Linaza", "Soya",
                             "Harina de carne", "Caseína",
                             "Girasol"))
```



5.10.1 Gráficas de barras agrupadas

En las gráficas de barras es posible incorporar una variable que sea representada por el color de relleno (`fill`) de cada barra. El conjunto de datos `pesos_seriola` puede usarse para una gráfica de este tipo

```
pesos_seriola

## # A tibble: 80 x 3
##       peso especie sexo
##   <dbl> <fct>   <fct>
## 1 15.1  zonata masculino
## 2 15.4  zonata masculino
## 3 15.8  zonata masculino
## 4 15.4  zonata masculino
## 5 16.0  zonata masculino
## 6 16.5  zonata masculino
## 7 16.6  zonata masculino
## 8 14.6  zonata masculino
## 9 16.3  zonata masculino
## 10 15.7 zonata masculino
## # ... with 70 more rows
```

Primero se crea la tabla con los promedios usando `group_by()` y `summarise()`:

```
promedio_seriola <- pesos_seriola %>%
  group_by(sexo, especie) %>%
  summarize(promedio_peso = mean(peso),
            desv_peso = sd(peso))
```

```

## `summarise()` has grouped output by 'sexo'. You can override using the
## `.groups` argument.

promedio_seriola

## # A tibble: 4 x 4
## # Groups:   sexo [2]
##   sexo     especie   promedio_peso desv_peso
##   <fct>    <fct>        <dbl>      <dbl>
## 1 masculino zonata       15.4       0.718
## 2 masculino rivoliana   9.80       1.46
## 3 femenino  zonata      13.8       0.608
## 4 femenino  rivoliana   10.7       0.504

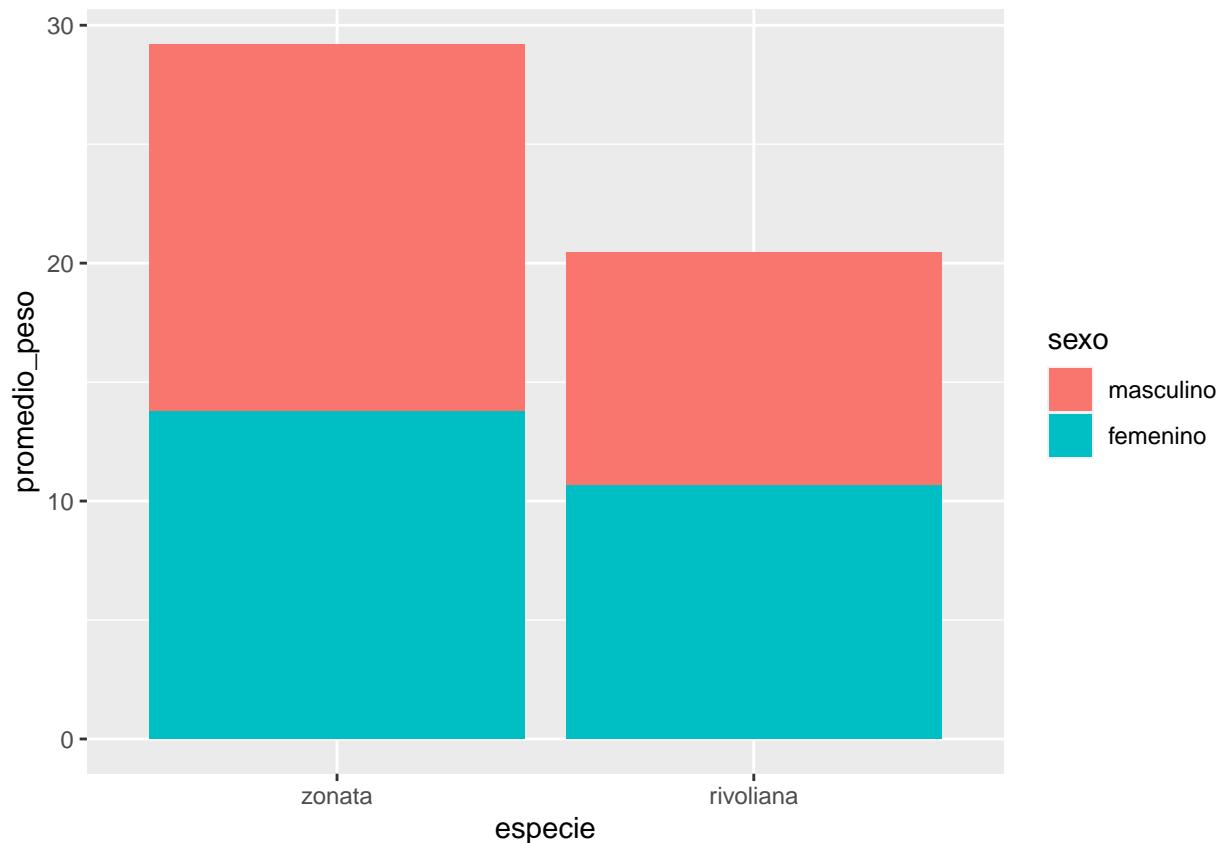
```

Para asignar una variable al color de relleno, únicamente se especifica dentro de `aes()`:

```

ggplot(data = promedio_seriola, aes(x = especie, y = promedio_peso,
                                    fill = sexo)) +
  geom_col()

```

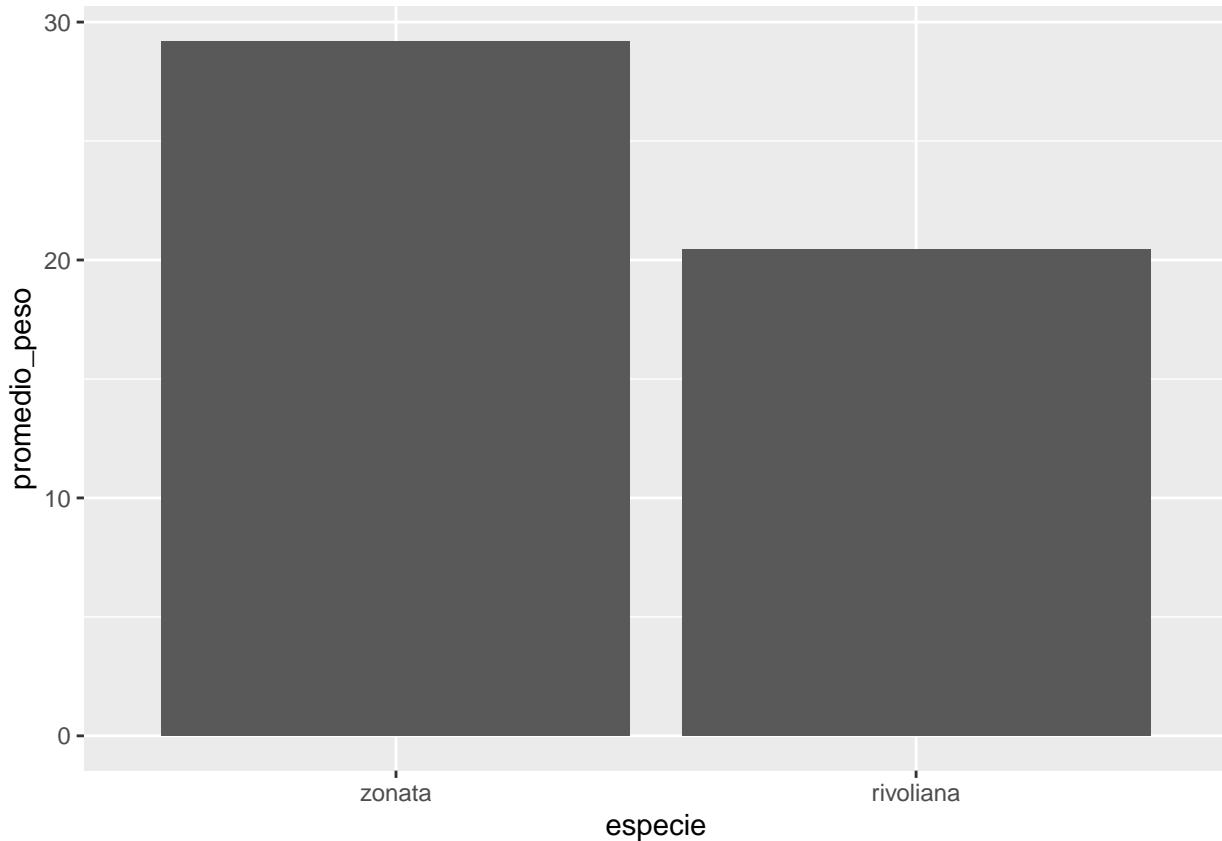


Por defecto, `sexo` solo se indica segmentando la barra correspondiente a la variable en el eje `x`, de acuerdo con la proporción que cada sexo representa del total de cada categoría en `x`. Una gráfica sin la distinción de sexo solo carece de la división por color en cada barra:

```

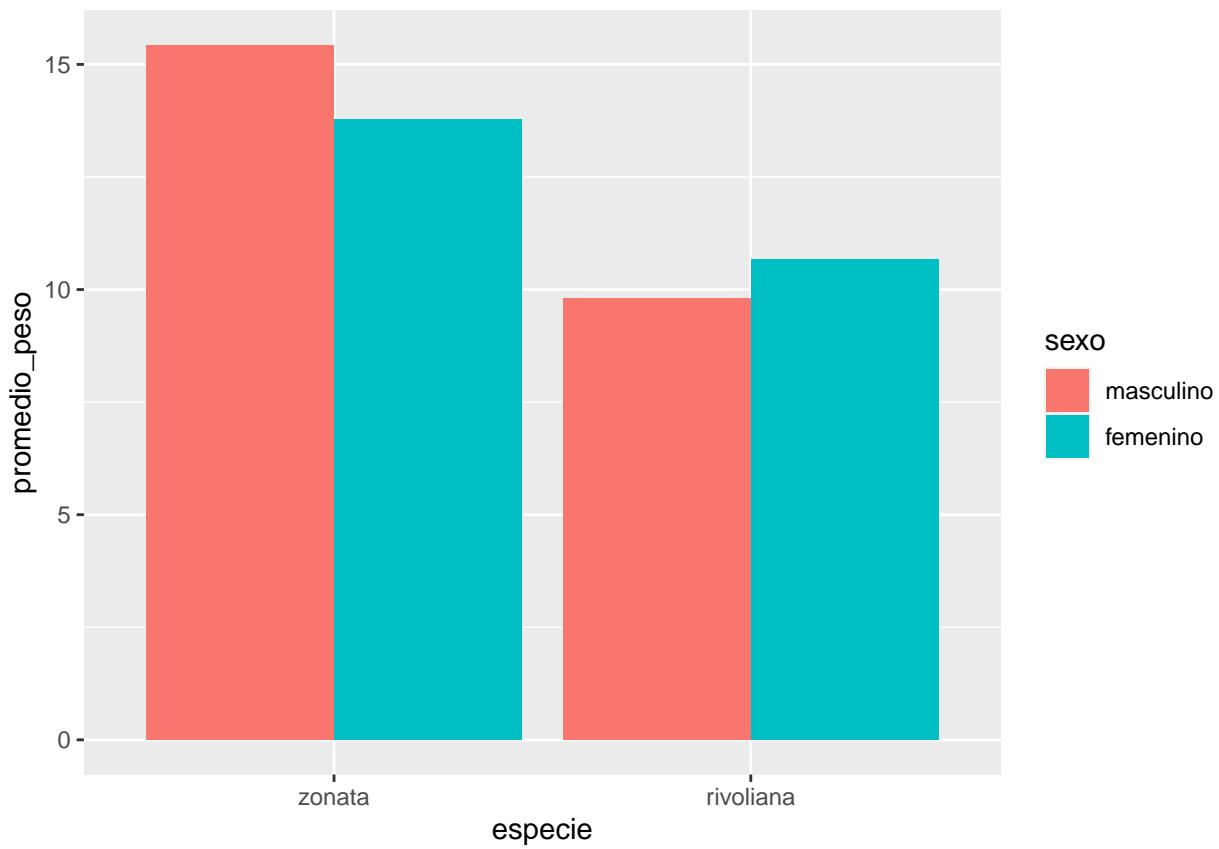
ggplot(data = promedio_seriola, aes(x = especie, y = promedio_peso)) +
  geom_col()

```



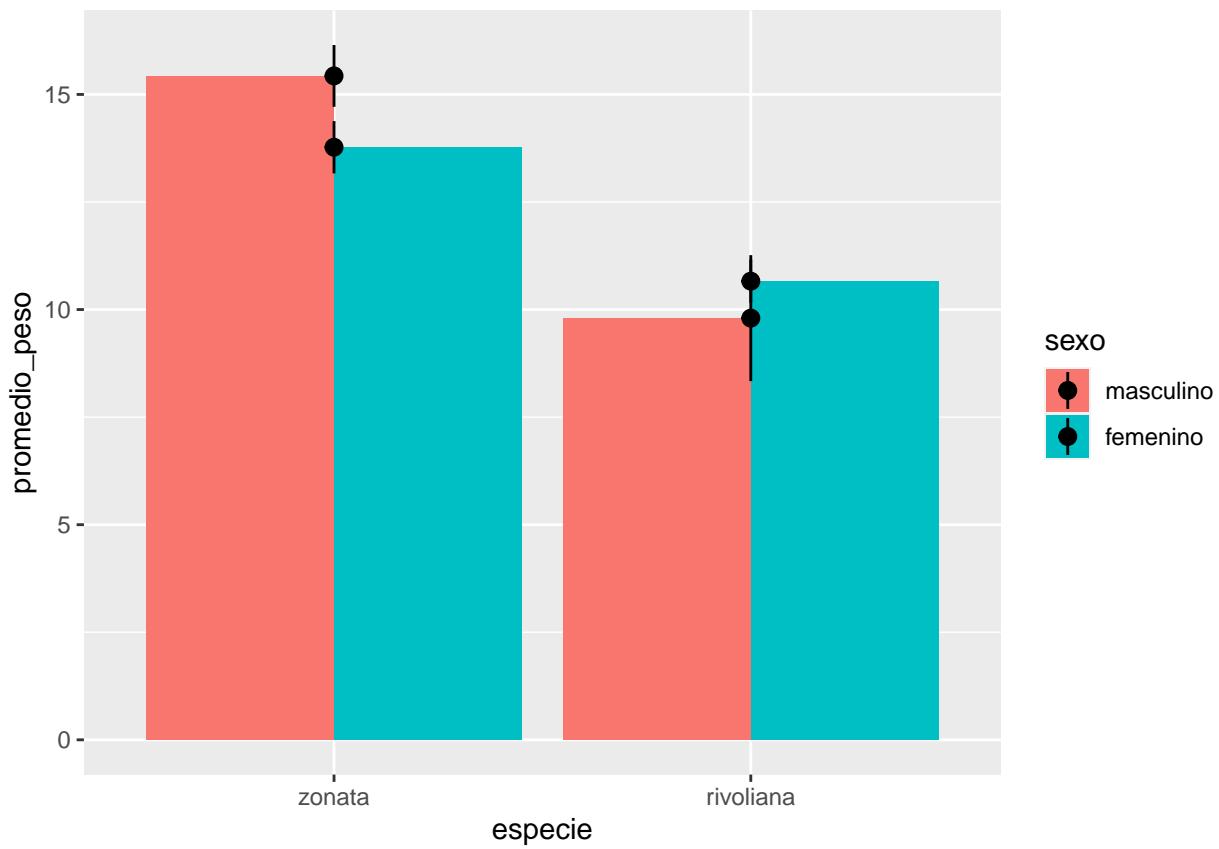
Otra forma de mostrar las diferentes categorías de `sexo` es creando barras distintas y mostrándolas agrupadas para cada especie. Esto puede hacerse modificando el argumento `position`, al cual se le debe asignar el valor "dodge":

```
ggplot(data = promedio_seriola, aes(x = especie, y = promedio_peso,
                                         fill = sexo)) +
  geom_col(position = "dodge")
```



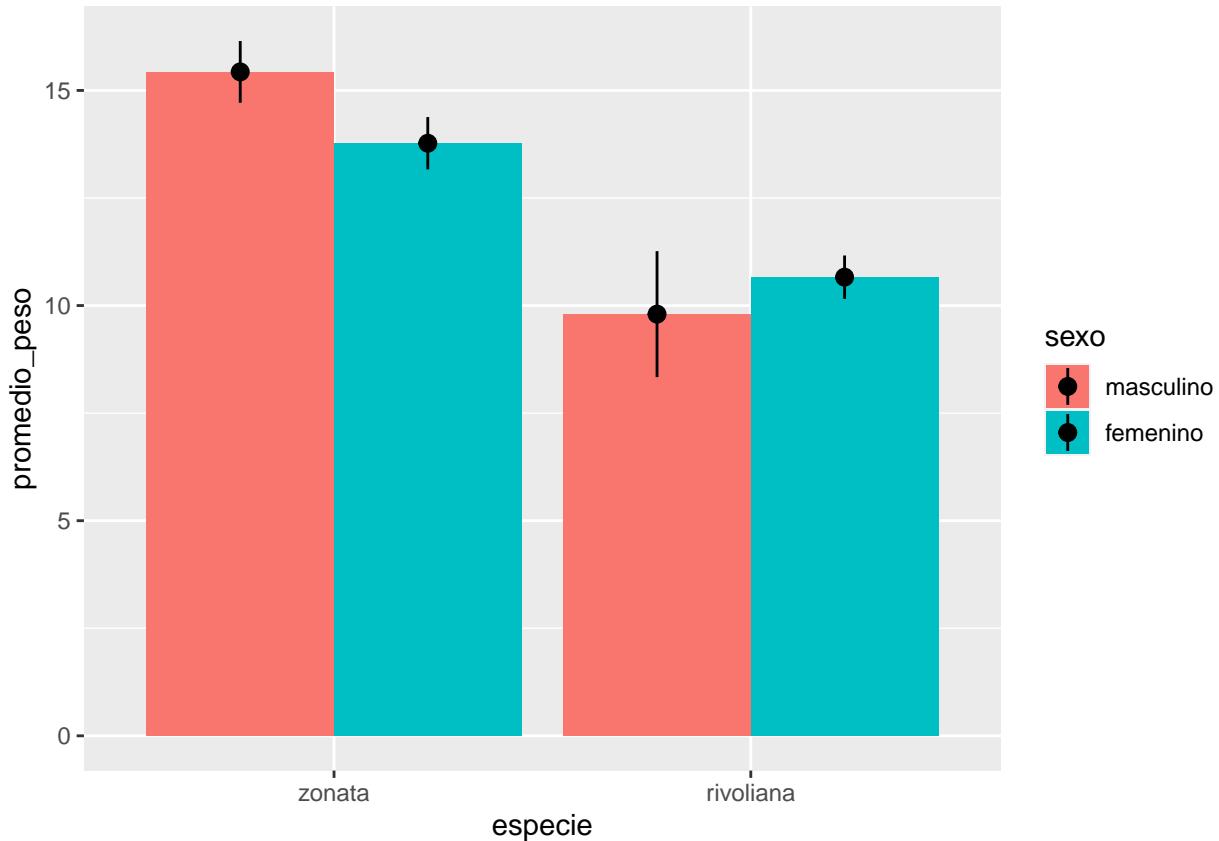
Para colocar barras de error es necesario usar un argumento que indique que existe separación entre las barras de cada categoría. Si se agrega la capa de las barras de error sin especificar que deben estar separadas, se acomodan en el centro de la categoría:

```
ggplot(data = promedio_seriola, aes(x = especie, y = promedio_peso,
                                         fill = sexo)) +
  geom_col(position = "dodge") +
  geom_pointrange(aes(ymin = promedio_peso - desv_peso,
                       ymax = promedio_peso + desv_peso))
```



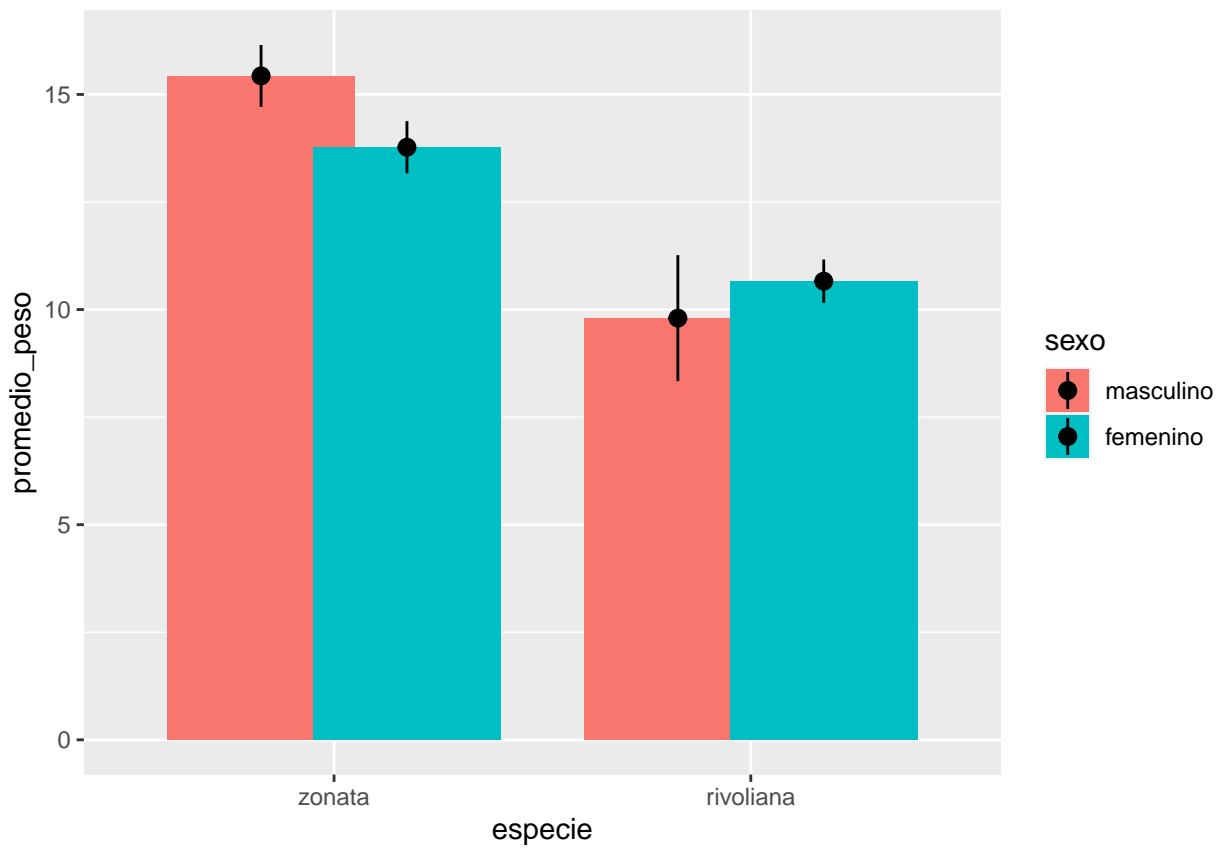
En el argumento `position` se indica el ancho de la separación entre cada barra con la función `position_dodge()` usando el argumento `width`. Esto debe colocarse tanto en `geom_col()` como en la capa de las barras de error.

```
ggplot(data = promedio_seriola, aes(x = especie, y = promedio_peso,
                                       fill = sexo)) +
  geom_col(position = position_dodge(width = .9)) +
  geom_pointrange(aes(ymin = promedio_peso - desv_peso,
                       ymax = promedio_peso + desv_peso),
                  position = position_dodge(width = .9))
```



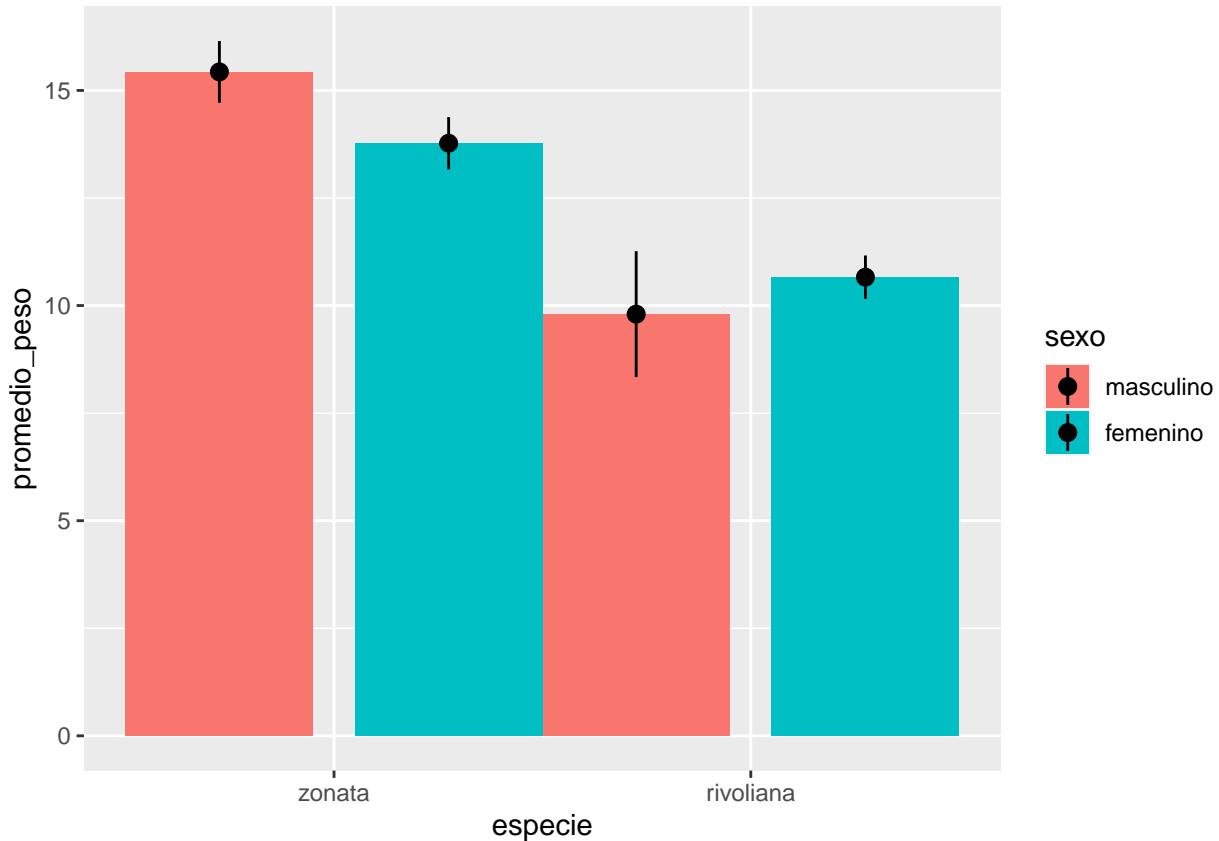
El valor de `width` no es arbitrario, .9 es necesario para la adecuada separación de las barras, cualquier valor por debajo ocasiona una superposición de las barras y por encima queda un espacio vacío:

```
ggplot(data = promedio_seriola, aes(x = especie, y = promedio_peso,
                                       fill = sexo)) +
  geom_col(position = position_dodge(width = .7)) +
  geom_pointrange(aes(ymax = promedio_peso + desv_peso,
                      ymin = promedio_peso - desv_peso),
                  position = position_dodge(width = .7))
```



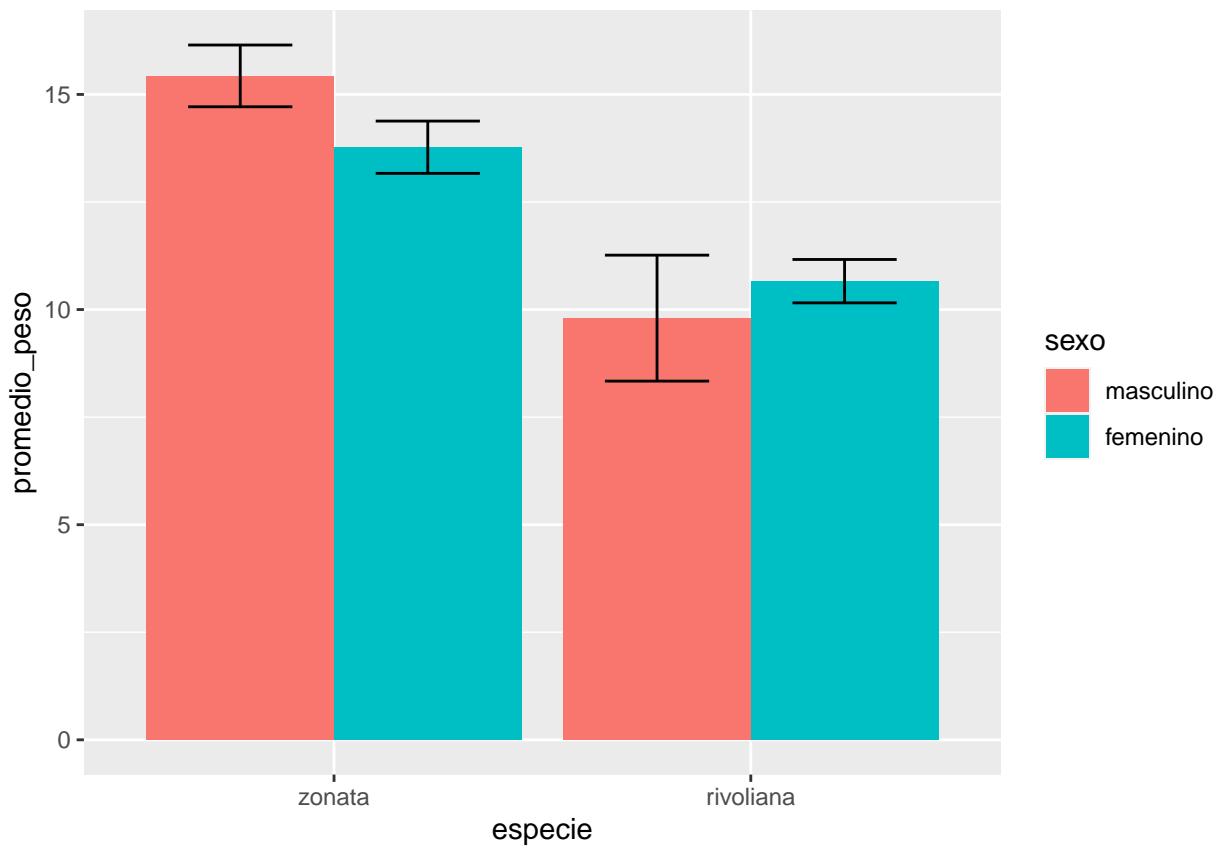
```
ggplot(data = promedio_seriola, aes(x = especie, y = promedio_peso,
                                         fill = sexo)) +
  geom_col(position = position_dodge(width = 1.1)) +
  geom_pointrange(aes(ymin = promedio_peso - desv_peso,
                       ymax = promedio_peso + desv_peso),
                  position = position_dodge(width = 1.1))

## Warning: position_dodge requires non-overlapping x intervals
```

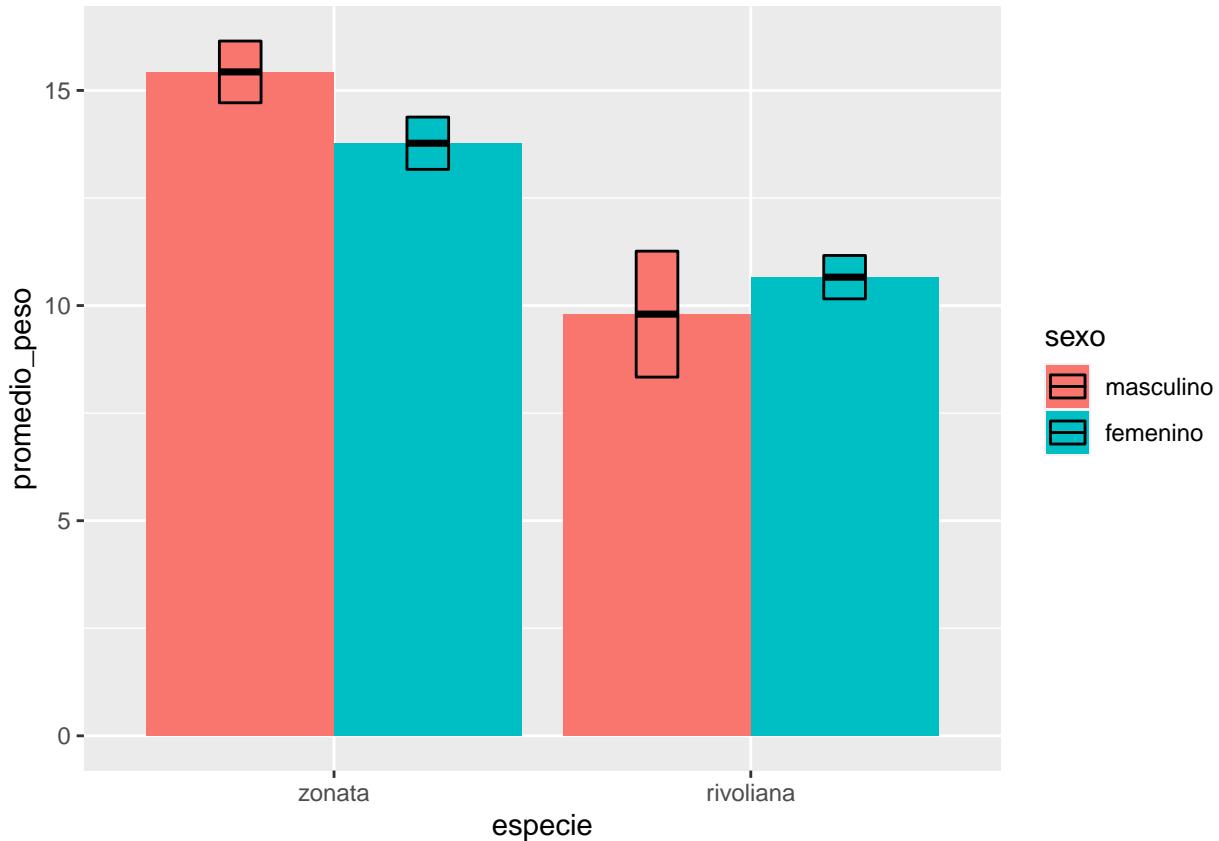


Es importante tener en cuenta que `width` puede tener diferentes implicaciones en una gráfica. Dentro de `position_dodge()` indica el ancho de separación entre cada barra, mientras que `width` como argumento de `geom_errorbar()` o `geom_crossbar()` modifica el ancho de las barras de error:

```
ggplot(data = promedio_seriola, aes(x = especie, y = promedio_peso,
                                       fill = sexo)) +
  geom_col(position = position_dodge(width = .9)) +
  geom_errorbar(aes(ymin = promedio_peso - desv_peso,
                     ymax = promedio_peso + desv_peso),
                position = position_dodge(width = .9),
                width = .5)
```

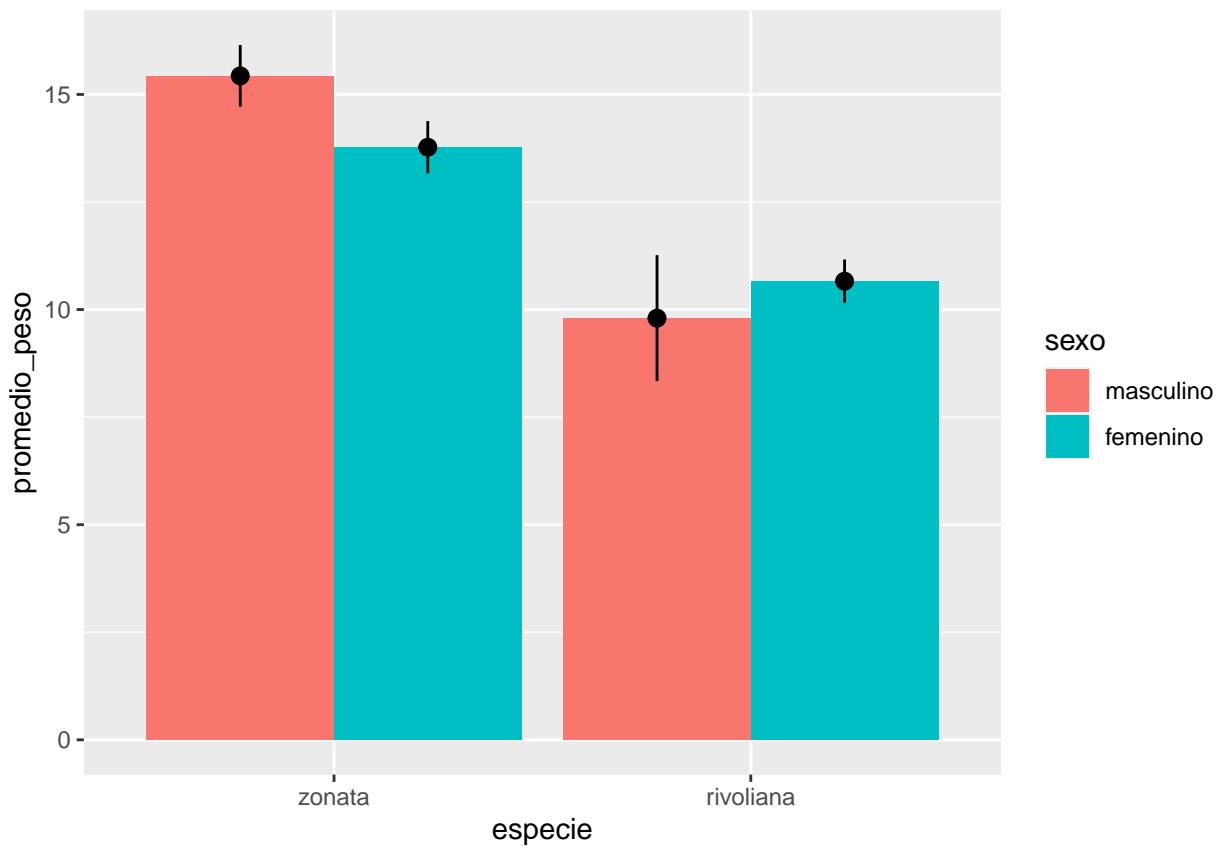


```
ggplot(data = promedio_seriola, aes(x = especie, y = promedio_peso,
                                         fill = sexo)) +
  geom_col(position = position_dodge(width = .9)) +
  geom_crossbar(aes(ymin = promedio_peso - desv_peso,
                     ymax = promedio_peso + desv_peso),
                position = position_dodge(width = .9),
                width = .2)
```

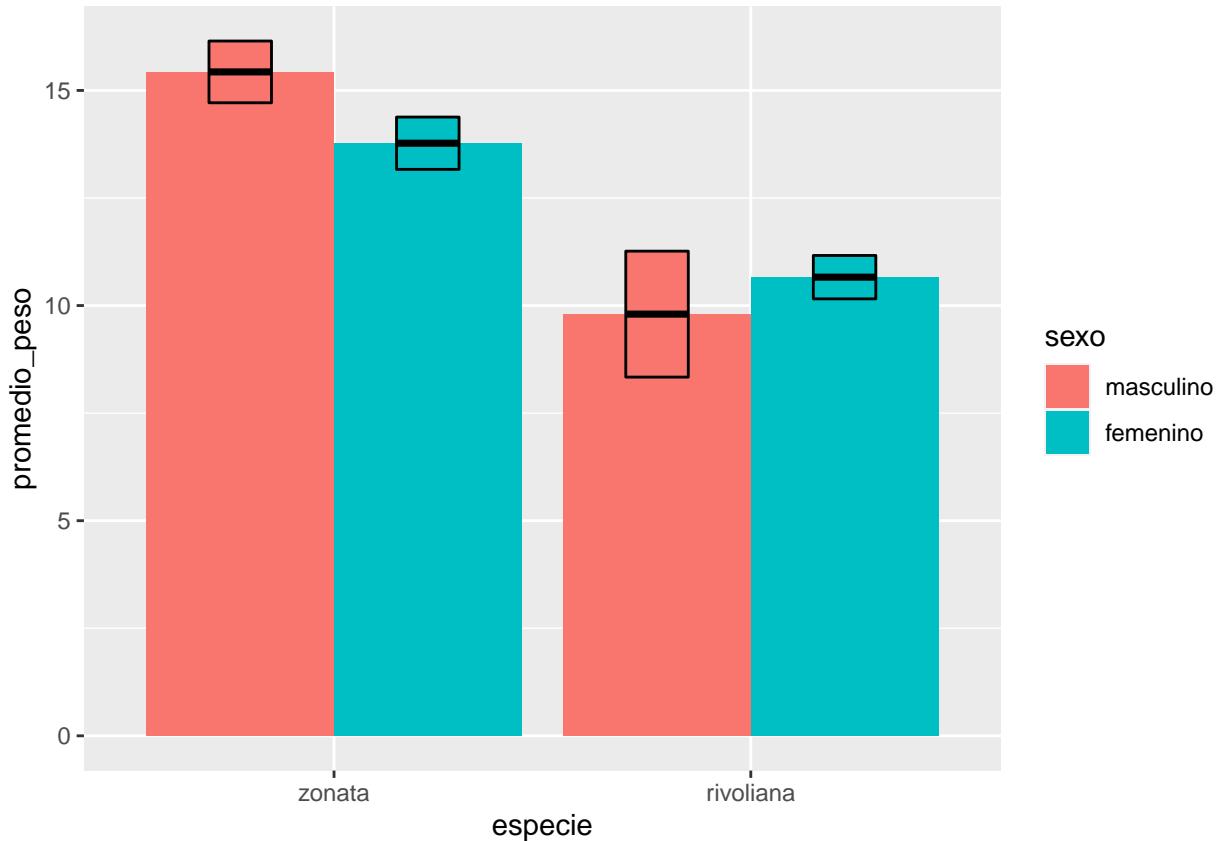


Al usar `geom_pointrange()` y `geom_crossbar()` es posible quitar la barra de error de la leyenda con el argumento `show.legend`:

```
ggplot(data = promedio_seriola, aes(x = especie, y = promedio_peso,
                                       fill = sexo)) +
  geom_col(position = position_dodge(width = .9)) +
  geom_pointrange(aes(ymax = promedio_peso + desv_peso,
                      ymin = promedio_peso - desv_peso),
                  position = position_dodge(width = .9),
                  show.legend = FALSE)
```



```
ggplot(data = promedio_seriola, aes(x = especie, y = promedio_peso,
                                         fill = sexo)) +
  geom_col(position = position_dodge(width = .9)) +
  geom_crossbar(aes(ymin = promedio_peso - desv_peso,
                     ymax = promedio_peso + desv_peso),
                position = position_dodge(width = .9),
                width = .3,
                show.legend = FALSE)
```



El uso de `position_dodge()` para indicar el ancho de separación entre las barras también funciona cuando se usa `facet_wrap()`. En el caso del conjunto de datos `peso_seco`, se pueden indicar los factores `parte` y `tratamiento` en el eje x y con `fill`, respectivamente. Para separar los datos en función del sitio se usa `facet_wrap()`, colocando únicamente el ancho de separación en `geom_col()` y en la capa de las barras de error.

Primero se construye el data frame con los promedios y desviaciones estándar:

```
promedio_peso_plantas <- peso_seco %>%
  group_by(partido, sitio, tratamiento) %>%
  summarise(promedio_peso = mean(pesos),
            desv_peso = sd(pesos))

## `summarise()` has grouped output by 'partido', 'sitio'. You can override using
## the `groups` argument.

promedio_peso_plantas

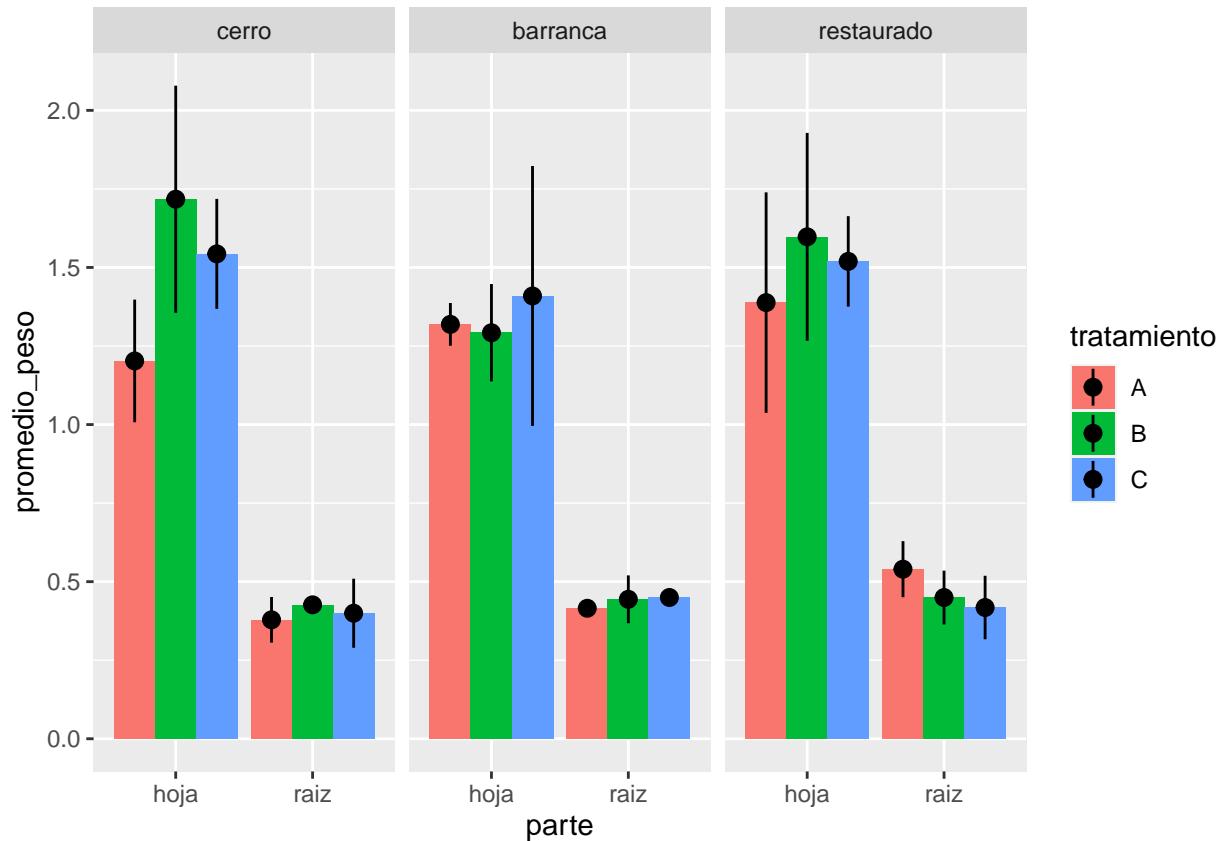
## # A tibble: 18 x 5
## # Groups:   partido, sitio [6]
##       partido sitio     tratamiento promedio_peso desv_peso
##       <fct>   <fct>      <fct>           <dbl>      <dbl>
## 1 hoja     cerro        A             1.20      0.195
## 2 hoja     cerro        B             1.72      0.361
## 3 hoja     cerro        C             1.54      0.175
## 4 hoja     barranca    A             1.32      0.0680
## 5 hoja     barranca    B             1.29      0.155
## 6 hoja     barranca    C             1.41      0.414
## 7 hoja     restaurado A             1.39      0.351
```

```

## 8 hoja restaurado B           1.60      0.331
## 9 hoja restaurado C           1.52      0.144
## 10 raiz cerro A              0.379     0.0727
## 11 raiz cerro B              0.426     0.0136
## 12 raiz cerro C              0.400     0.110
## 13 raiz barranca A           0.415     0.0172
## 14 raiz barranca B           0.444     0.0762
## 15 raiz barranca C           0.450     0.0194
## 16 raiz restaurado A          0.540     0.0892
## 17 raiz restaurado B          0.450     0.0858
## 18 raiz restaurado C          0.418     0.101

ggplot(data = promedio_peso_plantas, aes(x = parte,
                                           y = promedio_peso,
                                           fill = tratamiento)) +
  geom_col(position = position_dodge(width = .9)) +
  geom_pointrange(aes(ymin = promedio_peso - desv_peso,
                       ymax = promedio_peso + desv_peso),
                  position = position_dodge(width = .9)) +
  facet_wrap(~ sitio)

```



Esta última gráfica terminada podría verse así:

```

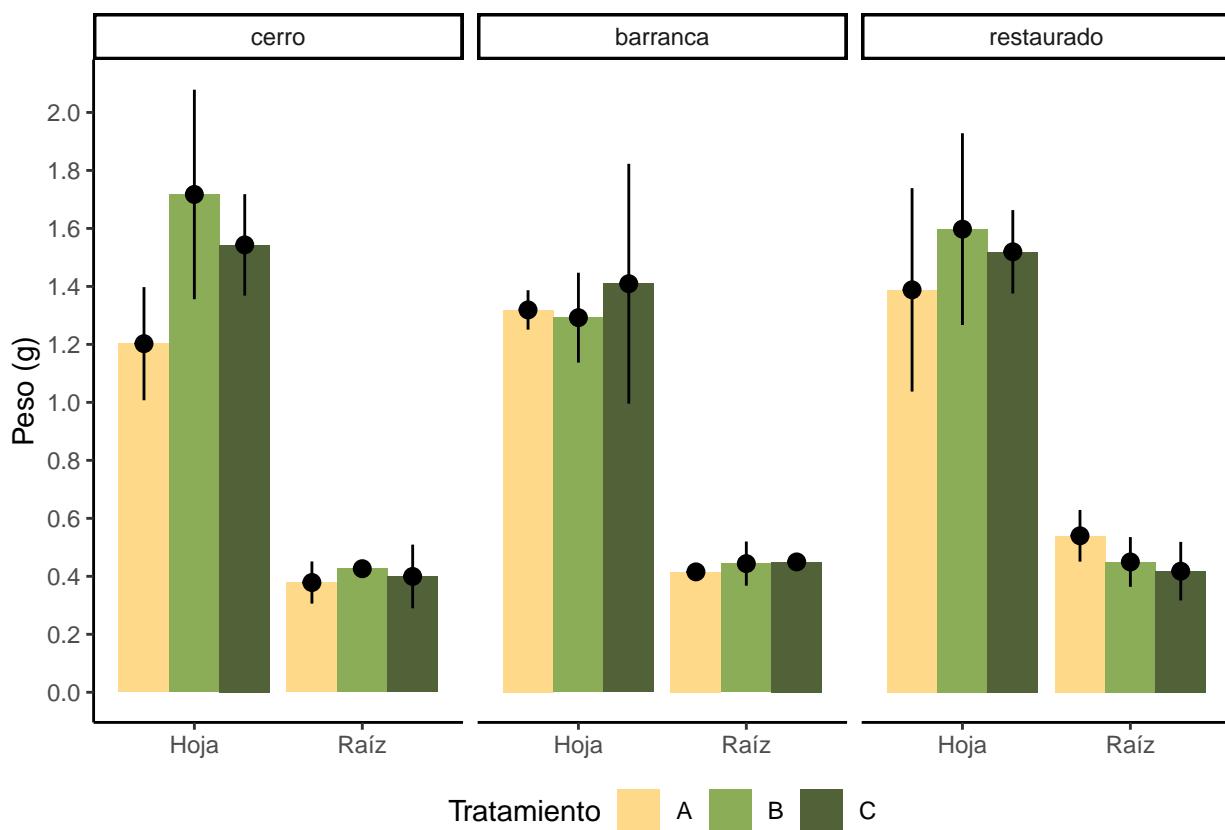
ggplot(data = promedio_peso_plantas, aes(x = parte,
                                           y = promedio_peso,
                                           fill = tratamiento)) +
  geom_col(position = position_dodge(width = .9)) +
  geom_pointrange(aes(ymin = promedio_peso - desv_peso,

```

```

    ymax = promedio_peso + desv_peso),
    position = position_dodge(width = .9),
    show.legend = FALSE) +
facet_wrap(~ sitio) +
labs(
  x = NULL,
  y = "Peso (g)",
  fill = "Tratamiento"
) +
scale_x_discrete(labels = c("Hoja", "Raíz")) +
scale_y_continuous(breaks = seq(from = 0, to = 2, by = .2)) +
theme_classic() +
theme(legend.position = "bottom",
      legend.box.spacing = unit(x = 2, units = "mm")) +
scale_fill_paletteer_d("calecopal::sierra2")

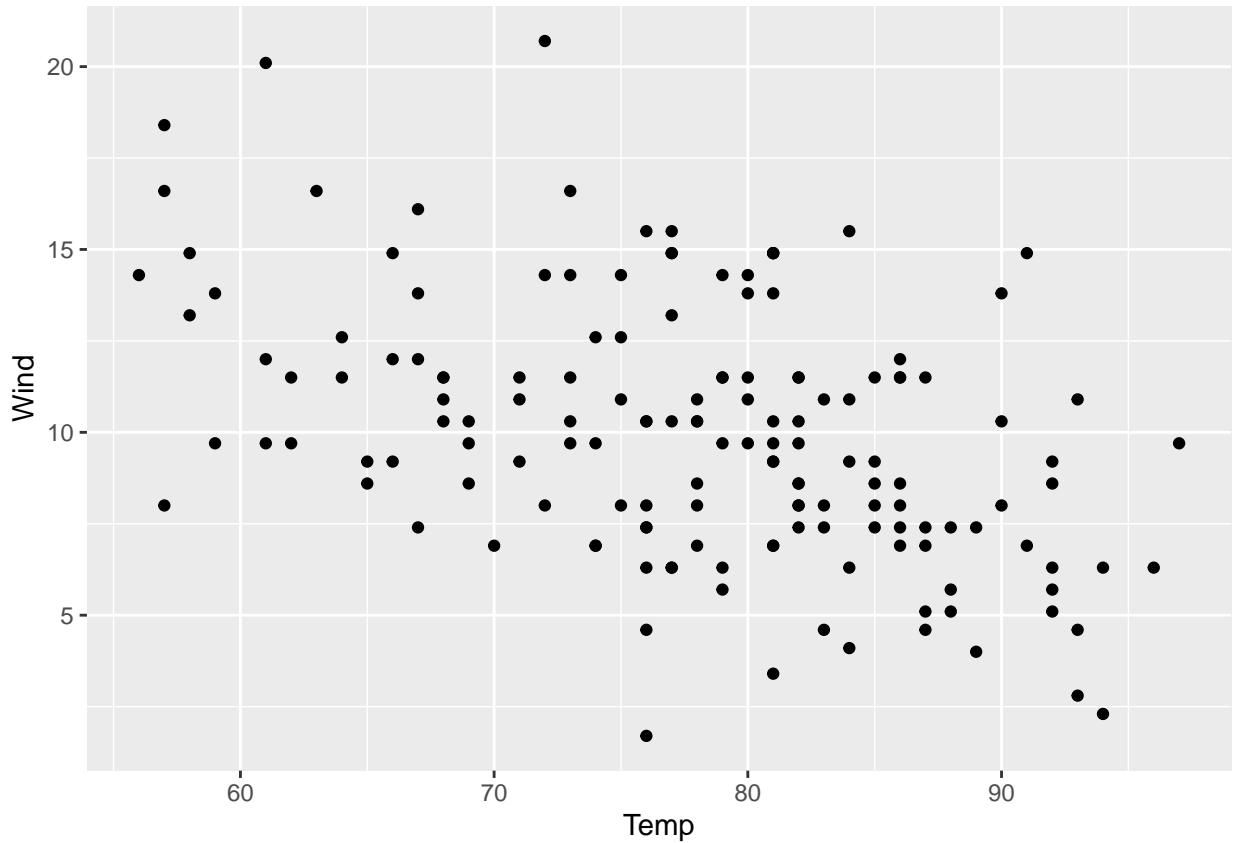
```



5.11 Gráficas de dispersión

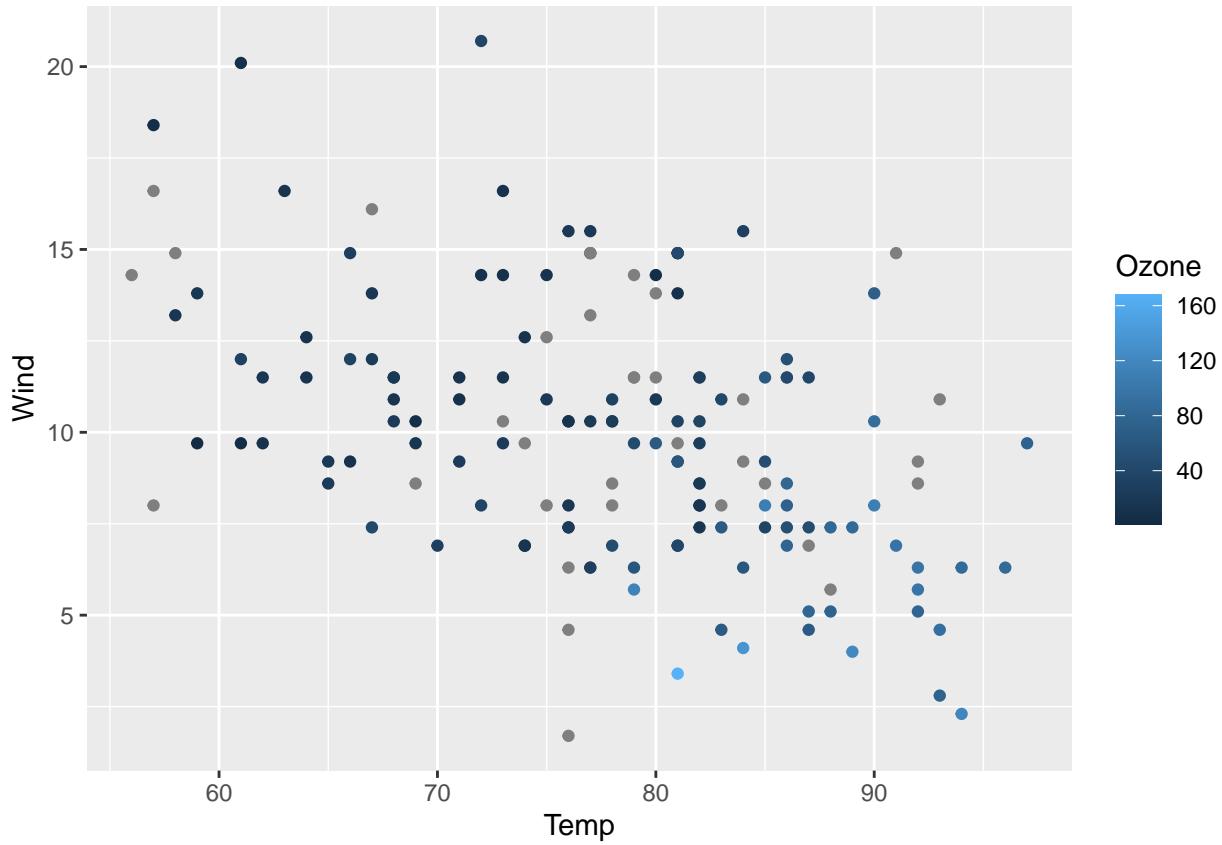
Las gráficas de dispersión permiten graficar dos variables continuas mediante `geom_point()`, por ejemplo, la velocidad del viento en función de la temperatura en el data frame `airquality`:

```
ggplot(data = airquality, aes(x = Temp, y = Wind)) +
  geom_point()
```



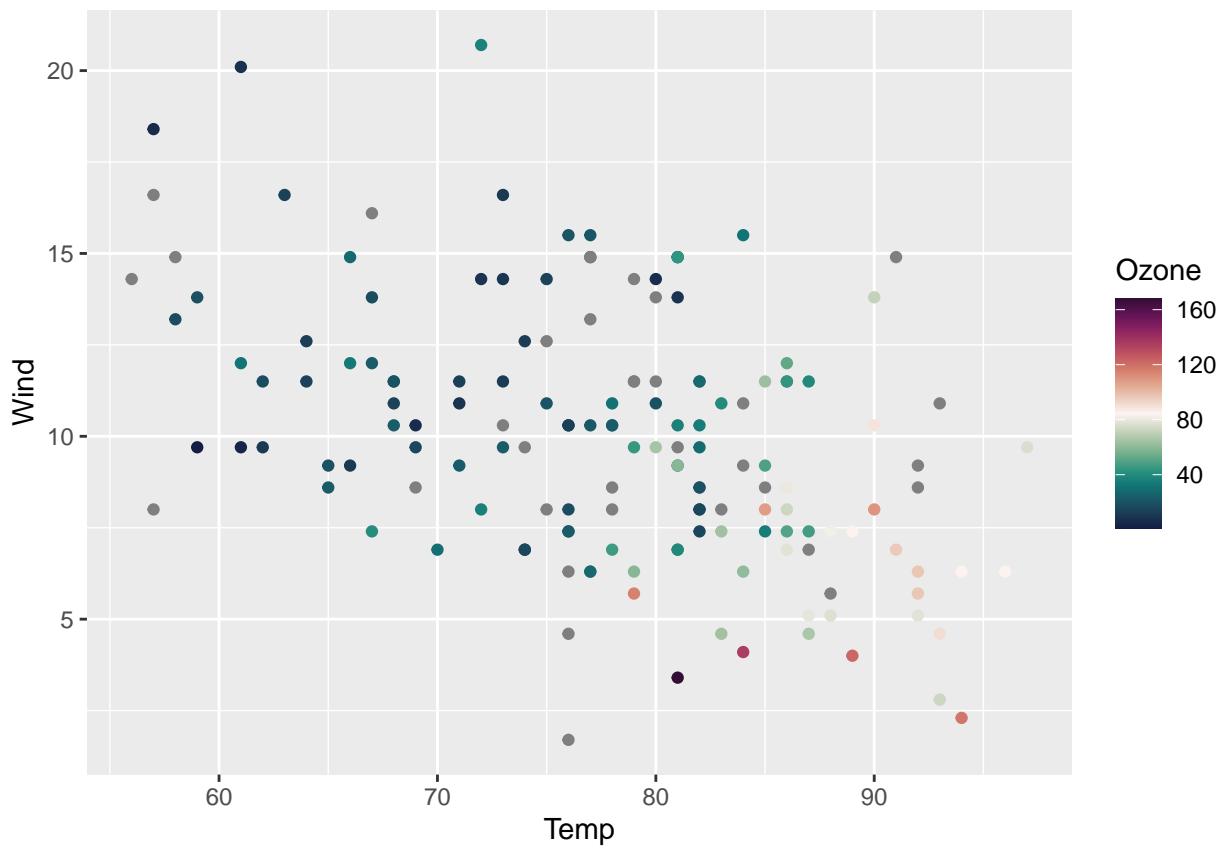
Las estéticas que pueden usarse para incluir variables adicionales son color (`color`), carácter (`pch`) y tamaño (`size`). Para variables continuas, el color es representado por un gradiente:

```
ggplot(data = airquality, aes(x = Temp, y = Wind,
                               col = Ozone)) +
  geom_point()
```



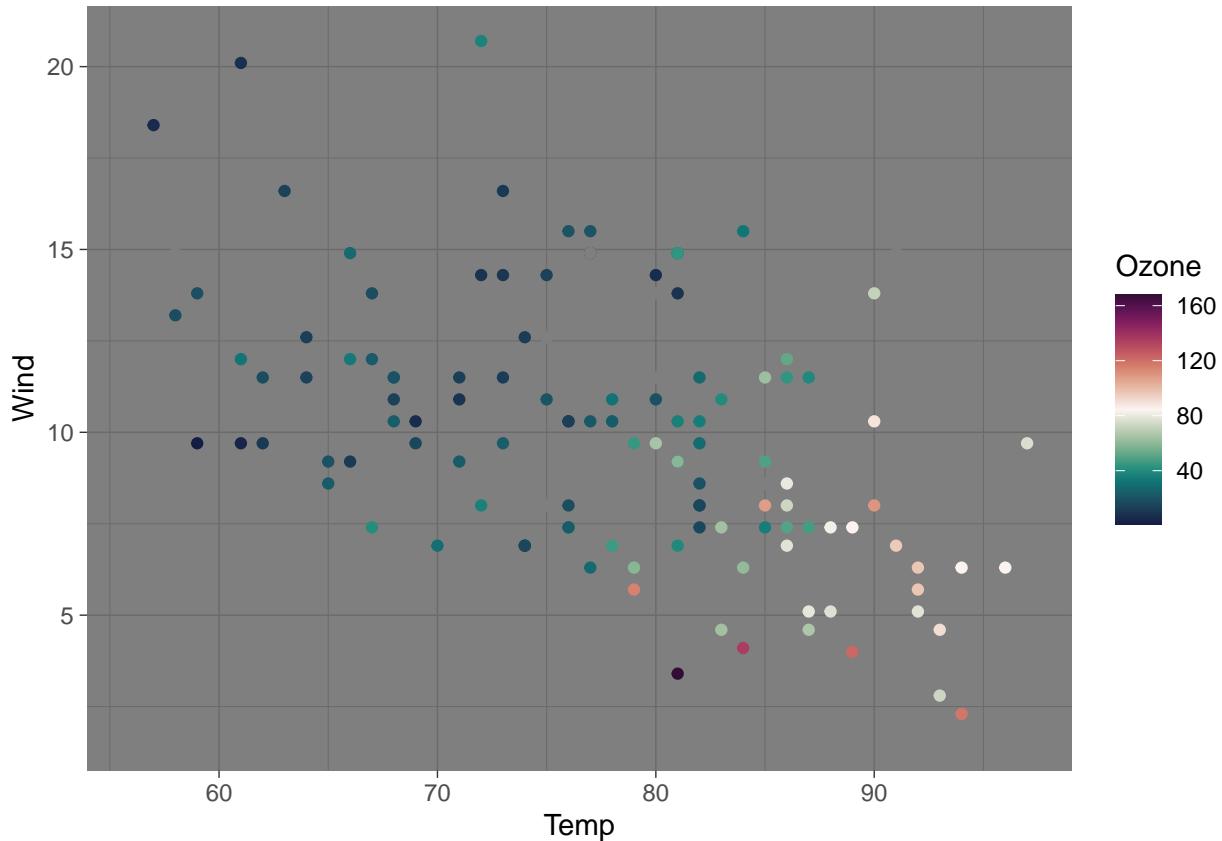
Este gradiente puede ser cambiado con `scale_color_palettes_c()`, eligiendo alguna de las paletas de `palettes_c_names`:

```
ggplot(data = airquality, aes(x = Temp, y = Wind,
                               col = Ozone)) +
  geom_point() +
  scale_color_palettes_c("pals::ocean(curl")
```



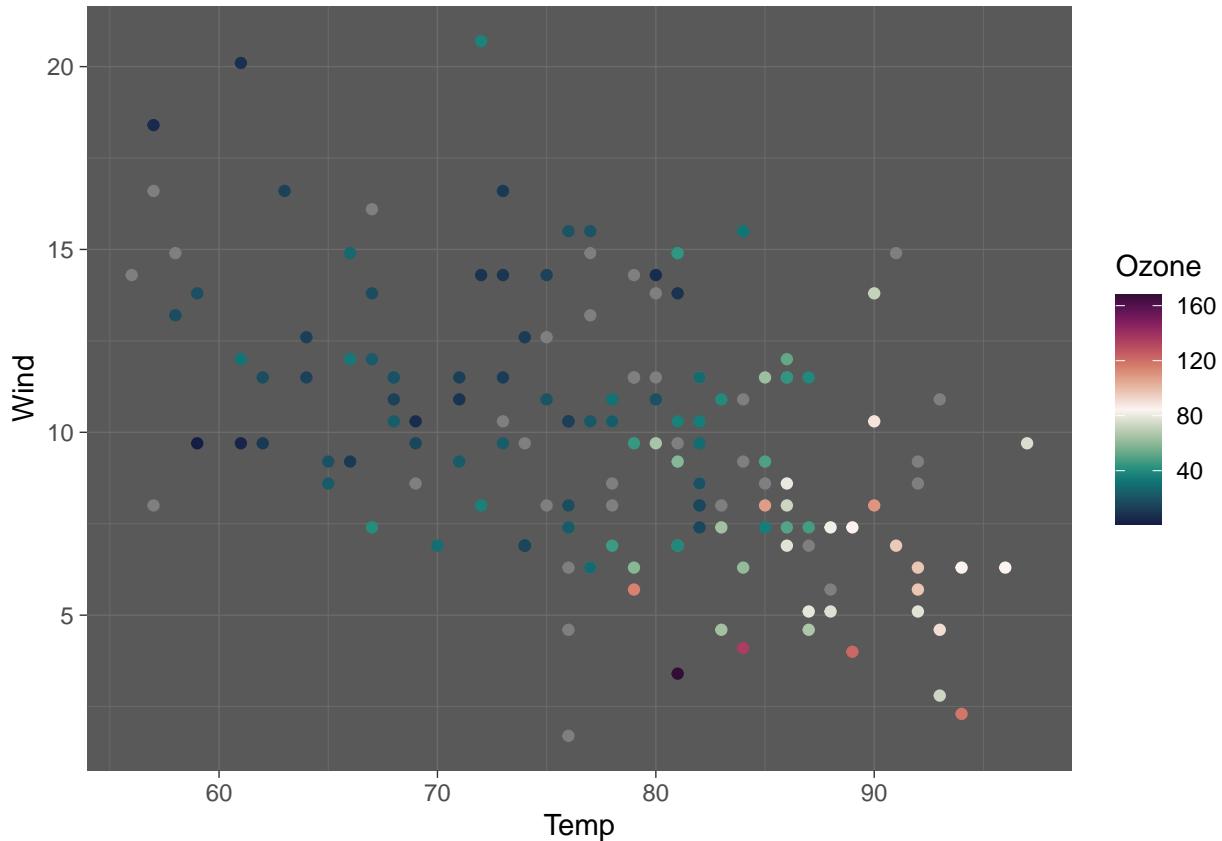
El uso del gradiente puede ser un problema si alguno de los colores no son visibles, como es el caso de los valores cercanos a 80 en el ejemplo anterior. Usando `theme_dark()` se puede mejorar la gráfica:

```
ggplot(data = airquality, aes(x = Temp, y = Wind,
                               col = Ozone)) +
  geom_point() +
  scale_color_palatteer_c("pals::ocean(curl") +
  theme_dark()
```



Ahora el problema es que algunos datos son demasiados oscuros para poder verse. El color de fondo puede cambiarse con `theme()` usando el argumento `panel.background`. Para este argumento debe usarse la función `element_rect()`, la cual tiene los argumentos para cambiar el color de fondo `fill`, entre otros posibles parámetros:

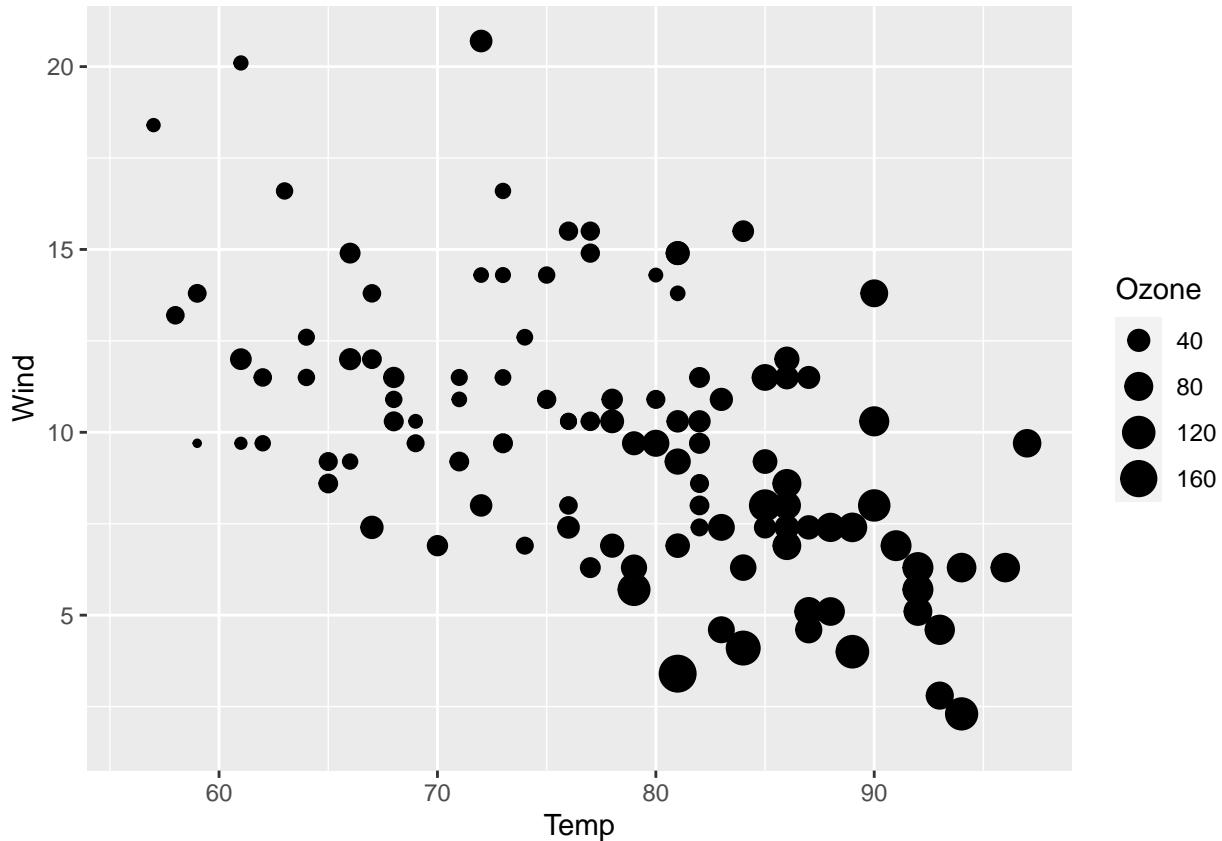
```
ggplot(data = airquality, aes(x = Temp, y = Wind,
                               col = Ozone)) +
  geom_point() +
  scale_color_palatteer_c("pals::ocean(curl") +
  theme_dark() +
  theme(panel.background = element_rect(fill = "grey35"))
```



De este modo se preservan todas las características de `theme_dark()`, solo se cambia el color de fondo y es posible ver nuevos datos que el fondo anterior ocultaba.

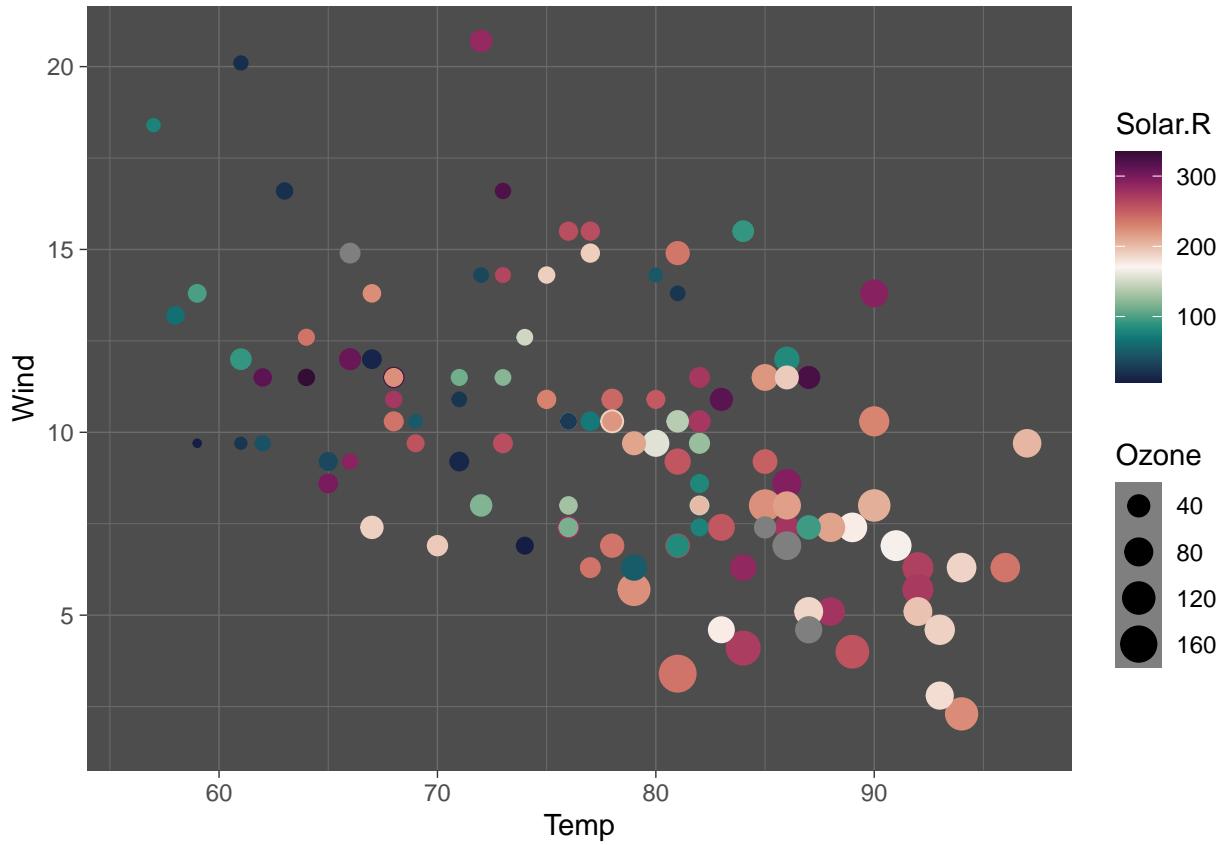
Cuando se usa el tamaño de los caracteres (`size`) para indicar una variable continua, cada observación toma un tamaño de acuerdo con su valor dentro del rango de los datos. En la leyenda únicamente se muestran algunos valores como referencia general:

```
ggplot(data = airquality, aes(x = Temp, y = Wind,
                               size = Ozone)) +
  geom_point()
## Warning: Removed 37 rows containing missing values (geom_point).
```



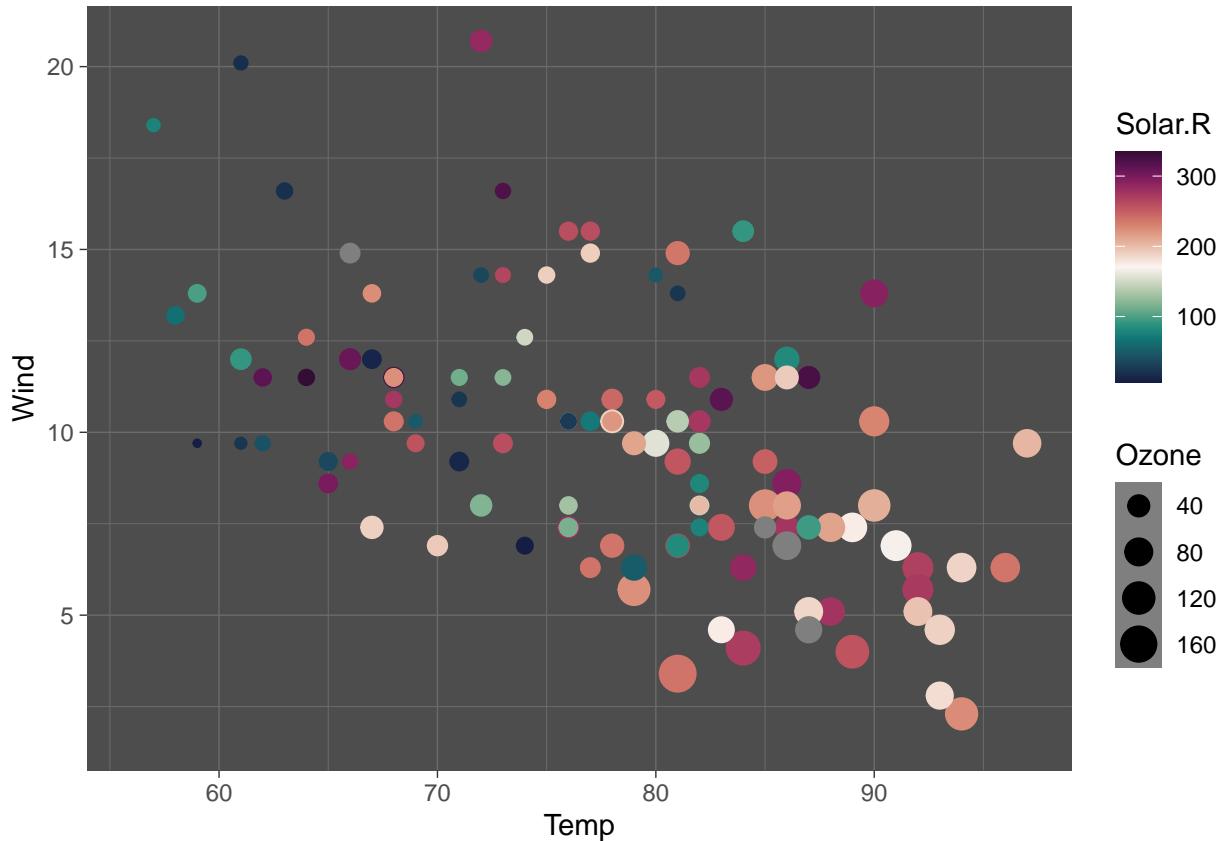
De esta manera es posible ver que a mayor temperatura y menos viento la cantidad de ozono incrementa. Cuando se usa más de una variable se muestran las leyendas correspondientes:

```
ggplot(data = airquality, aes(x = Temp, y = Wind,
                               col = Solar.R,
                               size = Ozone)) +
  geom_point() +
  scale_color_palatteer_c("pals::oceancurl") +
  theme_dark() +
  theme(panel.background = element_rect(fill = "grey30"))
## Warning: Removed 37 rows containing missing values (geom_point).
```



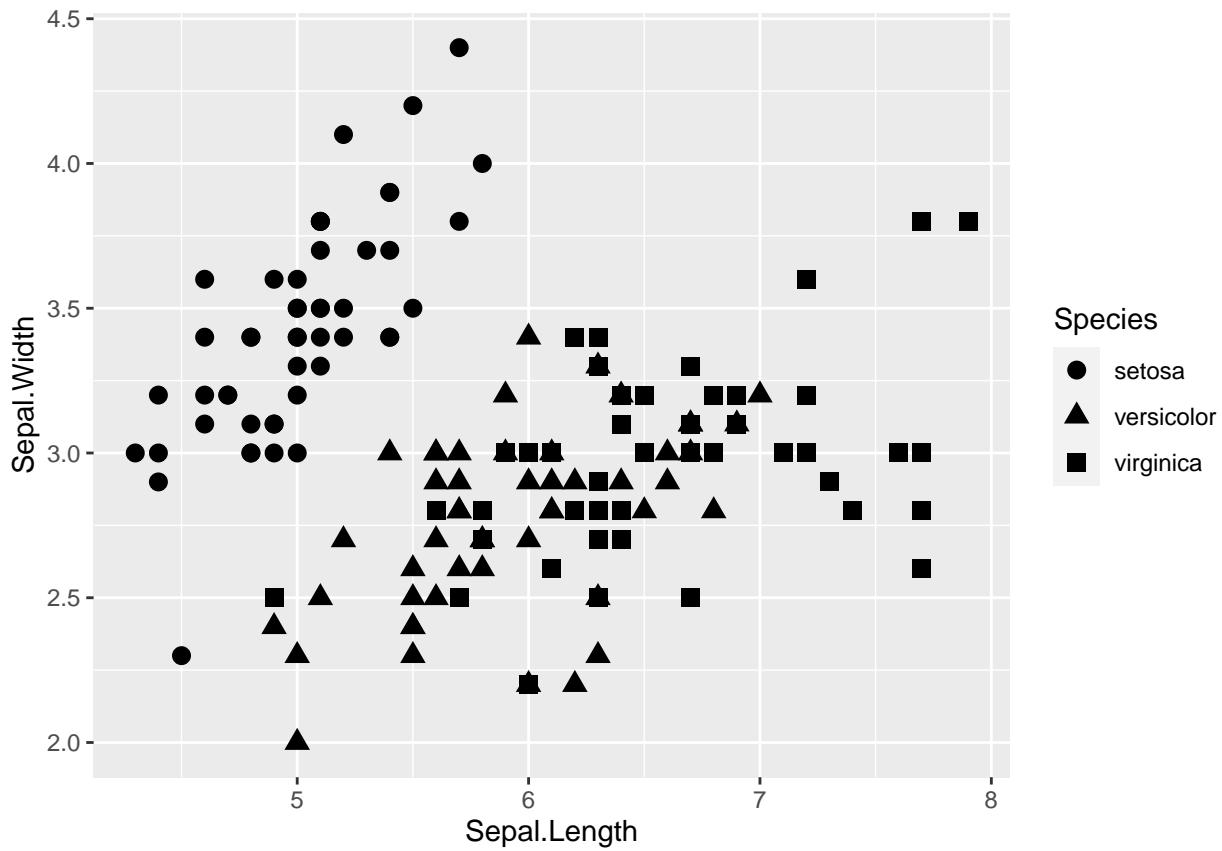
El acomodo de ambas puede modificarse con `legend.box` dentro de `theme()`:

```
ggplot(data = airquality, aes(x = Temp, y = Wind,
                               col = Solar.R,
                               size = Ozone)) +
  geom_point() +
  scale_color_palleteer_c("pals::ocean(curl") +
  theme_dark() +
  theme(panel.background = element_rect(fill = "grey30"),
        legend.box = "vertical")
## Warning: Removed 37 rows containing missing values (geom_point).
```



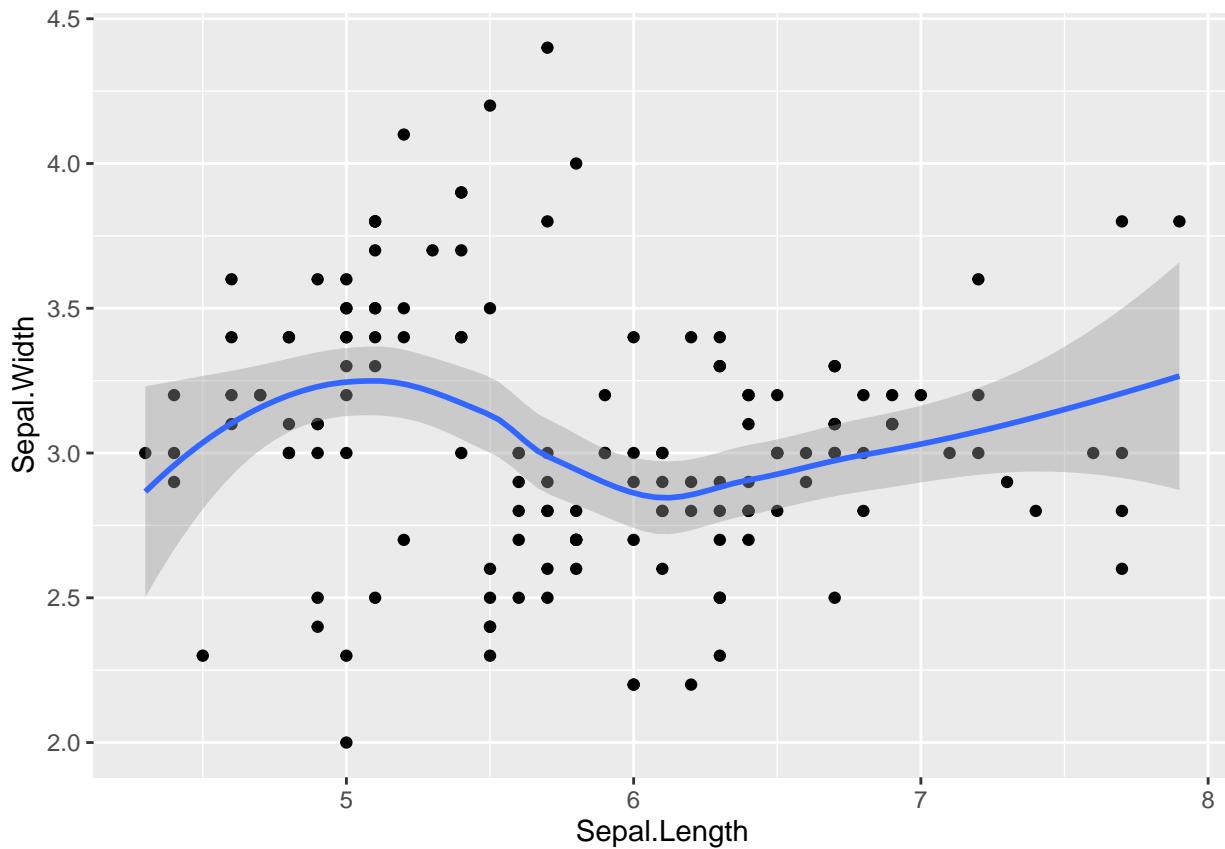
El uso de los caracteres (pch) es solo adecuado para variables categóricas, como es el caso de las diferentes especies presentes en `iris`:

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width,
                        pch = Species)) +
  geom_point(size = 3)
```



En las gráficas de dispersión, `geom_smooth()` muestra una linea que representa un modelo. Cuando el conjunto de datos tiene menos de 1000 observaciones se usa por defecto una regresión local (`loess`):

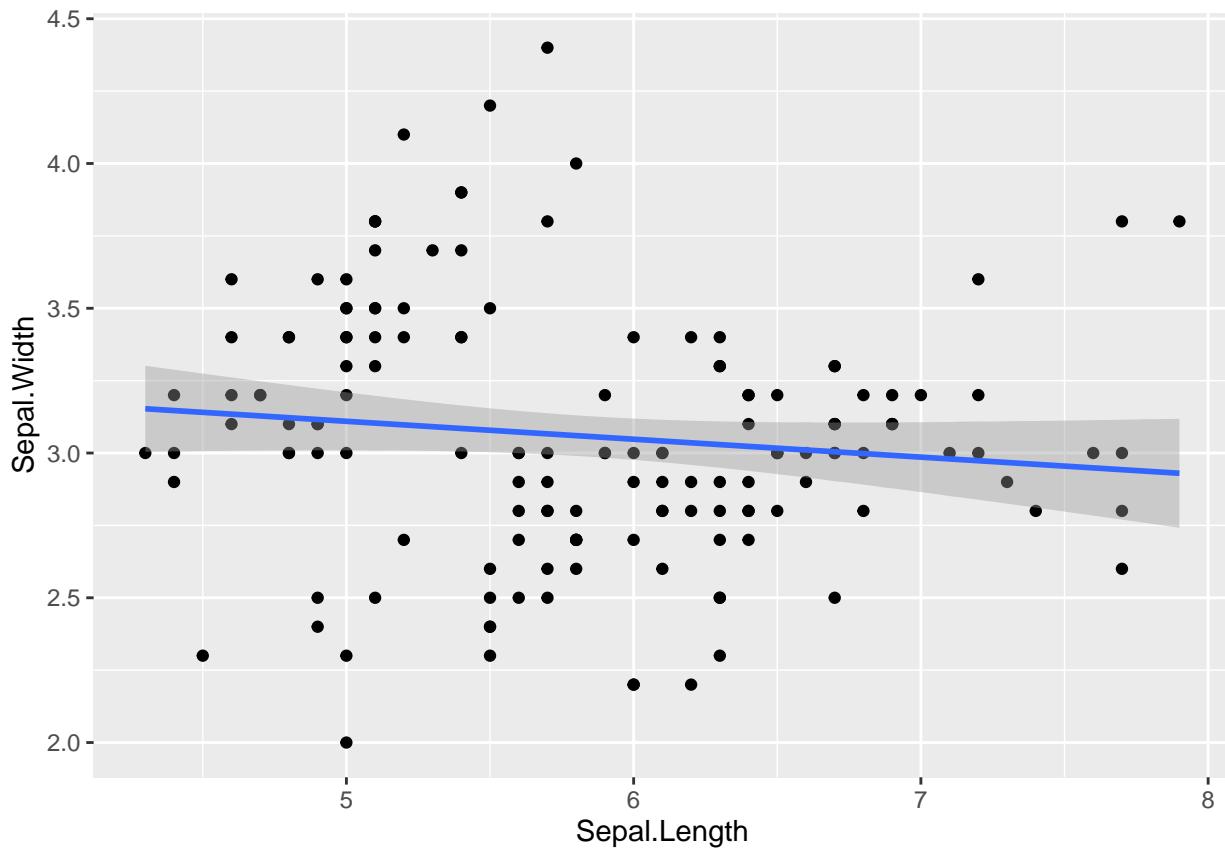
```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point() +
  geom_smooth()
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



En gris se muestra el intervalo de confianza, que por defecto es 0.95. Para cambiar el método a una regresión lineal se cambia el argumento `method`, colocando el valor `lm` (linear model).

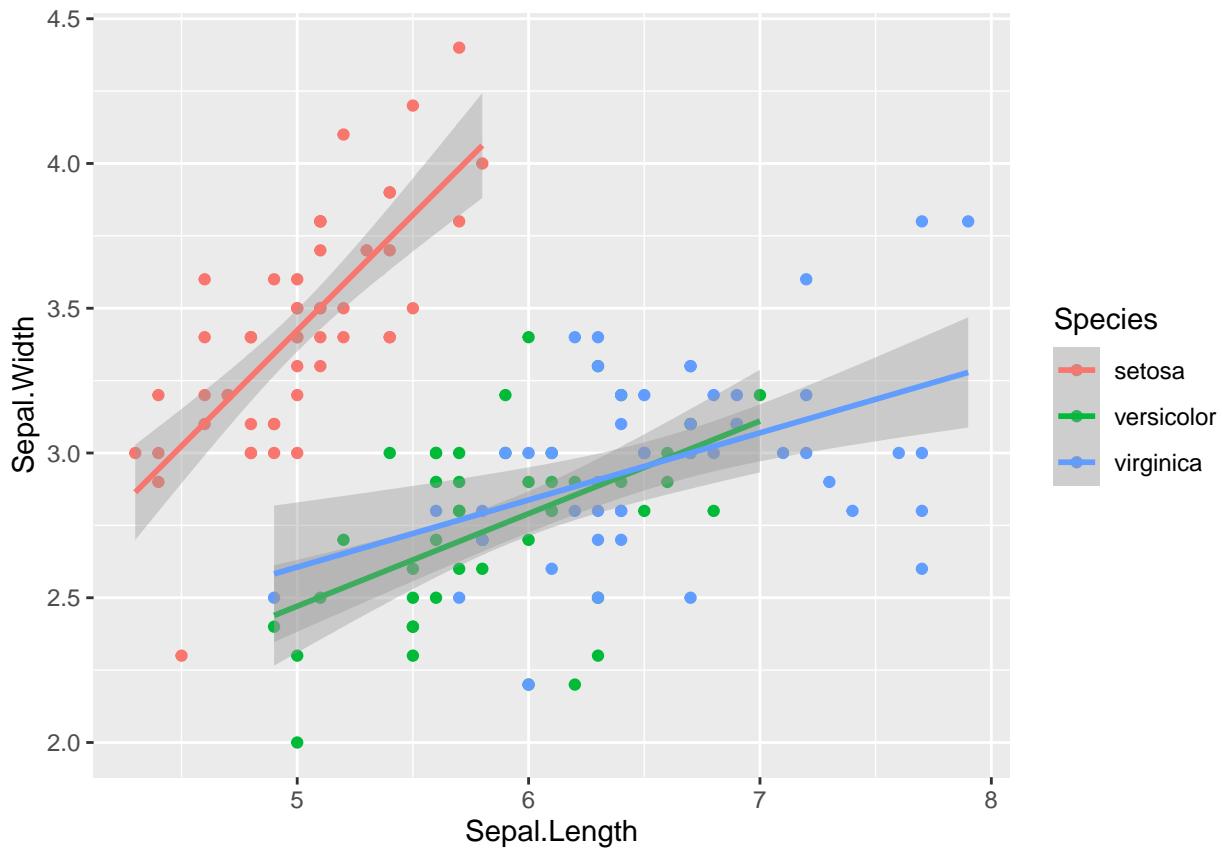
```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point() +
  geom_smooth(method = "lm")

## `geom_smooth()` using formula 'y ~ x'
```



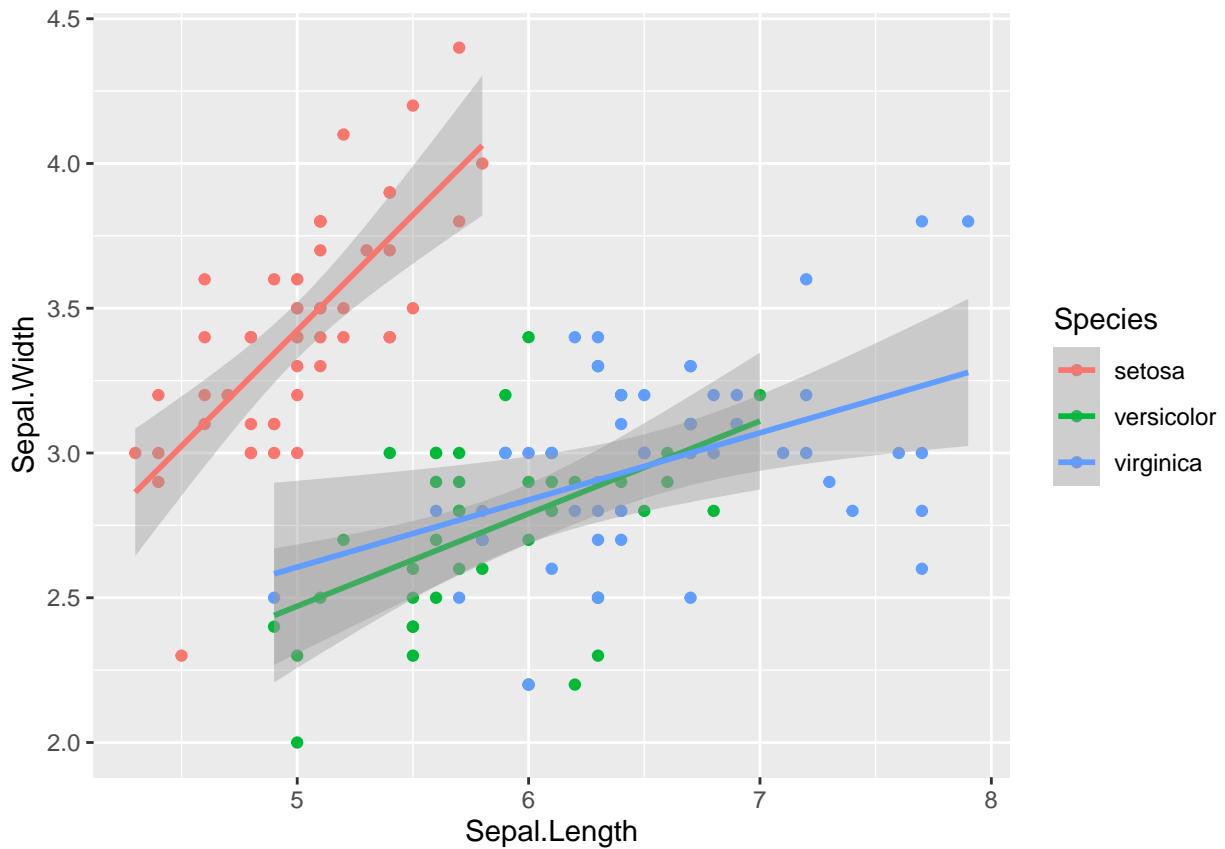
Al asignar una variable al color, las rectas también se ajustan a la separación, creando un modelo por cada categoría:

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width,
                        color = Species)) +
  geom_point() +
  geom_smooth(method = "lm")
## `geom_smooth()` using formula 'y ~ x'
```



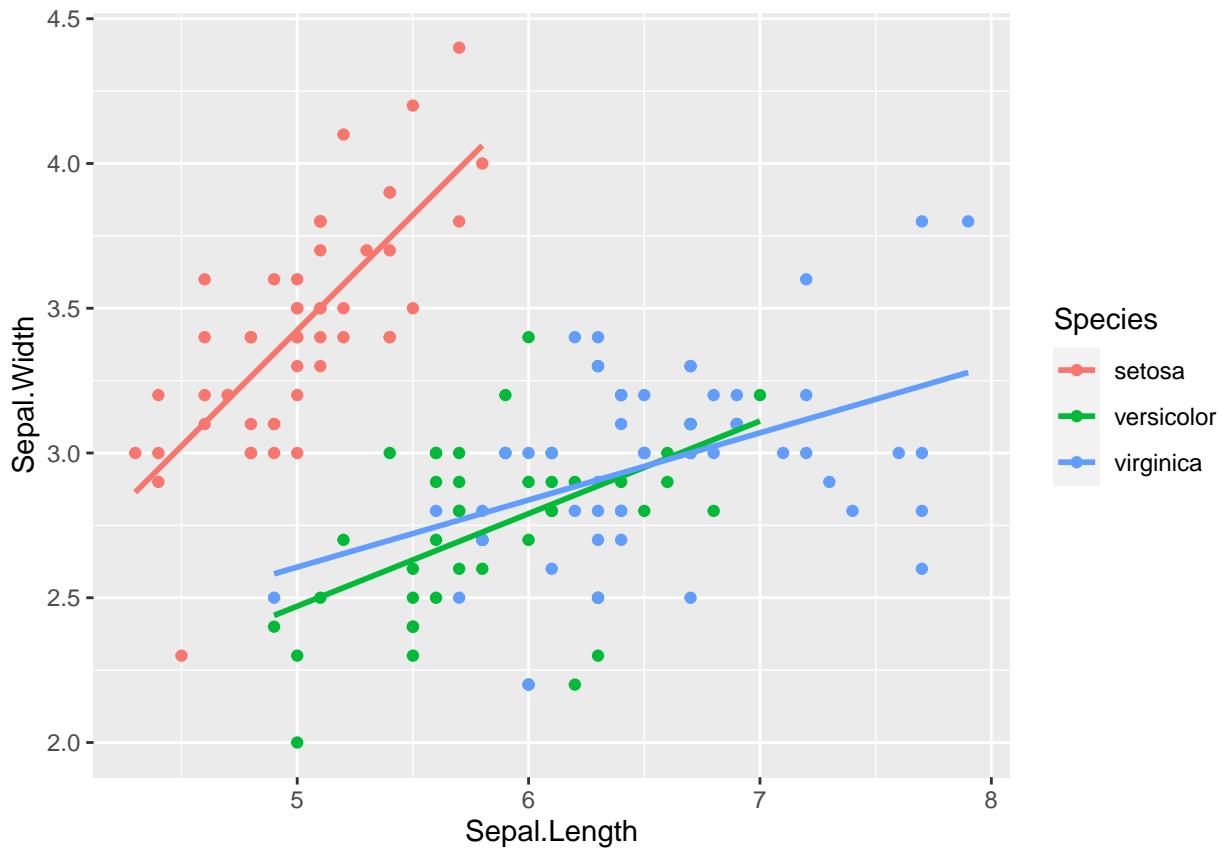
Si se desea cambiar el intervalo de confianza se modifica el argumento `level` con el valor deseado:

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width,
                        color = Species)) +
  geom_point() +
  geom_smooth(method = "lm", level = 0.99)
## `geom_smooth()` using formula 'y ~ x'
```



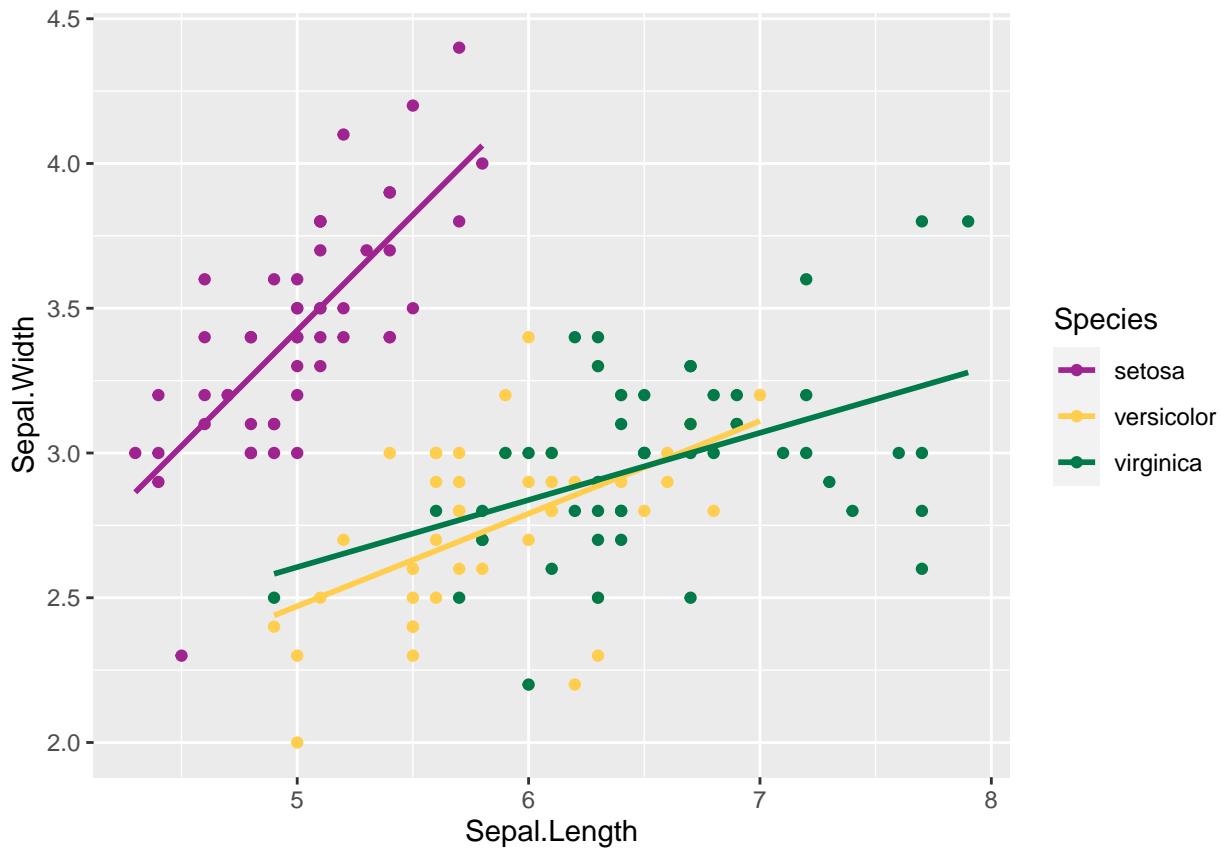
Por otra parte, para remover el intervalo de confianza se modifica el argumento `se`:

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width,
                        color = Species)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
## `geom_smooth()` using formula 'y ~ x'
```



El color de la linea es el mismo que el de los datos debido a que `color` fue indicado en `aes()`. Al cambiar estos colores con `scale_color_paletteseer_d()` los colores de las líneas también cambian:

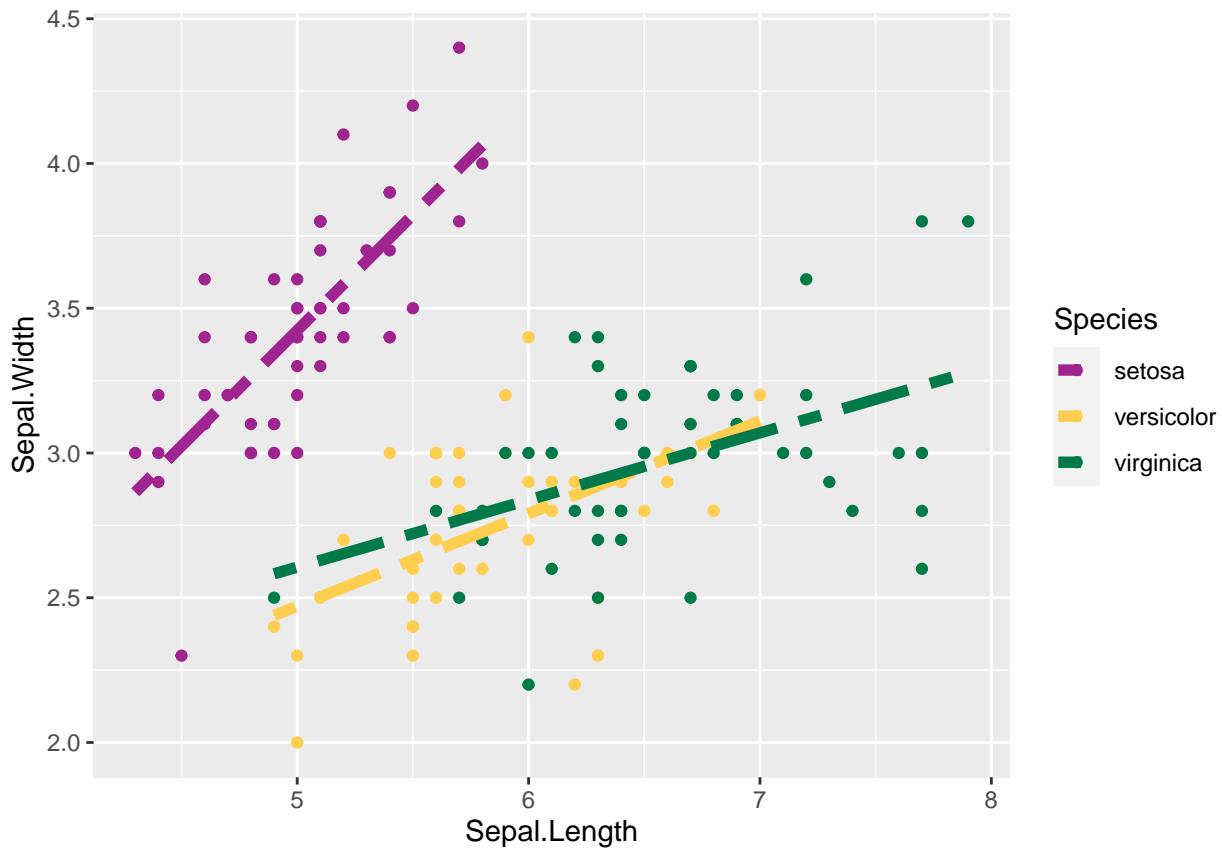
```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width,
                        color = Species)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  scale_color_paletteseer_d("awtools::spalettes")  
## `geom_smooth()` using formula 'y ~ x'
```



Otros atributos gráficos por fuera de `aes()` que pueden modificarse son el ancho y el tipo de línea dentro de `geom_smooth()`:

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width,
                        col = Species)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE,
              linetype = 6, size = 2) +
  scale_color_pasteer_d("awtools::spalette")

## `geom_smooth()` using formula 'y ~ x'
```



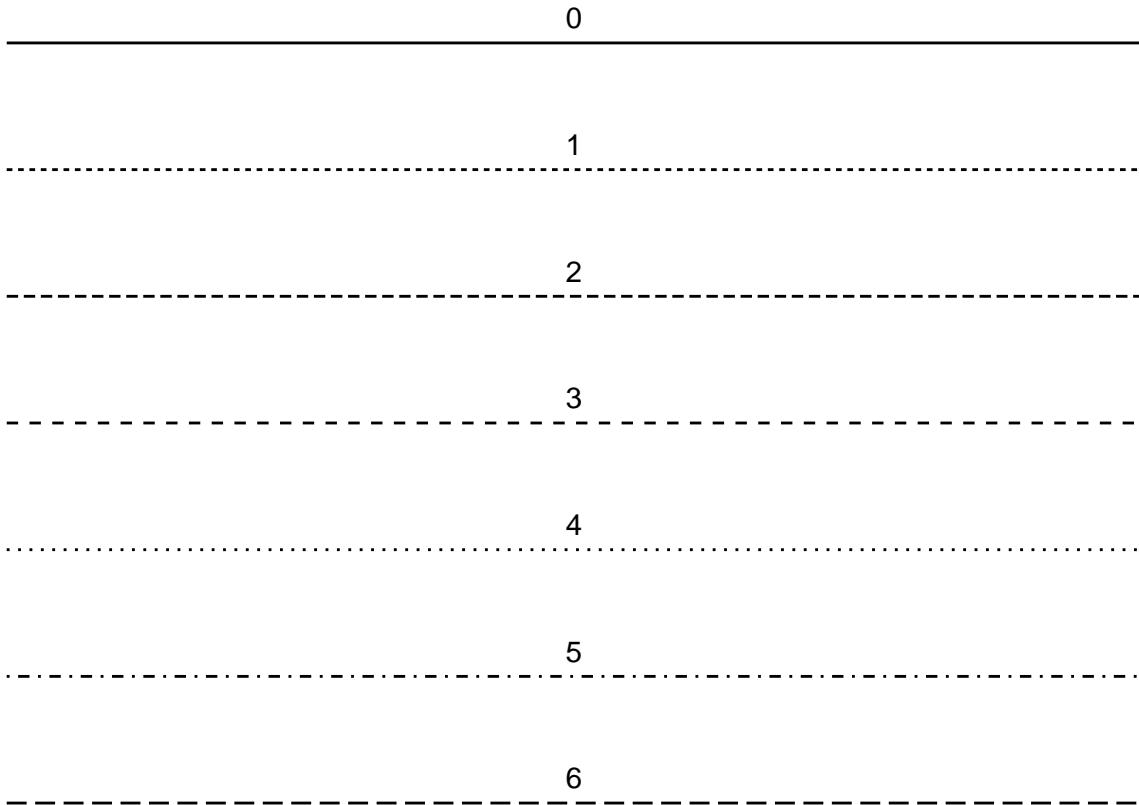
Los tipos de línea se especifican con número del 0 al 6, las cuales se muestran en la siguiente gráfica:

```
x <- rep(x = 5, times = 11)
y <- c()

for (i in 1:7) {
  y <- append(x = y, values = x - i)
}

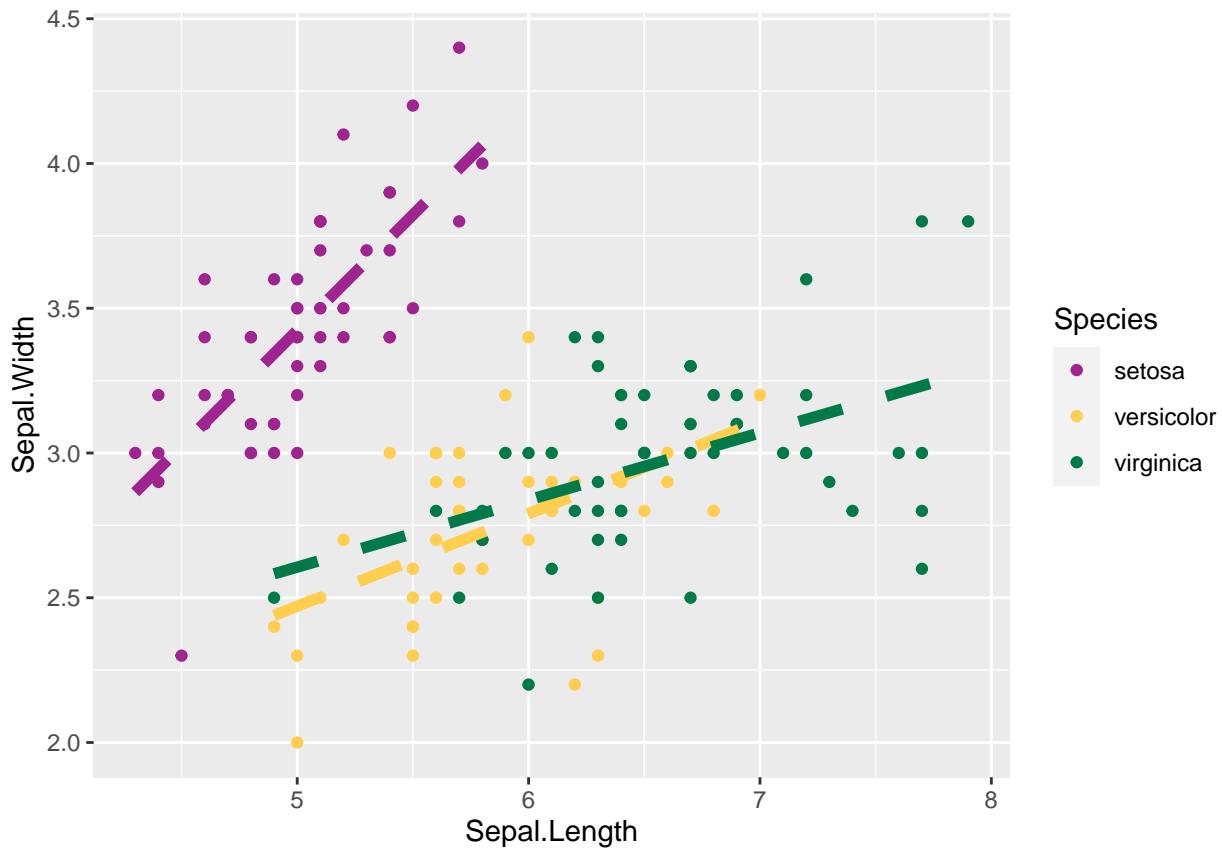
lineas <- tibble(x = rep(x = 0:10, times = 7),
                  y = y,
                  tipo = rep(x = 0:6, each = 11))

ggplot(data = lineas, aes(x = x, y = y,
                           linetype = factor(tipo))) +
  geom_line(color = "black") +
  annotate(geom = "text", x = rep(x = 5, times = 7),
           y = unique(lineas$y) + .2, label = 0:6) +
  theme_void() +
  theme(legend.position = "none")
```



Al usar `geom_smooth()` cambia la leyenda de la gráfica, la cual ahora muestra la línea con sus atributos como el ancho y tipo de línea para cada categoría. Para removerla de la leyenda se usa `show.legend`, que por defecto es verdadero:

```
ggplot(data = iris, aes(x = Sepal.Length, y = Sepal.Width,
                        col = Species)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE,
              linetype = 2, size = 2,
              show.legend = FALSE) +
  scale_color_palatteer_d("awtools::spalette")
## `geom_smooth()` using formula 'y ~ x'
```



5.12 Gráfica de líneas

Se construyen con `geom_line()`. Son usadas cuando los datos pueden ser organizados secuencialmente, por ejemplo, si es que siguen una variable que refleja el paso del tiempo. En el data frame `airquality` las observaciones cuentan con información sobre el mes y el día en que fueron registradas. De este modo se puede evaluar la temperatura de todo el mes de septiembre:

```
temp_sept <- airquality %>%
  filter(Month == 9)

temp_sept

## #  Ozone Solar.R Wind Temp Month Day
## 1    96     167  6.9   91     9    1
## 2    78     197  5.1   92     9    2
## 3    73     183  2.8   93     9    3
## 4    91     189  4.6   93     9    4
## 5    47     95   7.4   87     9    5
## 6    32     92  15.5   84     9    6
## 7    20    252 10.9   80     9    7
## 8    23    220 10.3   78     9    8
## 9    21    230 10.9   75     9    9
## 10   24    259  9.7   73     9   10
## 11   44    236 14.9   81     9   11
## 12   21    259 15.5   76     9   12
## 13   28    238  6.3   77     9   13
## 14    9     24 10.9   71     9   14
## 15   13    112 11.5   71     9   15
```

```

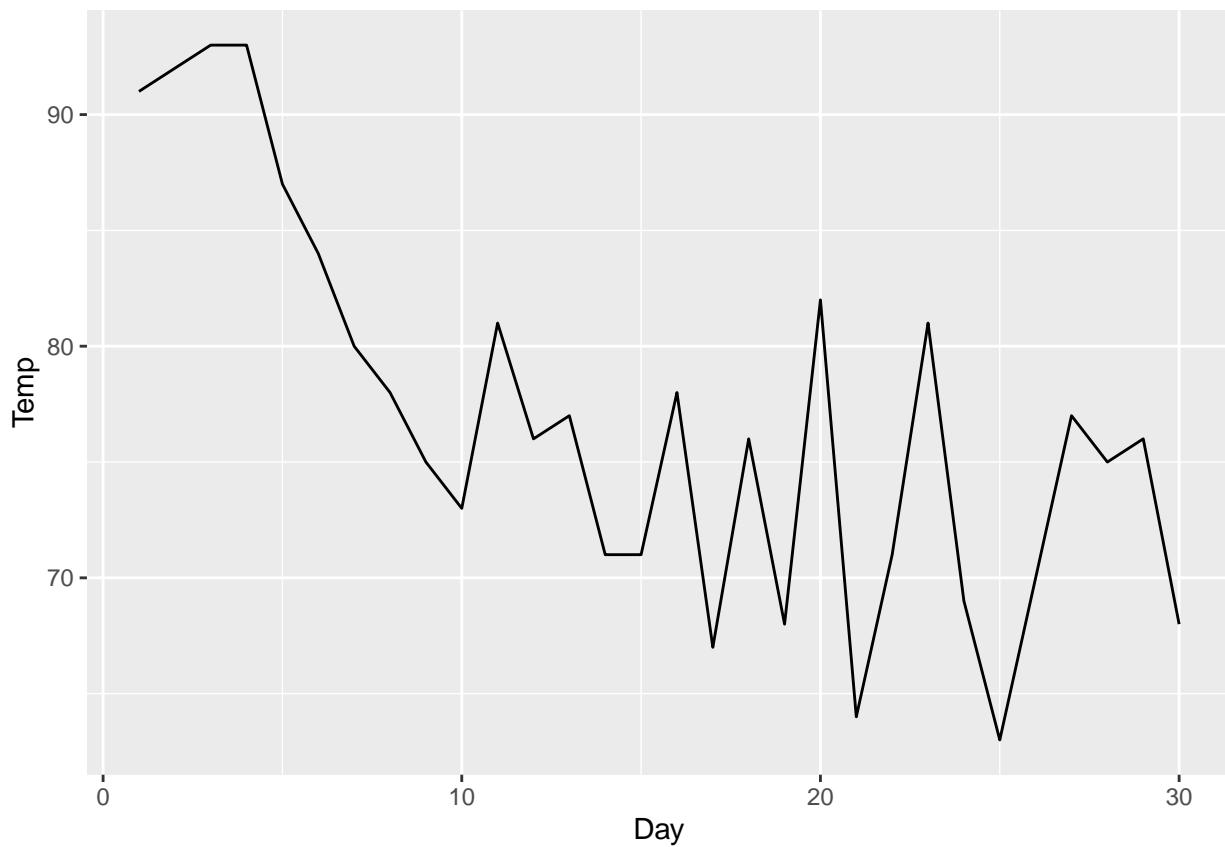
## 16    46    237   6.9    78      9    16
## 17    18    224  13.8    67      9    17
## 18    13     27 10.3    76      9    18
## 19    24    238 10.3    68      9    19
## 20    16    201   8.0    82      9    20
## 21    13    238 12.6    64      9    21
## 22    23     14  9.2    71      9    22
## 23    36    139 10.3    81      9    23
## 24     7     49 10.3    69      9    24
## 25    14     20 16.6    63      9    25
## 26    30    193   6.9    70      9    26
## 27    NA    145 13.2    77      9    27
## 28    14    191 14.3    75      9    28
## 29    18    131   8.0    76      9    29
## 30    20    223 11.5    68      9    30

```

```

ggplot(data = temp_sept, aes(x = Day, y = Temp)) +
  geom_line()

```

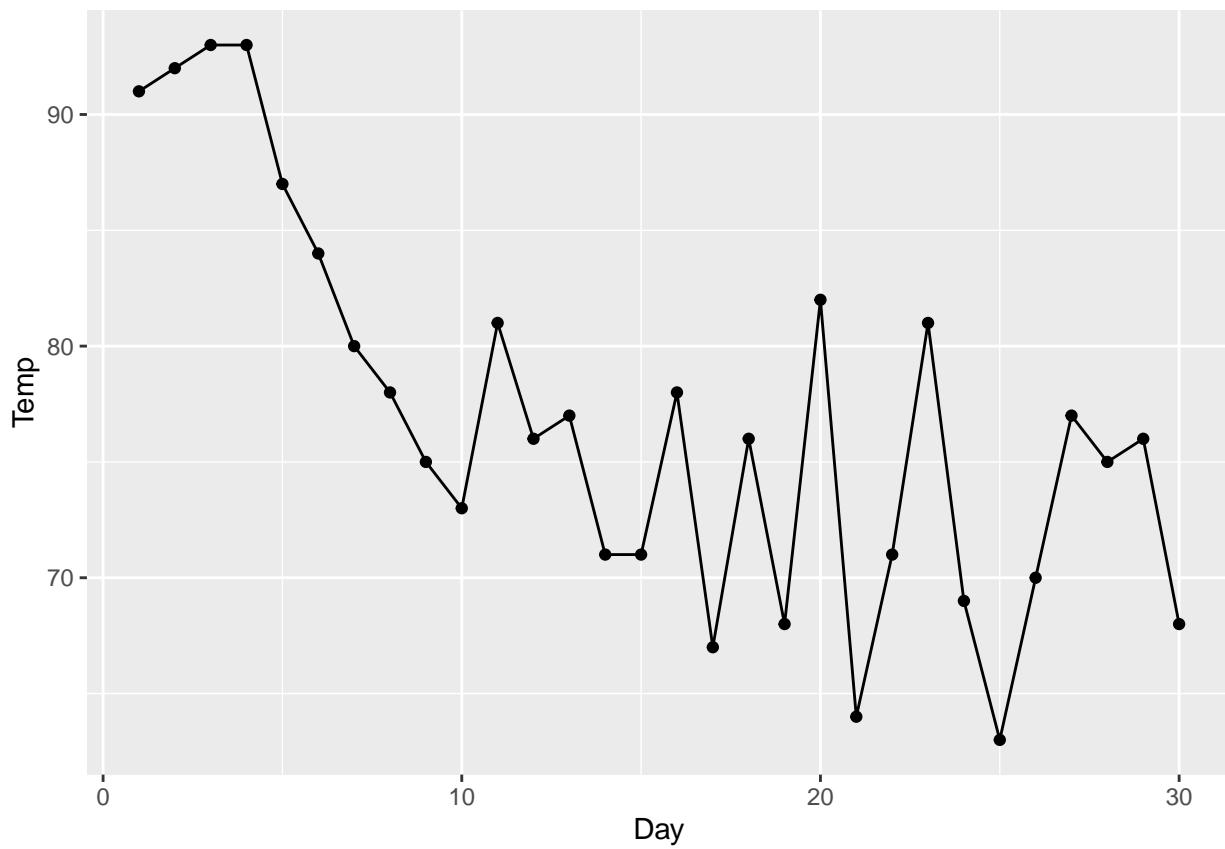


Se puede usar también `geom_point()` para crear una gráfica de puntos y líneas:

```

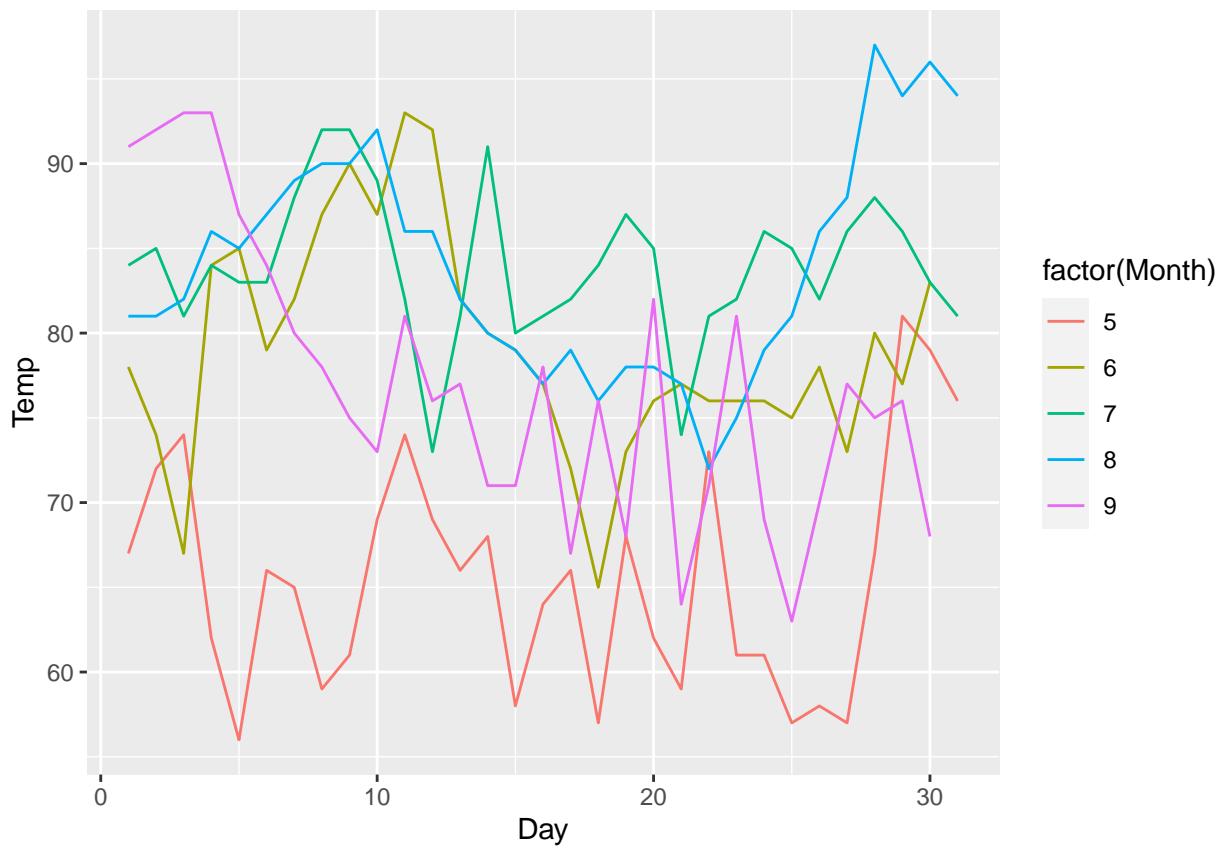
ggplot(data = temp_sept, aes(x = Day, y = Temp)) +
  geom_line() +
  geom_point()

```



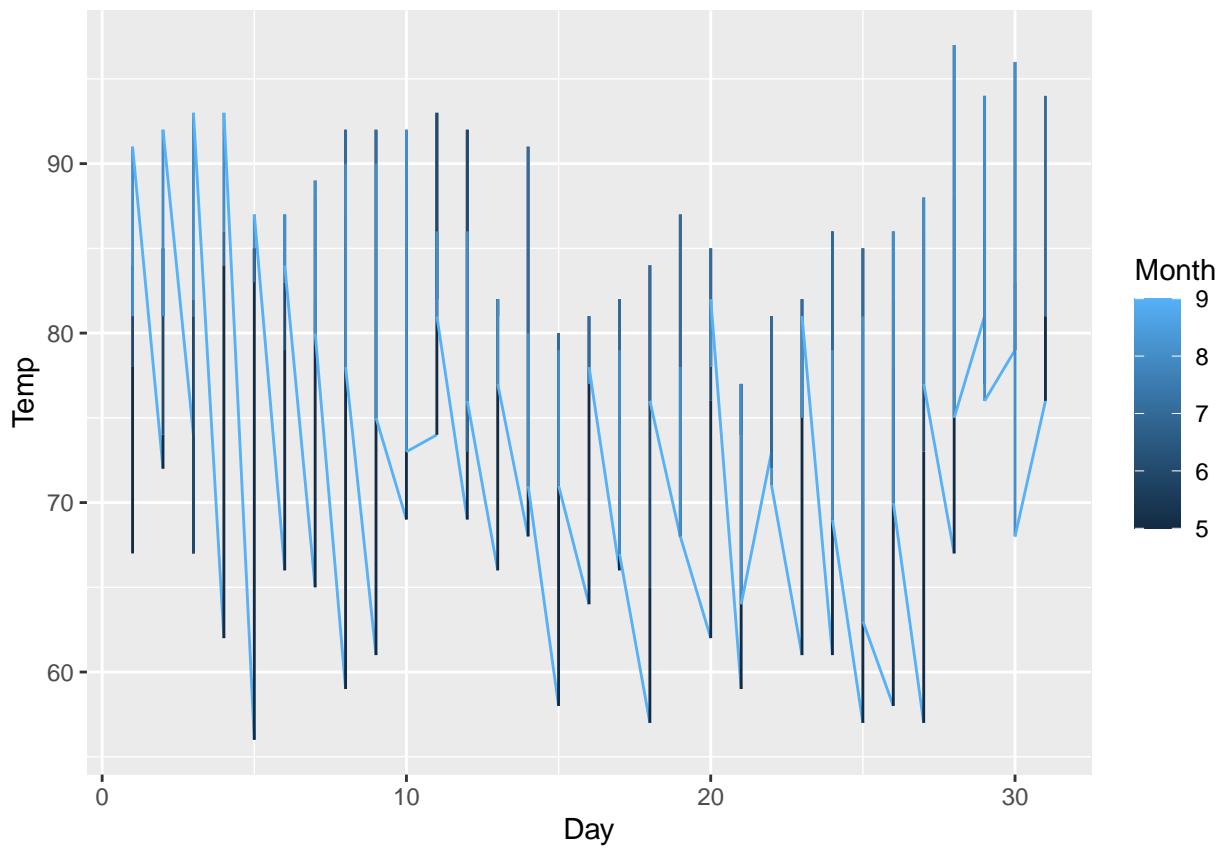
Usando todos los datos de `airquality` es posible graficar la temperatura de todos los meses al colocar el color en las estéticas para separar los datos de cada mes:

```
ggplot(data = airquality, aes(x = Day, y = Temp,  
                               color = factor(Month))) +  
  geom_line()
```



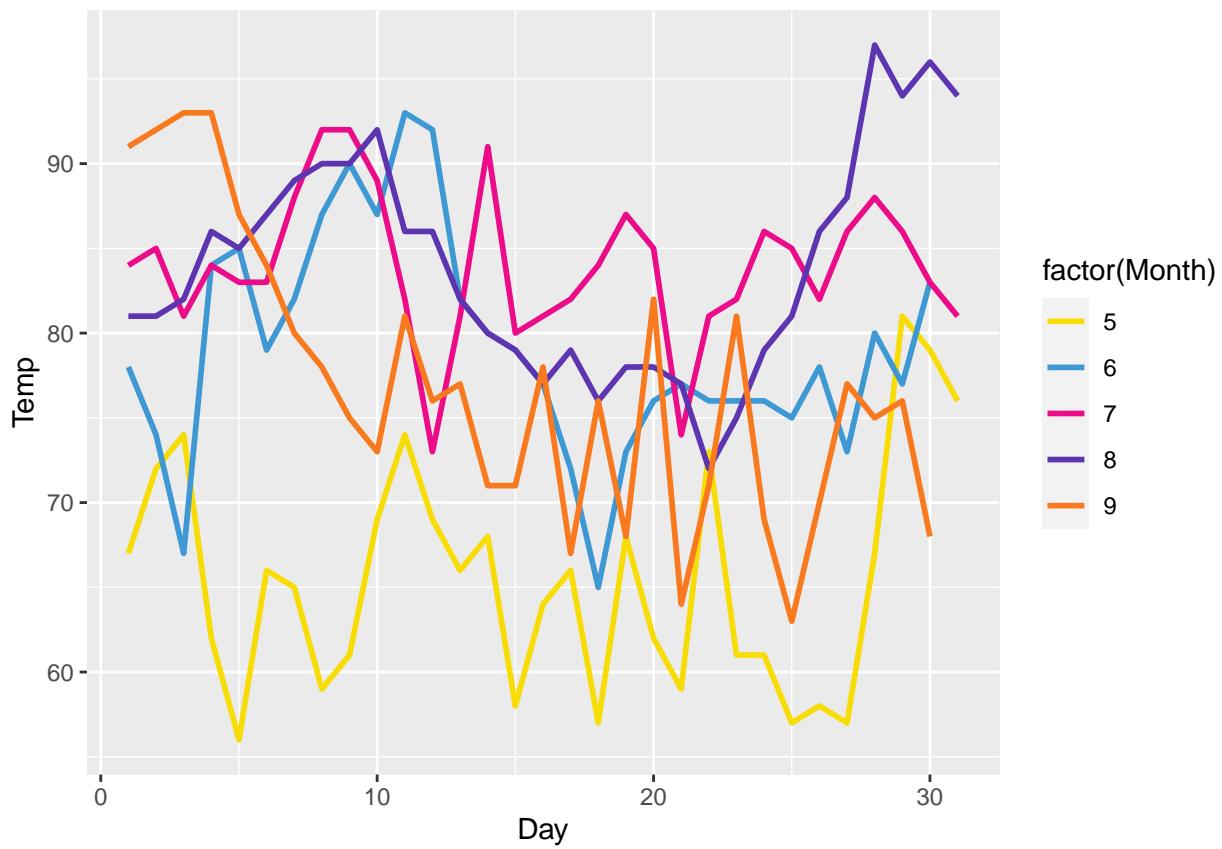
Month debe ser introducido como un factor debido a que de lo contrario es interpretado como una variable continua y la separación no ocurre adecuadamente:

```
ggplot(data = airquality, aes(x = Day, y = Temp,
                               color = Month)) +
  geom_line()
```



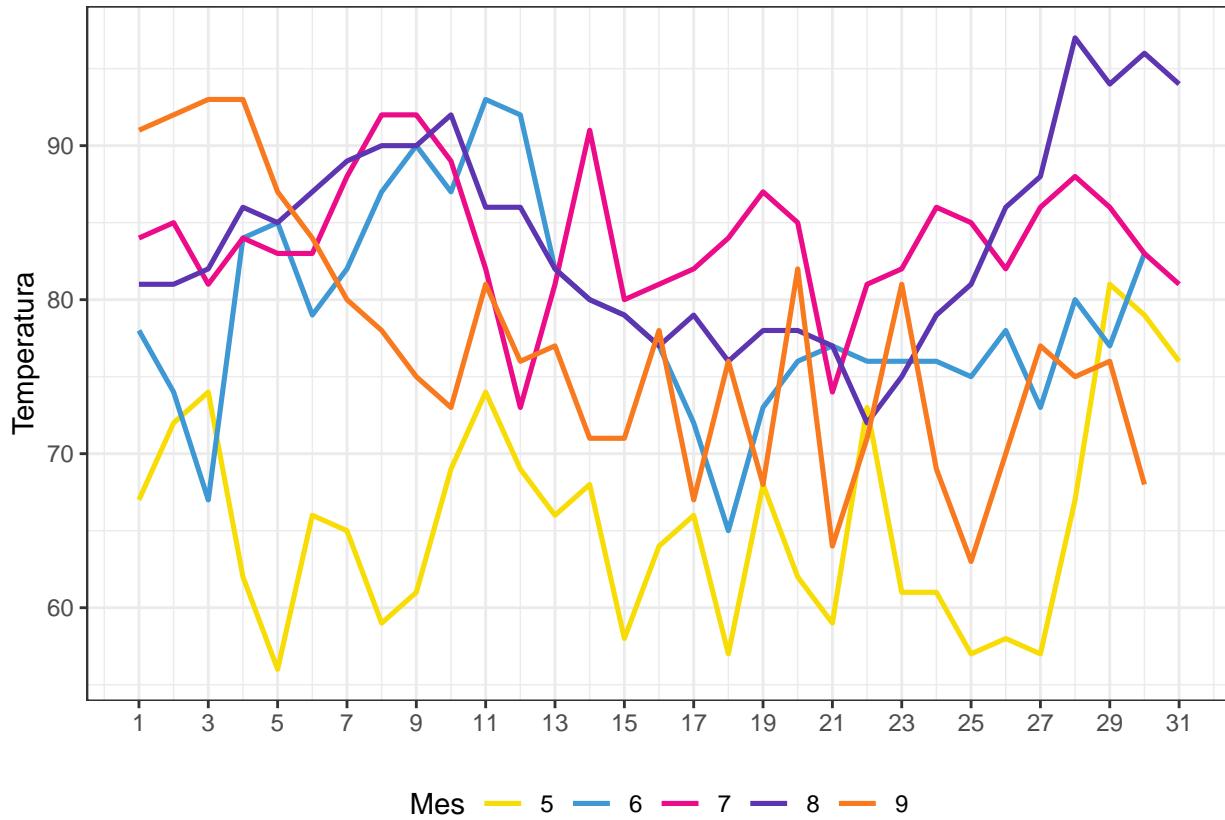
Para mejorar la separación de cada línea se puede usar una paleta diferente e incrementar el ancho de la línea:

```
ggplot(data = airquality, aes(x = Day, y = Temp,
                               color = factor(Month))) +
  geom_line(size = 1) +
  scale_color_pasteer_d("awtools::ppalette")
```



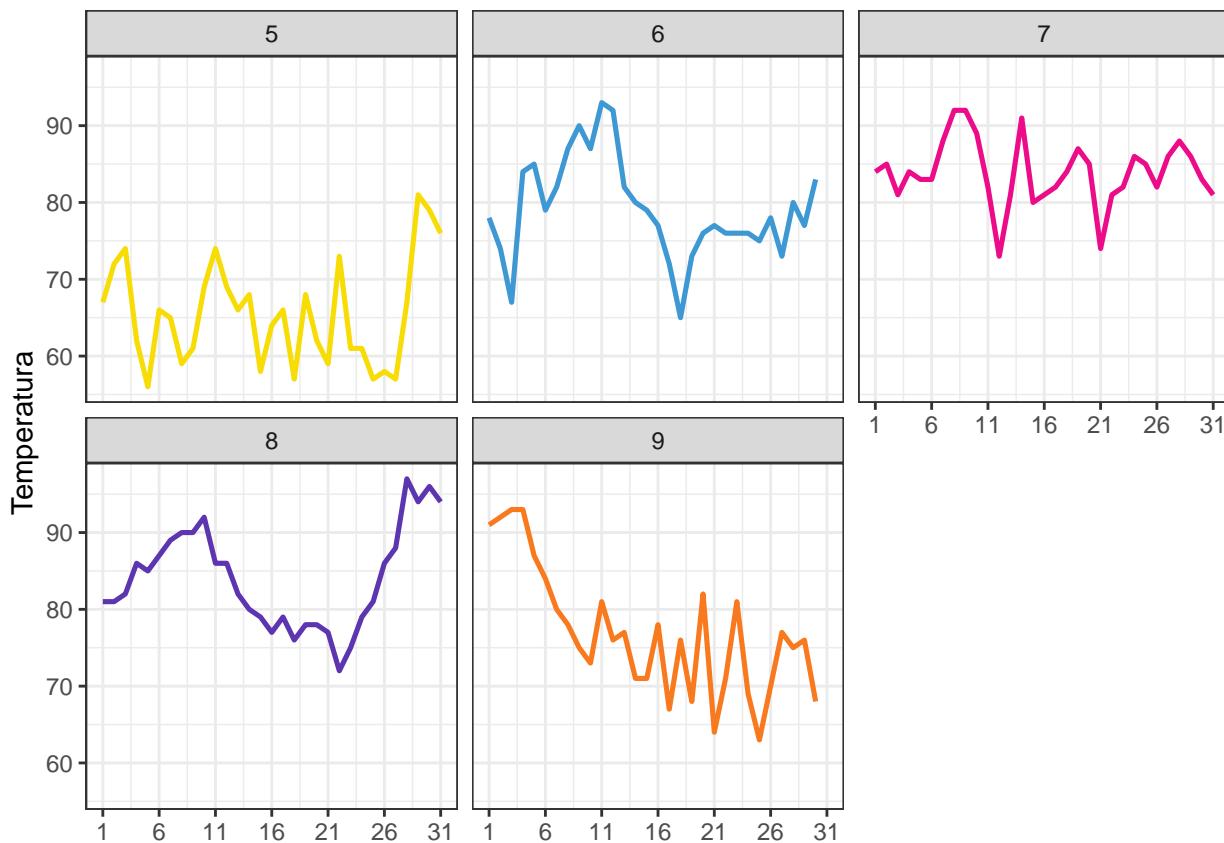
La gráfica terminada:

```
ggplot(data = airquality, aes(x = Day, y = Temp,
                               color = factor(Month))) +
  geom_line(size = .9) +
  scale_color_paletteer_d("awtools::ppalette") +
  labs(
    x = NULL,
    y = "Temperatura",
    color = "Mes",
  ) +
  scale_x_continuous(breaks = seq(from = 1, to = 31, by = 2)) +
  theme_bw() +
  theme(legend.position = "bottom")
```



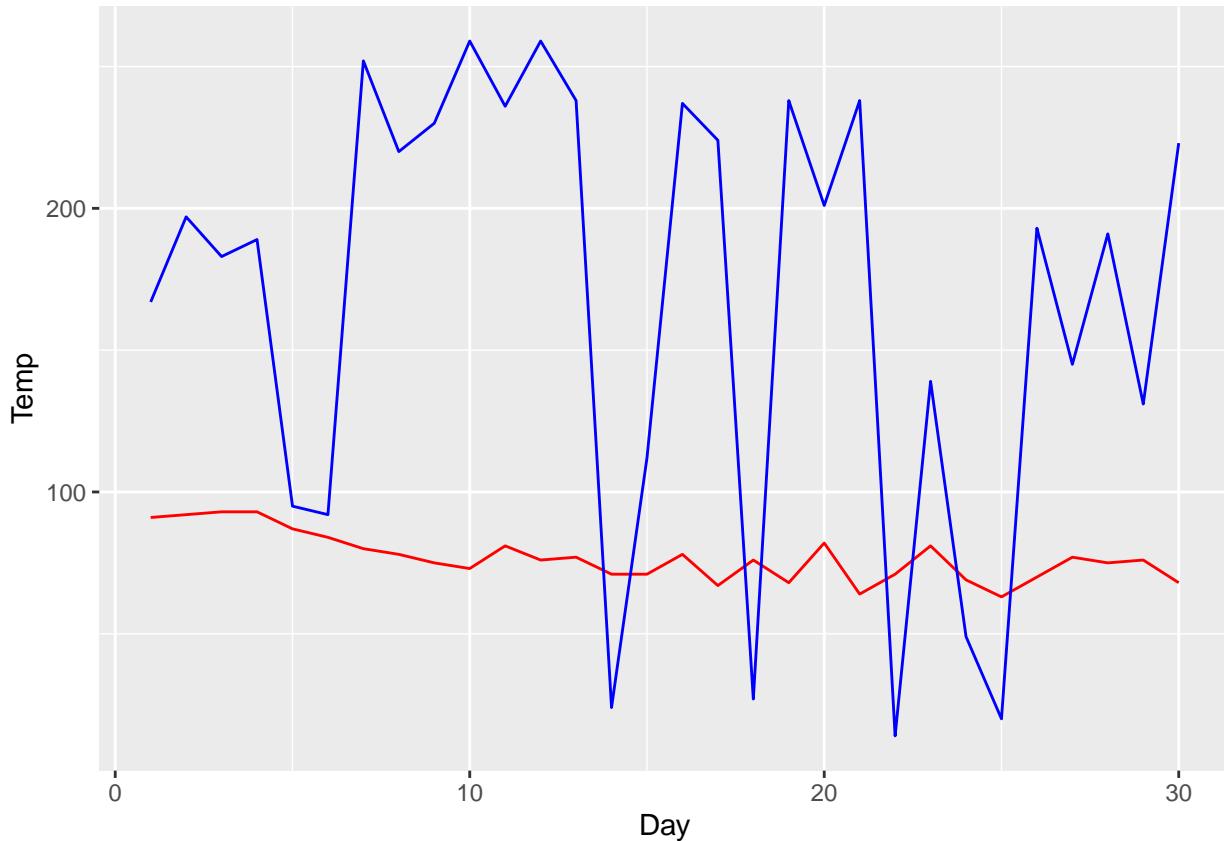
Otra manera de presentar la información es mediante paneles:

```
ggplot(data = airquality, aes(x = Day, y = Temp,
                               color = factor(Month))) +
  geom_line(size = .9) +
  scale_color_paletteer_d("awtools::ppalette") +
  labs(
    x = NULL,
    y = "Temperatura",
    color = "Mes",
  ) +
  scale_x_continuous(breaks = seq(from = 1, to = 31, by = 5)) +
  theme_bw() +
  theme(legend.position = "none") +
  facet_wrap(~ Month)
```



Una manera incorrecta de representar la información contenida en `airquality` sería incorporar en la misma gráfica datos de temperatura y cantidad de ozono, por ejemplo. El error en este caso consiste en que cada una de estas variables usa unidades diferentes, la temperatura está medida en grados Fahrenheit y la concentración de ozono en partes por mil millones. Al colocar ambas líneas en la misma gráfica, es posible asumir que son datos comparables cuando en realidad no es el caso:

```
ggplot(data = temp_sept) +
  geom_line(aes(x = Day, y = Temp), color = "red") +
  geom_line(aes(x = Day, y = Solar.R), color = "blue")
```



En general, cuando se quieran graficar datos de variables con unidades diferentes, deben colocarse en gráficas separadas.

6 Mapas

Los mapas se realizan con los paquetes `ggplot2` y con `sf`. Este último permite el fácil manejo de archivos shapefile (.shp), formato en el cual se usarán los datos espaciales.

6.1 Obtención de datos y creación de capas

Esta primera parte describe cómo crear mapas de distribución en México. Para ello, primero se necesita un archivo shapefile de la división política del país, la cual puede ser obtenida en el portal de geoinformación de la CONABIO. Los archivos requeridos se encuentran en la categoría de división política, en las subcategorías de Estatal y Límite se encuentran la División política estatal 1:250000.2020 y el Límite Nacional 1:250000, respectivamente. Al seleccionar cualquiera de estos dos Temas, en la pestaña Metadatos se encuentran los detalles de cada uno. En la sección de Descargas se debe seleccionar la opción Shapefile (Coordenadas geográficas) para su uso en esta sección.

Se obtendrá un archivo comprimido, el cual contiene una carpeta con todos los archivos requeridos por el formato .shp. Para introducir esta información se usa la función `st_read()` del paquete `sf`. Con el argumento `dsn` se selecciona únicamente el archivo con terminación .shp:

```
mapa_mexico <- st_read(dsn = "datos_manual/mapas/mexico/mapa_mexico/contdv1mgw.shp")
## Reading layer `contdv1mgw` from data source
##   `/home/leot/Documents/programacion/R/introduccion_R_biologia/datos_manual/mapas/mexico/mapa_mexico'
##   using driver `ESRI Shapefile'
## Simple feature collection with 358 features and 4 fields
```

```

## Geometry type: POLYGON
## Dimension: XY
## Bounding box: xmin: -118.366 ymin: 14.53401 xmax: -86.71074 ymax: 32.71877
## Geodetic CRS: WGS 84

```

Al leer el archivo se imprimen también algunas características del mapa, de las cuales **Geodetic CRS** (Coordinate Reference System) será útil después. Al consultar el tipo de objeto se obtiene **sf**, que significa “simple features”, una manera de almacenar información geográfica:

```

class(mapa_mexico)
## [1] "sf"           "data.frame"

```

Al imprimir el objeto se obtiene el conjunto de datos relacionados con los polígonos del mapa. Para la construcción de las capas no se usarán coordenadas, en cambio se especificará dentro de las estéticas la columna **geometry**, la cual contiene las coordenadas de cada polígono.

```

head(mapa_mexico)

## Simple feature collection with 6 features and 4 fields
## Geometry type: POLYGON
## Dimension: XY
## Bounding box: xmin: -117.301 ymin: 14.53401 xmax: -86.74039 ymax: 32.71877
## Geodetic CRS: WGS 84
##          AREA    PERIMETER COV_ID      geometry
## 1 1.731249e+02 185.03666278     2 404 POLYGON((-114.7575 32.7162...
## 2 4.694670e-05  0.03188133     3     2 POLYGON((-117.2939 32.4342...
## 3 1.770398e-05  0.01756151     4     3 POLYGON((-117.2585 32.4173...
## 4 1.608039e-04  0.06796011     5     4 POLYGON((-117.2393 32.3949...
## 5 1.271809e-02   0.51603622     6     8 POLYGON((-114.7698 31.7067...
## 6 3.505660e-05  0.02479165     7     6 POLYGON((-116.8074 31.8138...

```

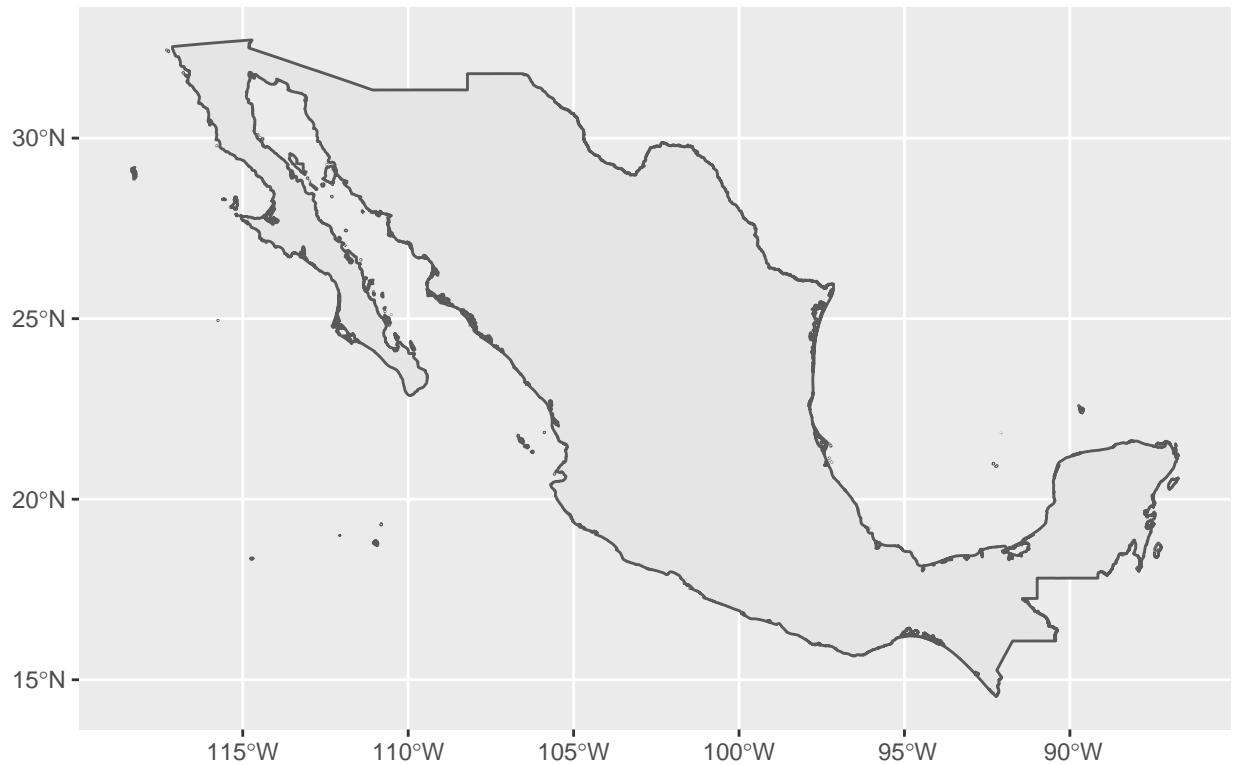
La función que permite usar esta información con **ggplot()** es **geom_sf()**, la cual funciona como cualquier otra capa usada previamente. Para la creación del mapa se usarán objetos diferentes, por lo que es mejor dejar **ggplot()** vacía y en cambio especificar **data** y **aes()** en cada una de las capas usadas. Para usar **geom_sf()**, la única estética requerida es **geometry**, en la cual debe indicarse la columna que tiene las coordenadas de cada polígono. Usualmente esta columna

tiene el nombre **geometry**, por lo cual es necesario colocar **geometry = geometry** en las estéticas:

```

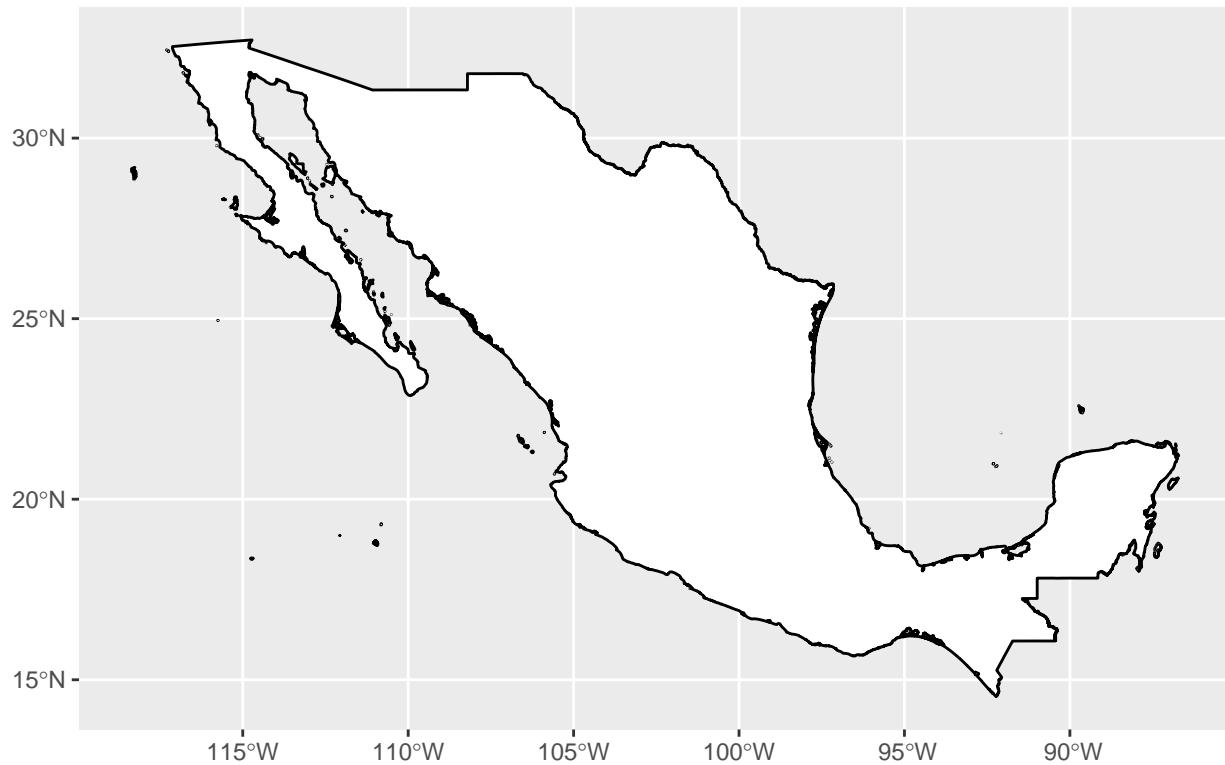
ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry))

```



Algunos atributos visuales que pueden mejorar el mapa son el color de relleno (`fill`), color de contorno (`color`) y el ancho del contorno (`size`):

```
ggplot() +  
  geom_sf(data = mapa_mexico, aes(geometry = geometry),  
          fill = "white", color = "black", size = 1/2)
```



En caso de que se desee realizar un mapa con división estatal, solo se debe de usar el conjunto de datos correspondiente:

```
mapa_estados <- st_read(dsn = "datos_manual/mapas/mexico/mapa_estados/dest20gw.shp")
## Reading layer `dest20gw' from data source
##   `/home/leot/Documents/programacion/R/introduccion_R_biotologia/datos_manual/mapas/mexico/mapa_estados'
##   using driver `ESRI Shapefile'
## Simple feature collection with 32 features and 8 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -118.3651 ymin: 14.53507 xmax: -86.71074 ymax: 32.71863
## Geodetic CRS:  WGS 84

ggplot() +
  geom_sf(data = mapa_estados, aes(geometry = geometry),
          fill = "white", color = "black", size = 1/3)
```



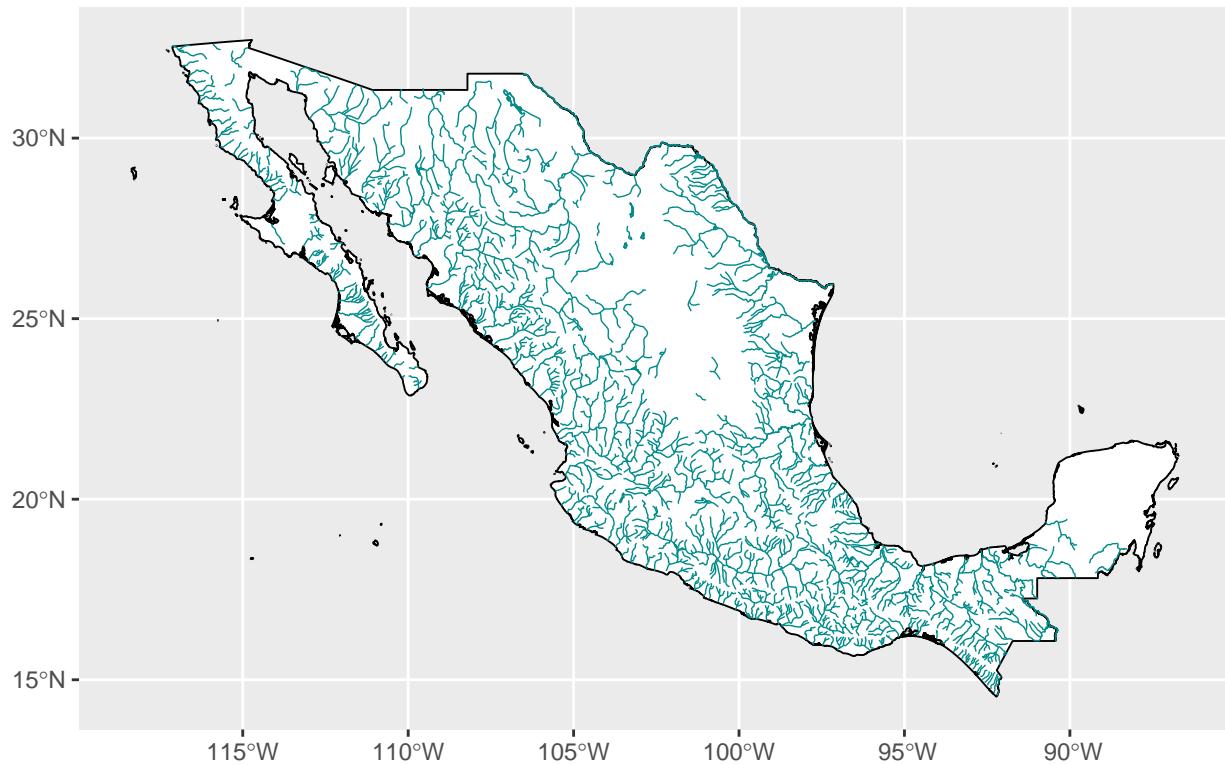
Usando múltiples capas de `geom_sf()` se pueden representar diferentes tipos de información, como es el caso de los ríos de México. La información de esta capa también puede ser obtenida del portal de la CONABIO, en las categorías Hidrología>Ríos:

```
mapa_rios <- st_read(dsn = "datos_manual/mapas/mexico/mapa_rios/hidro4mgw.shp")
## Reading layer `hidro4mgw' from data source
##   `/home/leot/Documents/programacion/R/introduccion_R_biotologia/datos_manual/mapas/mexico/mapa_rios/h
##   using driver `ESRI Shapefile'
## Simple feature collection with 2234 features and 8 fields
## Geometry type: LINESTRING
## Dimension:      XY
## Bounding box:  xmin: -117.1316 ymin: 14.56617 xmax: -88.27261 ymax: 32.60709
## Geodetic CRS:  WGS 84
```

Debe tomarse en cuenta que `Geodetic CRS` debe coincidir en todas las capas, por lo que deben seleccionarse los archivos que estén almacenados en coordenadas geográficas.

El acomodo de las capas en el código es importante, deben colocarse primero las que irán abajo y después las que irán por encima:

```
ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry),
          fill = "white", color = "black", size = 1/3) +
  geom_sf(data = mapa_rios, aes(geometry = geometry),
          color = "cyan4", size = 1/4)
```



Otra manera de combinar las capas sería para darle diferente aspecto al límite nacional y a la división entre estados:

```
ggplot() +  
  geom_sf(data = mapa_mexico, aes(geometry = geometry),  
          fill = "white", color = "black", size = 1/2) +  
  geom_sf(data = mapa_estados, aes(geometry = geometry),  
          fill = "transparent", color = "grey40", size = 1/4)
```



6.2 Datos para un mapa de distribución de especies

Los registros de presencia de una especie o un conjunto de especies son observaciones individuales que pueden ser graficadas como puntos.

Para realizar un mapa de la distribución de la familia Commelinaceae en México se puede comenzar con todos los registros de una base de datos, por ejemplo de GBIF. En el buscador de la página se introdujo el nombre de la familia y se seleccionó el primer resultado de la búsqueda. En el botón de la esquina superior derecha que muestra el total de registros se puede consultar una tabla con la información para cada observación. Allí mismo se pueden aplicar una serie de filtros antes de descargar la información, pero para ejemplificar cómo se pueden usar los métodos de este manual se usarán todas las observaciones. En la pestaña Descargar, se selecciona el formato .csv y después de unos minutos la descarga estará lista. Para usar descargar información será necesario una cuenta y para conjuntos de datos muy grandes se enviará un link al correo de la cuenta para acceder a la descarga.

Al descomprimir el archivo se obtiene toda la información en formato .csv, sin embargo, el archivo está separado por tabuladores. Debido a esto, la mejor opción es usar `read_delim()` para leer el archivo:

```
datos_commelina <- read_delim(file = "datos_manual/mapas/datos_commelina.csv")
## # Rows: 115935 Columns: 50
## -- Column specification --
## Delimiter: "\t"
## chr (32): datasetKey, occurrenceID, kingdom, phylum, class, order, family, ...
## dbl (15): gbifID, individualCount, decimalLatitude, decimalLongitude, coord...
## dttm (3): eventDate, dateIdentified, lastInterpreted
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
str(datos_commelina)
```

```

## spec_tbl_df [115,935 x 50] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
## $ gbifID : num [1:115935] 2.52e+09 2.52e+09 2.52e+09 2.52e+09 2.52e+09 ...
## $ datasetKey : chr [1:115935] "15f819bd-6612-4447-854b-14d12ee1022d" "15f819bd...
## $ occurrenceID : chr [1:115935] "https://data.biodiversitydata.nl/naturalis/spec...
## $ kingdom : chr [1:115935] "Plantae" "Plantae" "Plantae" "Plantae" ...
## $ phylum : chr [1:115935] "Tracheophyta" "Tracheophyta" "Tracheophyta" "Tr...
## $ class : chr [1:115935] "Liliopsida" "Liliopsida" "Liliopsida" "Liliops...
## $ order : chr [1:115935] "Commelinales" "Commelinales" "Commelinales" "Co...
## $ family : chr [1:115935] "Commelinaceae" "Commelinaceae" "Commelinaceae" ...
## $ genus : chr [1:115935] "Commelina" "Commelina" "Commelina" "Commelina" ...
## $ species : chr [1:115935] "Commelina albescens" "Commelina capitata" "Comm...
## $ infraspecificEpithet : chr [1:115935] NA NA "erecta" NA ...
## $ taxonRank : chr [1:115935] "SPECIES" "SPECIES" "SUBSPECIES" "SPECIES" ...
## $ scientificName : chr [1:115935] "Commelina albescens Hassk." "Commelina capitata...
## $ verbatimScientificName : chr [1:115935] "Commelina albescens Hassk." "Commelina capitata...
## $ verbatimScientificNameAuthorship : chr [1:115935] "Hassk." "Benth." NA "C.B.Clarke" ...
## $ countryCode : chr [1:115935] "ET" "CI" "GH" "GA" ...
## $ locality : chr [1:115935] "30 km W of Jijiga, on the road to Harrar near 60...
## $ stateProvince : chr [1:115935] "Harerge" "Abidjan" "Greater Accra Region" "Nyan...
## $ occurrenceStatus : chr [1:115935] "PRESENT" "PRESENT" "PRESENT" "PRESENT" ...
## $ individualCount : num [1:115935] NA NA NA NA NA NA NA NA NA ...
## $ publishingOrgKey : chr [1:115935] "396d5f30-dea9-11db-8ab4-b8a03c50a862" "396d5f30...
## $ decimalLatitude : num [1:115935] 9.28 5.33 5.65 -2.33 -1.3 ...
## $ decimalLongitude : num [1:115935] 42.633 -4.05 -0.193 11.083 36.8 ...
## $ coordinateUncertaintyInMeters : num [1:115935] NA NA NA NA NA NA NA NA NA ...
## $ coordinatePrecision : num [1:115935] NA NA NA NA NA NA NA NA NA ...
## $ elevation : num [1:115935] NA NA NA NA NA NA NA NA NA ...
## $ elevationAccuracy : num [1:115935] NA NA NA NA NA NA NA NA NA ...
## $ depth : num [1:115935] NA NA NA NA NA NA NA NA NA ...
## $ depthAccuracy : num [1:115935] NA NA NA NA NA NA NA NA NA ...
## $ eventDate : POSIXct[1:115935], format: "1972-05-10" "1976-06-10" ...
## $ day : num [1:115935] 10 10 1 25 14 7 10 19 1 31 ...
## $ month : num [1:115935] 5 6 11 11 5 8 8 10 11 10 ...
## $ year : num [1:115935] 1972 1976 1951 1986 1974 ...
## $ taxonKey : num [1:115935] 2764104 2764370 7227470 2764366 2764438 ...
## $ speciesKey : num [1:115935] 2764104 2764370 2764389 2764366 2764438 ...
## $ basisOfRecord : chr [1:115935] "PRESERVED_SPECIMEN" "PRESERVED_SPECIMEN" "PRESE...
## $ institutionCode : chr [1:115935] NA NA NA NA ...
## $ collectionCode : chr [1:115935] "Botany" "Botany" "Botany" "Botany" ...
## $ catalogNumber : chr [1:115935] "WAG.1444373" "WAG.1444689" "WAG.1495061" "WAG.1...
## $ recordNumber : chr [1:115935] "Seegeler, CJP 2248" "Koning, J de 6983" "Mo...
## $ identifiedBy : chr [1:115935] "Faden RB" "Koning J de" "Brenan JPM" "Bos JJ" ...
## $ dateIdentified : POSIXct[1:115935], format: NA NA ...
## $ license : chr [1:115935] "CC0_1_0" "CC0_1_0" "CC0_1_0" "CC0_1_0" ...
## $ rightsHolder : chr [1:115935] "Naturalis Biodiversity Center" "Naturalis Biodi...
## $ recordedBy : chr [1:115935] "Seegeler CJP" "Koning J de" "Morton JK" "Wilde ...
## $ typeStatus : chr [1:115935] NA NA NA NA ...
## $ establishmentMeans : chr [1:115935] NA NA NA NA ...
## $ lastInterpreted : POSIXct[1:115935], format: "2022-05-01 09:57:39" "2022-05-01 09...
## $ mediaType : chr [1:115935] "StillImage" "StillImage" "StillImage" "StillImage" ...
## $ issue : chr [1:115935] "INSTITUTION_MATCH_NONE" "INSTITUTION_MATCH_NONE"
## - attr(*, "spec")=
## .. cols(
## ..   gbifID = col_double(),

```

```

## .. datasetKey = col_character(),
## .. occurrenceID = col_character(),
## .. kingdom = col_character(),
## .. phylum = col_character(),
## .. class = col_character(),
## .. order = col_character(),
## .. family = col_character(),
## .. genus = col_character(),
## .. species = col_character(),
## .. infraspecificEpithet = col_character(),
## .. taxonRank = col_character(),
## .. scientificName = col_character(),
## .. verbatimScientificName = col_character(),
## .. verbatimScientificNameAuthorship = col_character(),
## .. countryCode = col_character(),
## .. locality = col_character(),
## .. stateProvince = col_character(),
## .. occurrenceStatus = col_character(),
## .. individualCount = col_double(),
## .. publishingOrgKey = col_character(),
## .. decimalLatitude = col_double(),
## .. decimalLongitude = col_double(),
## .. coordinateUncertaintyInMeters = col_double(),
## .. coordinatePrecision = col_double(),
## .. elevation = col_double(),
## .. elevationAccuracy = col_double(),
## .. depth = col_double(),
## .. depthAccuracy = col_double(),
## .. eventDate = col_datetime(format = ""),
## .. day = col_double(),
## .. month = col_double(),
## .. year = col_double(),
## .. taxonKey = col_double(),
## .. speciesKey = col_double(),
## .. basisOfRecord = col_character(),
## .. institutionCode = col_character(),
## .. collectionCode = col_character(),
## .. catalogNumber = col_character(),
## .. recordNumber = col_character(),
## .. identifiedBy = col_character(),
## .. dateIdentified = col_datetime(format = ""),
## .. license = col_character(),
## .. rightsHolder = col_character(),
## .. recordedBy = col_character(),
## .. typeStatus = col_character(),
## .. establishmentMeans = col_character(),
## .. lastInterpreted = col_datetime(format = ""),
## .. mediaType = col_character(),
## .. issue = col_character()
## .. )
## - attr(*, "problems")=<externalptr>

```

Como se puede ver, la estructura del archivo es muy complicada y cuenta con una gran cantidad de variables:

```
colnames(datos_commelina)
```

```

## [1] "gbifID"                               "datasetKey"
## [3] "occurrenceID"                          "kingdom"
## [5] "phylum"                                "class"
## [7] "order"                                 "family"
## [9] "genus"                                 "species"
## [11] "infraspecificEpithet"                  "taxonRank"
## [13] "scientificName"                        "verbatimScientificName"
## [15] "verbatimScientificNameAuthorship"       "countryCode"
## [17] "locality"                             "stateProvince"
## [19] "occurrenceStatus"                     "individualCount"
## [21] "publishingOrgKey"                     "decimalLatitude"
## [23] "decimalLongitude"                     "coordinateUncertaintyInMeters"
## [25] "coordinatePrecision"                  "elevation"
## [27] "elevationAccuracy"                   "depth"
## [29] "depthAccuracy"                       "eventDate"
## [31] "day"                                  "month"
## [33] "year"                                 "taxonKey"
## [35] "speciesKey"                           "basisOfRecord"
## [37] "institutionCode"                      "collectionCode"
## [39] "catalogNumber"                         "recordNumber"
## [41] "identifiedBy"                         "dateIdentified"
## [43] "license"                              "rightsHolder"
## [45] "recordedBy"                           "typeStatus"
## [47] "establishmentMeans"                  "lastInterpreted"
## [49] "mediaType"                            "issue"

```

Debido a que la única información que se va a emplear tiene que ver con la información taxonómica, el país de registro y las coordenadas geográficas, se puede usar el argumento `col_select` para indicar únicamente aquellas columnas que de desean obtener de todo el archivo:

```

datos_commelina <- read_delim(file = "datos_manual/mapas/datos_commelina.csv",
                                col_select = c(family, genus, species,
                                              countryCode,
                                              decimalLatitude,
                                              decimalLongitude))

## Rows: 115935 Columns: 6
## -- Column specification -----
## Delimiter: "\t"
## chr (4): family, genus, species, countryCode
## dbl (2): decimalLatitude, decimalLongitude
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

colnames(datos_commelina)

## [1] "family"          "genus"           "species"         "countryCode"
## [5] "decimalLatitude" "decimalLongitude"

```

En la columna `countryCode` se encuentra el código de dos letras de cada país para el cual fue registrada cada observación:

```

unique(datos_commelina$countryCode)

## [1] "ET"  "CI"  "GH"  "GA"  "KE"  "ZZ"  "TH"  "US"  "CG"  NA    "ZM"  "BJ"  "ZA"  "MX"  "IN"
## [16] "PK"  "ID"  "ZW"  "TZ"  "CD"  "GW"  "GY"  "SR"  "PG"  "CM"  "BI"  "PH"  "CN"  "BO"  "PE"
## [31] "RW"  "PY"  "MG"  "SN"  "JP"  "NL"  "NG"  "AU"  "BF"  "BR"  "SB"  "PR"  "HN"  "CU"  "LR"

```

```

## [46] "AR" "TT" "MY" "GN" "DO" "AW" "GT" "CW" "SA" "KH" "MW" "SV" "MF" "TG" "BQ"
## [61] "SL" "VN" "GF" "GQ" "CR" "CH" "KM" "DM" "VE" "RE" "FR" "SG" "JM" "HT" "BS"
## [76] "PA" "CF" "UG" "AS" "ES" "CO" "NI" "LK" "MM" "PF" "VG" "EC" "MA" "BZ" "MZ"
## [91] "SS" "ML" "PW" "NP" "VI" "FI" "MS" "MQ" "GU" "UY" "CA" "GD" "YE" "SO" "TW"
## [106] "IT" "IL" "BW" "BL" "RU" "KR" "NE" "TC" "TF" "AO" "SC" "BD" "SD" "GM" "SK"
## [121] "VC" "WS" "DE" "YT" "HK" "AT" "CZ" "SE" "CL" "BN" "TL" "ER" "GE" "BM" "FM"
## [136] "TD" "MU" "NC" "KN" "VU" "AG" "SZ" "BT" "KY" "GP" "PT" "KP" "RS" "BE" "NO"
## [151] "OM" "AL" "AF" "LS" "MP" "SI" "HR" "RO" "HU" "BY" "UA" "LA" "GR" "CK" "NZ"
## [166] "GB" "TR" "LC" "EE" "ME" "AI" "TO" "MR" "FJ" "UM" "PN" "BB" "CV" "ST" "BG"
## [181] "MK" "NF" "PL" "SH" "SX" "CX" "MV" "IO" "EG" "NU" "IR" "DJ" "WF"

```

Usando esta información se pueden filtrar los datos que corresponden a México:

```

commelina_mexico <- datos_commelina %>%
  filter(countryCode == "MX")

commelina_mexico

## # A tibble: 12,396 x 6
##   family     genus   species      countryCode decimalLatitude decimalLongitude
##   <chr>      <chr>    <chr>        <chr>            <dbl>           <dbl>
## 1 Commelinaceae Commelina Commelina virginica MX                 NA             NA
## 2 Commelinaceae Commelina Commelina virginica MX                 NA             NA
## 3 Commelinaceae Commelina Commelina coelestis MX                NA             NA
## 4 Commelinaceae Commelina Commelina erecta   MX              21.9            -106.
## 5 Commelinaceae Commelina Commelina erecta   MX              21.9            -106.
## 6 Commelinaceae Commelina Commelina erecta   MX              21.9            -106.
## 7 Commelinaceae Commelina Commelina erecta   MX              21.9            -106.
## 8 Commelinaceae Commelina Commelina erecta   MX              21.9            -106.
## 9 Commelinaceae Commelina Commelina erecta   MX              21.9            -106.
## 10 Commelinaceae Commelina Commelina erecta  MX              21.9            -106.
## # ... with 12,386 more rows, and abbreviated variable names 1: decimalLatitude,
## #   2: decimalLongitude

```

Este conjunto se usará posteriormente, cuando se realice un mapa más complejo. De momento, también podemos filtrar los registros de la especie *Commelina coelestis*:

```

unique(commelina_mexico$species)

## [1] "Commelina virginica"          "Commelina coelestis"
## [3] "Commelina erecta"            "Commelina diffusa"
## [5] "Commelina graminifolia"       "Commelina pallida"
## [7] NA                            "Commelina dianthifolia"
## [9] "Commelina elliptica"          "Commelina scabra"
## [11] "Commelina tuberosa"          "Commelina leiocarpa"
## [13] "Commelina standleyi"         "Commelina communis"
## [15] "Commelina auriculata"        "Commelina benghalensis"
## [17] "Commelina obliqua"           "Commelina socorrogonzaleziae"
## [19] "Commelina angustifolia"       "Commelina rufipes"
## [21] "Commelina bambusifolia"       "Commelina japonica"
## [23] "Commelina texcocana"          "Commelina ramosissima"
## [25] "Commelina jaliscana"          "Commelina villosa"
## [27] "Commelina bravae"             "Commelina bambusifoloides"
## [29] "Commelina queretarensis"      "Commelina congestipantha"
## [31] "Commelina nivea"              "Commelina rzedowskii"
## [33] "Commelina persicariifolia"     "Commelina geniculata"
## [35] "Commelina orchioides"          "Commelina caroliniana"

```

```
## [37] "Commelina commelinoides"
```

Partiendo del conjunto de datos original, el nuevo filtro puede colocarse junto con aquel que incluye únicamente los datos de México. Alternativamente, este los datos de *C. coelestis* pueden ser separados a partir de `commelina_mexico`. De cualquier modo, es necesario remover todos los datos ausentes (`NA`), lo cual ahorra memoria ya que se genera un objeto con menos observaciones:

```
c_coelestis <- datos_commelina %>%
  filter(countryCode == "MX",
        species == "Commelina coelestis",
        is.na(decimalLatitude) == FALSE)

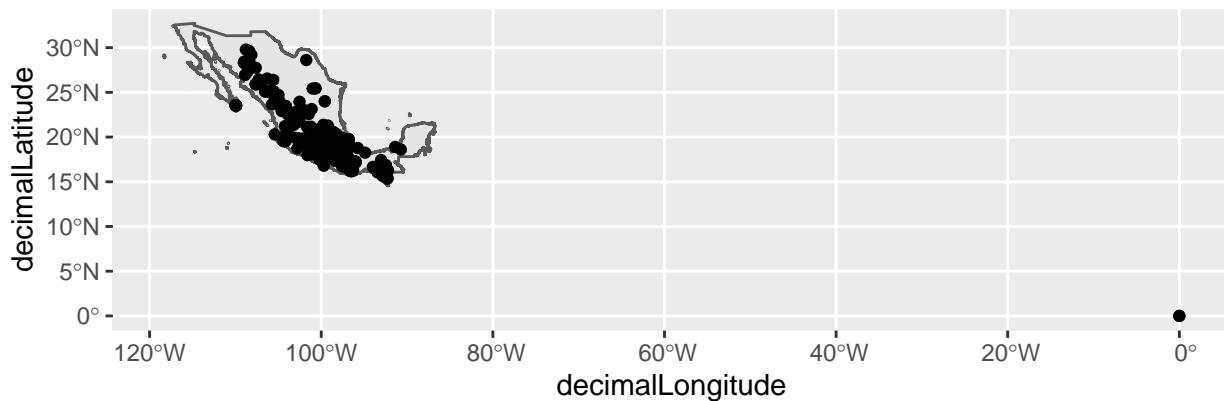
c_coelestis

## # A tibble: 627 x 6
##   family     genus   species      countryCode decimalLatitude decimalLongitude
##   <chr>      <chr>    <chr>        <chr>          <dbl>            <dbl>
## 1 Commelinaceae Commelina Commelina coelestis MX             16.8           -92.7
## 2 Commelinaceae Commelina Commelina coelestis MX             25.1           -106.
## 3 Commelinaceae Commelina Commelina coelestis MX             28.5           -108.
## 4 Commelinaceae Commelina Commelina coelestis MX             17.9           -97.0
## 5 Commelinaceae Commelina Commelina coelestis MX             17.8           -97.6
## 6 Commelinaceae Commelina Commelina coelestis MX             19.6           -99.1
## 7 Commelinaceae Commelina Commelina coelestis MX             19.0           -100.
## 8 Commelinaceae Commelina Commelina coelestis MX             18.1           -101.
## 9 Commelinaceae Commelina Commelina coelestis MX             17.6           -97.2
## 10 Commelinaceae Commelina Commelina coelestis MX            17.1           -96.2
## # ... with 617 more rows, and abbreviated variable names 1: decimalLatitude,
## #   2: decimalLongitude
```

El filtro para datos ausentes de latitud también elimina los `NA` de longitud debido a que no pude haber coordenadas con solo latitud o longitud.

Con el data frame listo, se puede agregar una nueva capa al mapa `geom_point()`, cuyas valores de `x` y `y` toman los datos de longitud y latitud, respectivamente:

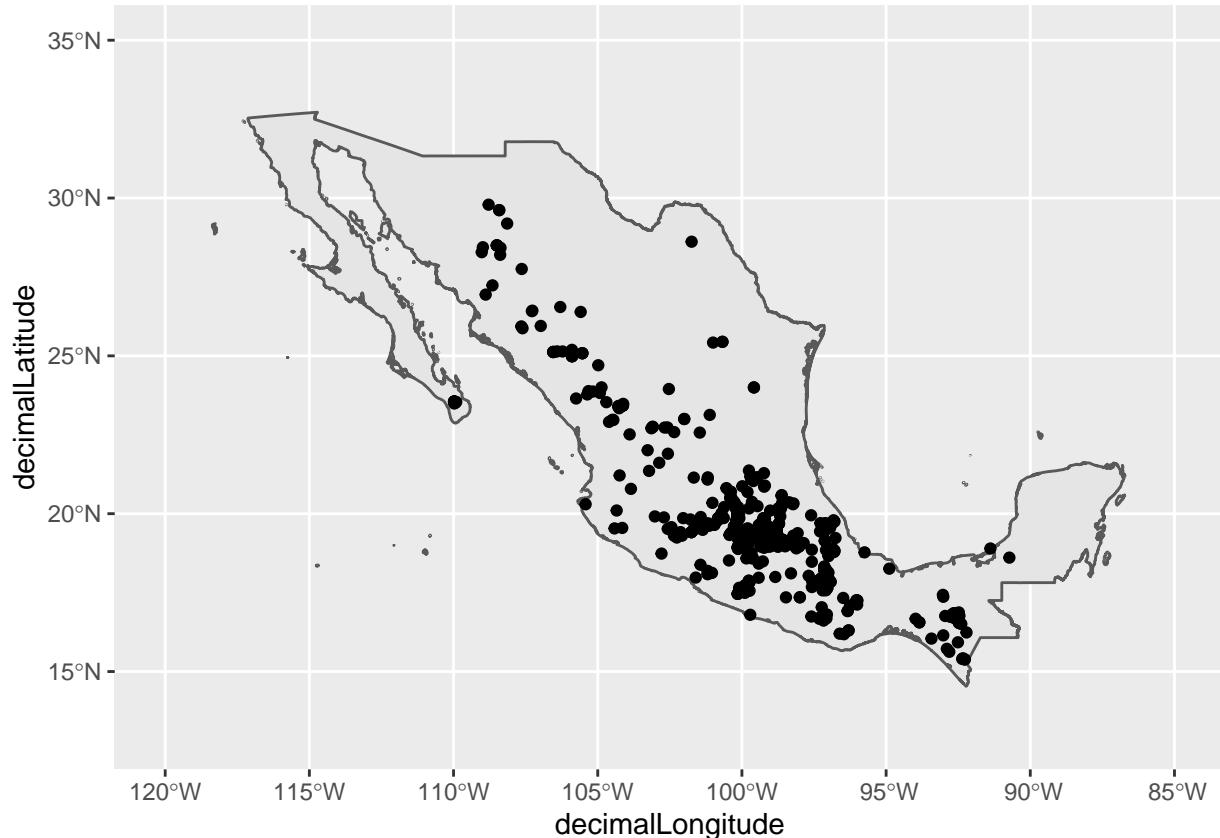
```
ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry)) +
  geom_point(data = c_coelestis, aes(x = decimalLongitude,
                                    y = decimalLatitude))
```



Aquí se observa que un dato erróneo ubicado fuera de México ocasiona la distorsión en el mapa resultante. Ese dato podría removese manualmente, pero una manera más sencilla de solucionar el problema consiste establecer límites con `lims()` para restringir el mapa únicamente a las coordenadas que encuadren a México.

Los argumentos `x` y `y` requieren un vector con el límite inferior y superior deseado:

```
ggplot() +  
  geom_sf(data = mapa_mexico, aes(geometry = geometry)) +  
  geom_point(data = c_coelestis, aes(x = decimalLongitude,  
                                     y = decimalLatitude)) +  
  lims(x = c(-120, -85), y = c(13, 35))  
  
## Warning: Removed 2 rows containing missing values (geom_point).
```



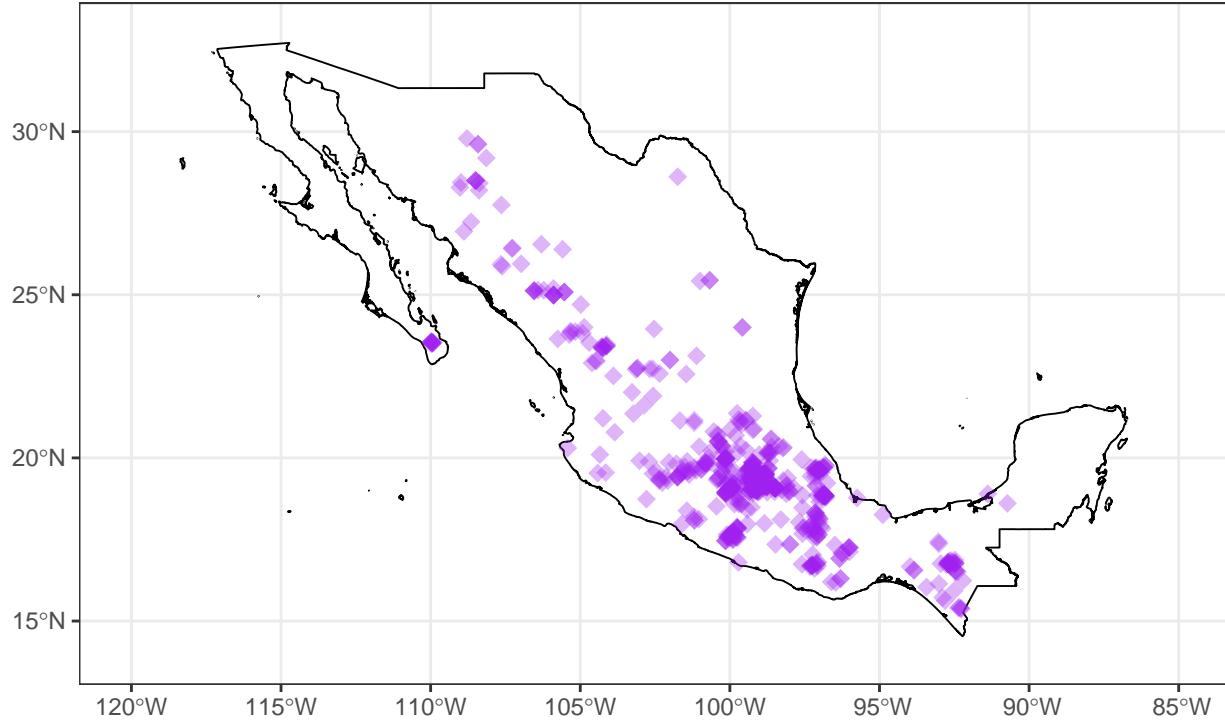
Usando `lims` no se eliminan los datos que no se desean ver, únicamente se limita el espacio de la gráfica para excluir aquellos datos que sabemos que están fuera de México. Para terminar el mapa se puede cambiar el color y carácter de cada observación. De igual manera se cambia la transparencia para mostrar aquellas zonas en las que se concentran la mayor cantidad de registros:

```
ggplot() +  
  geom_sf(data = mapa_mexico, aes(geometry = geometry),  
          fill = "white", color = "black", size = 1/3) +  
  geom_point(data = c_coelestis, aes(x = decimalLongitude,  
                                     y = decimalLatitude),  
             alpha = 1/3, size = 3, pch = 18,  
             color = "purple") +  
  lims(x = c(-120, -85), y = c(14, 33)) +  
  labs(  
    x = NULL,  
    y = NULL  
) +  
  theme(panel.background = element_rect(fill = "grey95"),  
        panel.grid = element_line(linetype = 3,
```

```

        color = "grey60")) +
theme_bw()
## Warning: Removed 2 rows containing missing values (geom_point).

```



6.3 Mapa de distribución para más de una especie

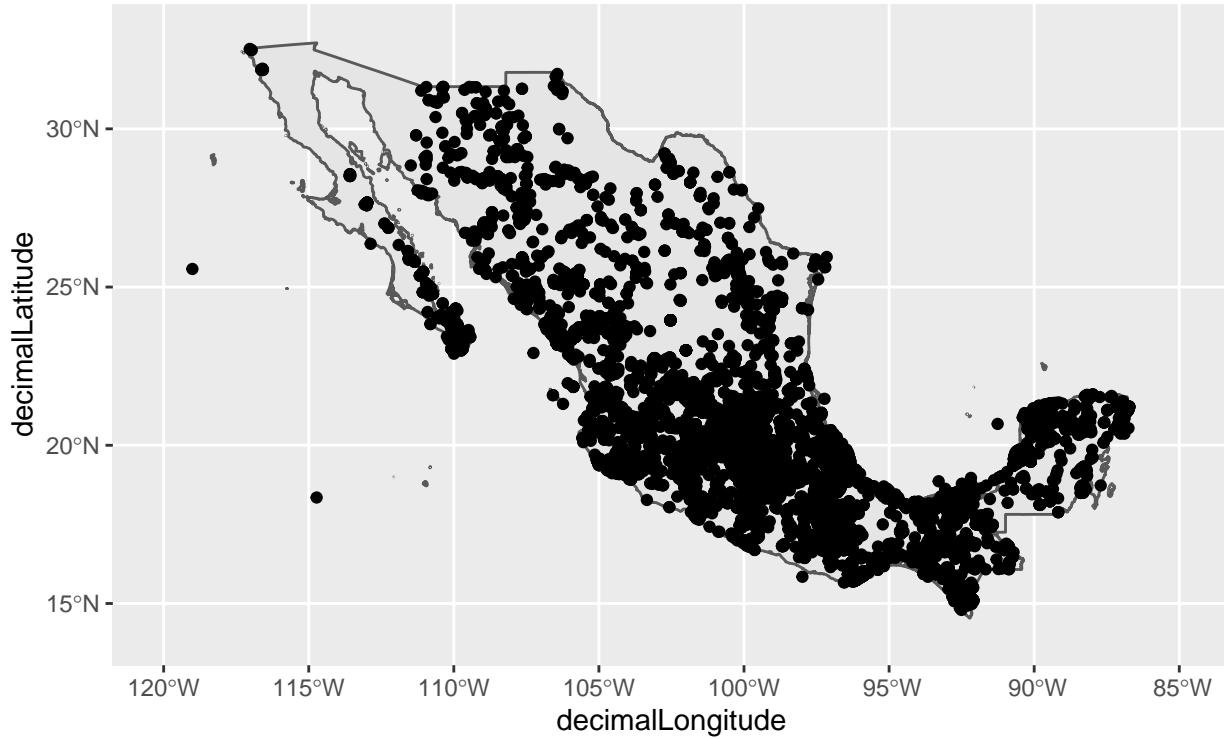
Cuando se desea representar los registros de más de una especie pueden surgir algunos problemas. En este caso se usarán todas las especies del conjunto de datos para mostrar diversas soluciones.

Usando el objeto `commelina_mexico`, nos encontramos con el mismo problema relacionado con los datos fuera de los límites de México:

```

ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry)) +
  geom_point(data = commelina_mexico, aes(x = decimalLongitude,
                                             y = decimalLatitude)) +
  lims(x = c(-120, -85), y = c(14, 33))
## Warning: Removed 3978 rows containing missing values (geom_point).

```



Remover cada punto manualmente no es viable. Una mejor solución consiste en seleccionar qué datos se encuentran dentro del área de los polígonos que forman la capa `geom_sf()`, es decir, aquellos puntos que se intersecan con los polígonos del objeto `mapa_mexico`. Para comparar este conjunto con los datos de distribución se primero transformar el data frame `commelina_mexico` a un objeto tipo `sf`:

```
class(commelina_mexico)
## [1] "tbl_df"     "tbl"        "data.frame"
```

Ello se realiza con la función `st_as_sf()`. Primero se deben filtrar las observaciones para que no haya datos ausentes (NA) en las coordenadas, si es que no se había hecho antes. Para facilitar el manejo de los datos en pasos posteriores también se remueven los datos ausentes de la columna `species`:

```
commelina_mexico <- commelina_mexico %>%
  filter(is.na(decimalLongitude) == F,
         is.na(species) == F)
```

Los datos filtrados deben convertirse a un objeto sf, especificando el argumento el mismo CRS que en el otro objeto con el cual se realizará la intersección, en este caso "WSG84":

```
head(mapa_mexico)
## Simple feature collection with 6 features and 4 fields
## Geometry type: POLYGON
## Dimension: XY
## Bounding box: xmin: -117.301 ymin: 14.53401 xmax: -86.74039 ymax: 32.71877
## Geodetic CRS: WGS 84
##   AREA  PERIMETER COV_ COV_ID           geometry
## 1 1.731249e+02 185.03666278    2    404 POLYGON ((-114.7575 32.7162...
## 2 4.694670e-05  0.03188133    3      2 POLYGON ((-117.2939 32.4342...
## 3 1.770398e-05  0.01756151    4      3 POLYGON ((-117.2585 32.4173...
## 4 1.608039e-04  0.06796011    5      4 POLYGON ((-117.2393 32.3949...
## 5 1.271809e-02  0.51603622    6      8 POLYGON ((-114.7698 31.7067...
## 6 3.505660e-05  0.02479165    7      6 POLYGON ((-116.8074 31.8138...
```

En el argumento `coords` deben colocarse los nombres columnas en donde se encuentran las coordenadas (latitud y longitud). Deben usarse comillas debido a que esta función no forma parte de `tidyverse`, en donde los nombres de las columnas para el objeto seleccionado pueden escribirse sin ellas:

```
commelina_mexico_sf <- st_as_sf(commelina_mexico,
  coords = c("decimalLongitude", "decimalLatitude"),
  crs = "WGS84")
```

```
class(commelina_mexico_sf)
## [1] "sf"           "tbl_df"        "tbl"           "data.frame"
```

Este nuevo objeto tiene una columna `geometry` que sustituye a las coordenadas:

```
head(commelina_mexico_sf)

## Simple feature collection with 6 features and 4 fields
## Geometry type: POINT
## Dimension: XY
## Bounding box: xmin: -105.8864 ymin: 21.85056 xmax: -105.8822 ymax: 21.85278
## Geodetic CRS: WGS 84
## # A tibble: 6 x 5
##   family     genus   species   countryCode      geometry
##   <chr>      <chr>    <chr>     <chr>          <POINT [°]>
## 1 Commelinaceae Commelina Commelina erecta MX (-105.8828 21.85056)
## 2 Commelinaceae Commelina Commelina erecta MX (-105.8831 21.85111)
## 3 Commelinaceae Commelina Commelina erecta MX (-105.8864 21.85278)
## 4 Commelinaceae Commelina Commelina erecta MX (-105.8822 21.85083)
## 5 Commelinaceae Commelina Commelina erecta MX (-105.8825 21.85083)
## 6 Commelinaceae Commelina Commelina erecta MX (-105.8833 21.85167)
```

Con ambos objetos de tipo `sf` se puede realizar la intersección usando la función `st_intersection()`, colocando en `x` los datos de presencia y en `y` el mapa de México. Es necesario guardar el resultado de la función como un nuevo objeto, el cual contendrá únicamente los registros dentro del mapa. Antes de mandar el código para crear el objeto debe tenerse en cuenta que operación tomará algunos minutos:

```
inter_commelina <- st_intersection(x = commelina_mexico_sf,
  y = mapa_mexico)

## Warning: attribute variables are assumed to be spatially constant throughout all
## geometries

inter_commelina

## Simple feature collection with 7991 features and 8 fields
## Geometry type: POINT
## Dimension: XY
## Bounding box: xmin: -117.031 ymin: 14.79667 xmax: -86.7157 ymax: 32.52555
## Geodetic CRS: WGS 84
## # A tibble: 7,991 x 9
##   family     genus   species   count~1  AREA PERIM~2 COV_ COV_ID
##   * <chr>      <chr>    <chr>     <chr>    <dbl>  <dbl> <dbl> <dbl>
## 1 Commelinaceae Commelina Commelina erecta MX    173.   185.    2   404
## 2 Commelinaceae Commelina Commelina diffusa MX    173.   185.    2   404
## 3 Commelinaceae Commelina Commelina erecta MX    173.   185.    2   404
## 4 Commelinaceae Commelina Commelina dianthi~ MX    173.   185.    2   404
## 5 Commelinaceae Commelina Commelina diffusa MX    173.   185.    2   404
## 6 Commelinaceae Commelina Commelina diffusa MX    173.   185.    2   404
```

```

## 7 Commelinaceae Commelina Commelina erecta MX 173. 185. 2 404
## 8 Commelinaceae Commelina Commelina erecta MX 173. 185. 2 404
## 9 Commelinaceae Commelina Commelina erecta MX 173. 185. 2 404
## 10 Commelinaceae Commelina Commelina diffusa MX 173. 185. 2 404
## # ... with 7,981 more rows, 1 more variable: geometry <POINT [°]>, and
## #   abbreviated variable names 1: countryCode, 2: PERIMETER

```

Como resultado, se obtiene un nuevo objeto `sf` que preserva toda la información relacionada con cada una de las observaciones, en este caso la variable más importante es la especie:

```

head(inter_commelina)

## Simple feature collection with 6 features and 8 fields
## Geometry type: POINT
## Dimension: XY
## Bounding box: xmin: -108.208 ymin: 16.5686 xmax: -94.9519 ymax: 28.1633
## Geodetic CRS: WGS 84
## # A tibble: 6 x 9
##   family     genus   species      count~1  AREA PERIM~2 COV_ COV_ID
##   <chr>      <chr>   <chr>       <chr>    <dbl>  <dbl> <dbl> <dbl>
## 1 Commelinaceae Commelina Commelina erecta MX 173. 185. 2 404
## 2 Commelinaceae Commelina Commelina diffusa MX 173. 185. 2 404
## 3 Commelinaceae Commelina Commelina erecta MX 173. 185. 2 404
## 4 Commelinaceae Commelina Commelina dianthif~ MX 173. 185. 2 404
## 5 Commelinaceae Commelina Commelina diffusa MX 173. 185. 2 404
## 6 Commelinaceae Commelina Commelina diffusa MX 173. 185. 2 404
## # ... with 1 more variable: geometry <POINT [°]>, and abbreviated variable
## #   names 1: countryCode, 2: PERIMETER

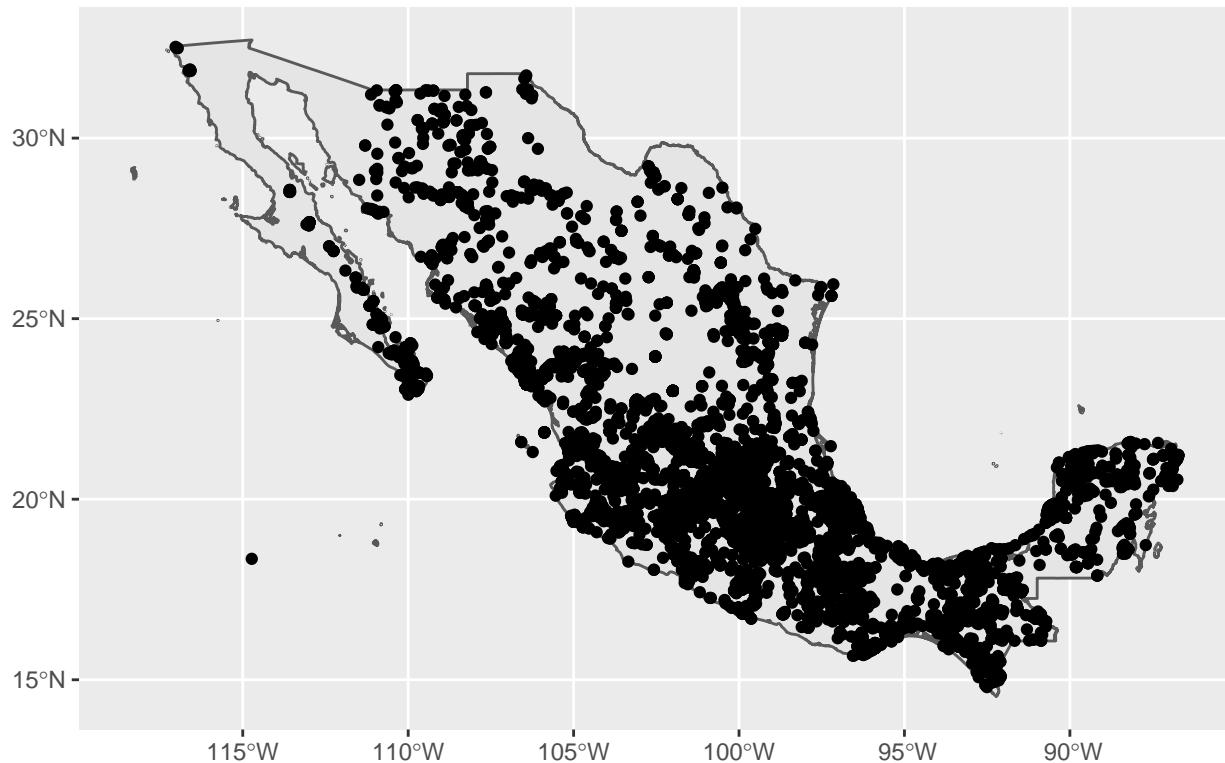
```

`inter_commelina` también tiene su propia columna `geometry`, la cual se usará para el mapa usando `geom_sf()` en lugar de `geom_point()`:

```

ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry)) +
  geom_sf(data = inter_commelina, aes(geometry = geometry))

```

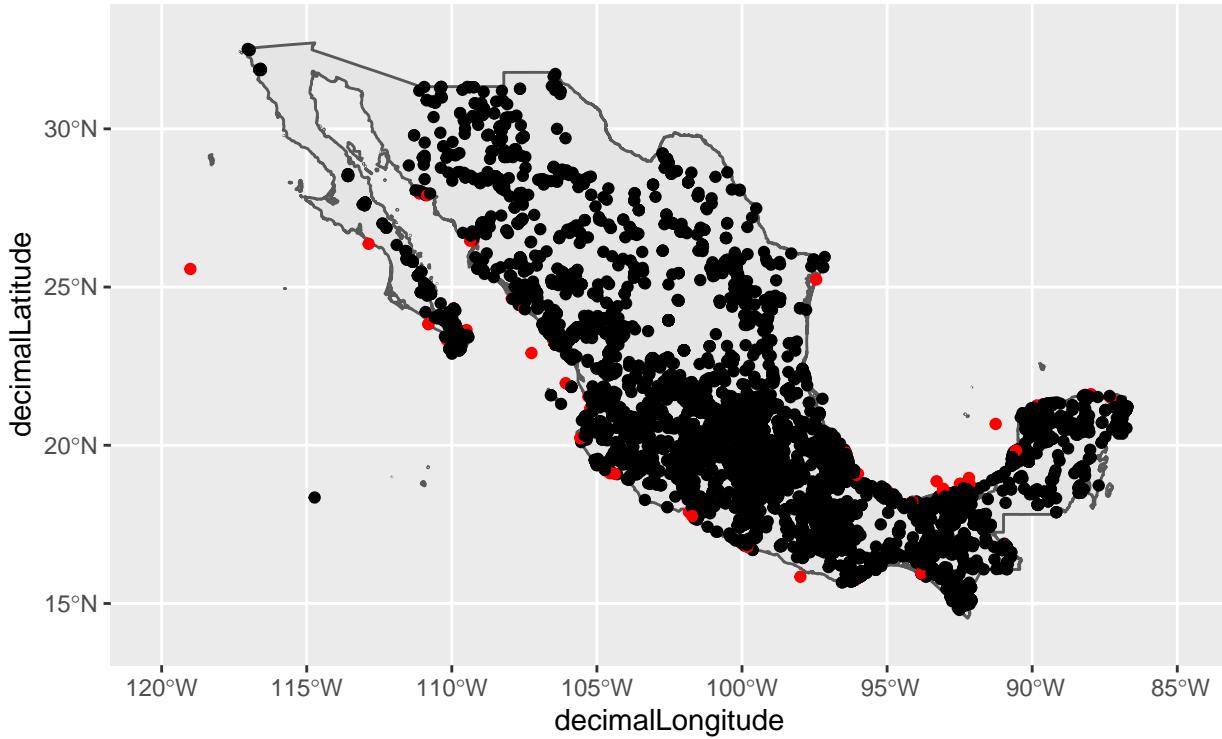


Son relativamente pocas las observaciones que fueron removidas, en el siguiente mapa los puntos rojos indican las observaciones que están fuera de la intersección:

```
outside <- sapply(st_intersects(x = commelina_mexico_sf,
                                 y = mapa_mexico),
                     function(x){length(x) == 0})

commelina_mexico_rm <- cbind(commelina_mexico, outside)

ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry)) +
  geom_point(data = commelina_mexico_rm, aes(x = decimalLongitude,
                                              y = decimalLatitude,
                                              color = outside)) +
  lims(x = c(-120, -85), y = c(14, 33)) +
  theme(legend.position = "none") +
  scale_color_manual(values = c("black", "red"))
```



Este mismo flujo de trabajo puede usarse para seleccionar y graficar los datos de un solo estado. En el mapa previamente cargado `mapa_estados`, la columna `NOM_ENT` contiene la información sobre el estado al que pertenece cada polígono:

```
str(mapa_estados)

## Classes 'sf' and 'data.frame': 32 obs. of 9 variables:
## $ CVE_ENT : chr "01" "22" "23" "24" ...
## $ NOM_ENT : chr "Aguascalientes" "Querétaro" "Quintana Roo" "San Luis Potosí" ...
## $ CVE_CAP : num 10001 140001 40001 280001 60001 ...
## $ NOM_CAP : chr "Aguascalientes" "Santiago de Querétaro" "Chetumal" "San Luis Potosí" ...
## $ COV_ : num 0 21 22 23 24 25 26 27 28 29 ...
## $ COV_ID : num 1 22 23 24 25 26 27 28 29 30 ...
## $ AREA : num 555867 1158927 4312072 6049996 5497433 ...
## $ PERIMETER: num 423 874 3290 2154 4637 ...
## $ geometry :sfc_MULTIPOINT of length 32; first list element: List of 1
## ..$ :List of 1
## ... .$. : num [1:2778, 1:2] -102 -102 -102 -102 -102 ...
## ...- attr(*, "class")= chr [1:3] "XY" "MULTIPOINT" "sfg"
## - attr(*, "sf_column")= chr "geometry"
## - attr(*, "agr")= Factor w/ 3 levels "constant","aggregate",...: NA NA NA NA NA NA NA NA
## ..- attr(*, "names")= chr [1:8] "CVE_ENT" "NOM_ENT" "CVE_CAP" "NOM_CAP" ...

unique(mapa_estados$NOM_ENT)

## [1] "Aguascalientes"           "Querétaro"
## [3] "Quintana Roo"             "San Luis Potosí"
## [5] "Sinaloa"                  "Sonora"
## [7] "Tabasco"                  "Tamaulipas"
## [9] "Tlaxcala"                 "Veracruz de Ignacio de la Llave"
## [11] "Yucatán"                  "Zacatecas"
## [13] "México"                   "Michoacán de Ocampo"
```

```

## [15] "Morelos"                      "Nayarit"
## [17] "Nuevo León"                   "Oaxaca"
## [19] "Puebla"                       "Guerrero"
## [21] "Hidalgo"                      "Jalisco"
## [23] "Guanajuato"                  "Durango"
## [25] "Ciudad de México"              "Chihuahua"
## [27] "Chiapas"                      "Colima"
## [29] "Coahuila de Zaragoza"          "Campeche"
## [31] "Baja California Sur"           "Baja California"

```

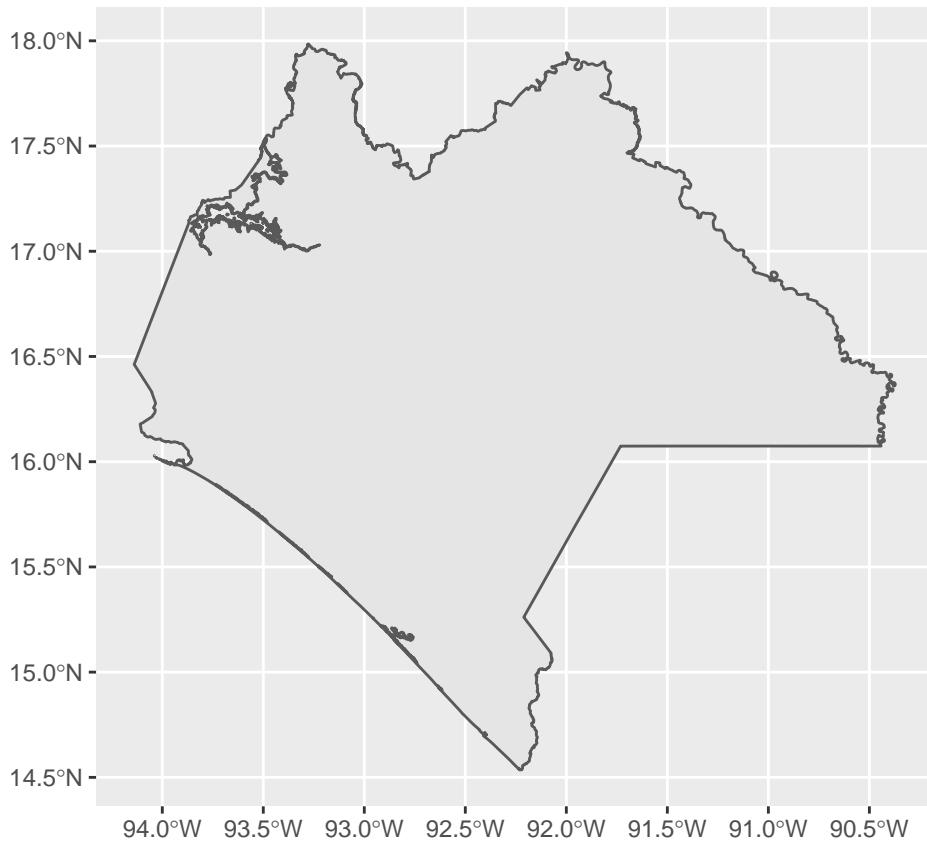
Usando esta variable se pueden seleccionar aquellos polígonos del estado de Chiapas:

```

mapa_chiapas <- mapa_estados %>%
  filter(NOM_ENT == "Chiapas")

ggplot() +
  geom_sf(data = mapa_chiapas, aes(geometry = geometry))

```

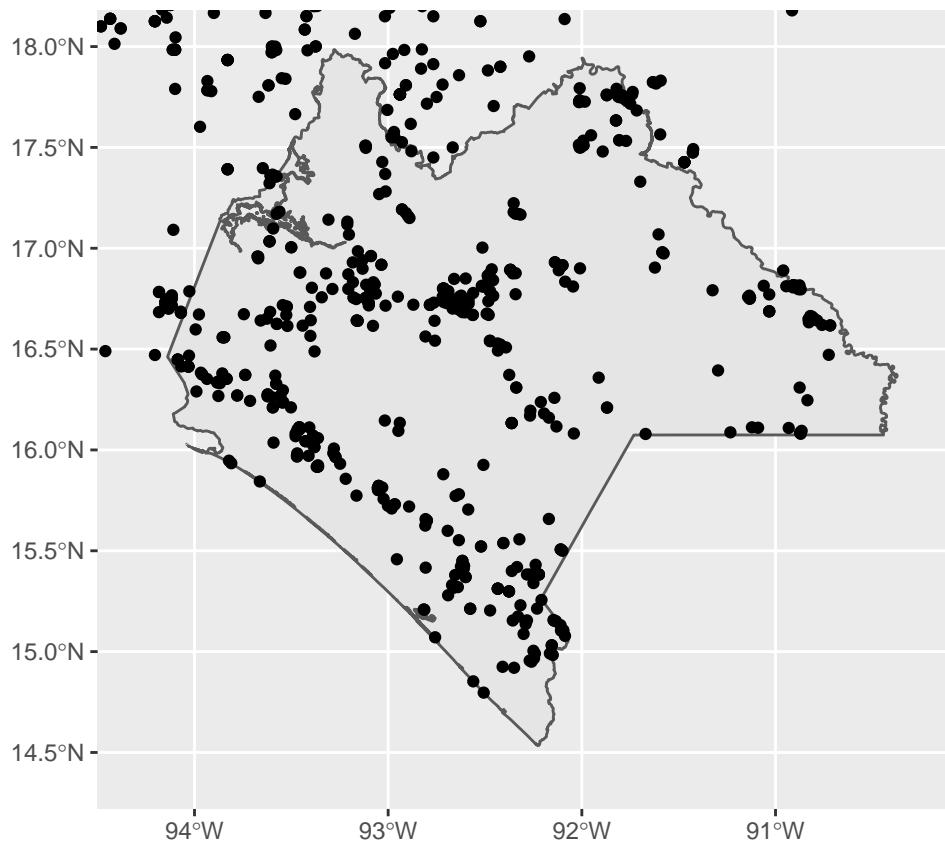


Nuevamente encontramos el mismo problema, pero ahora son muchas más las observaciones que están por fuera del área del estado:

```

ggplot() +
  geom_sf(data = mapa_chiapas, aes(geometry = geometry)) +
  geom_sf(data = inter_commelina, aes(geometry = geometry)) +
  lims(x = c(-94.3, -90.3), y = c(14.4, 18))

```



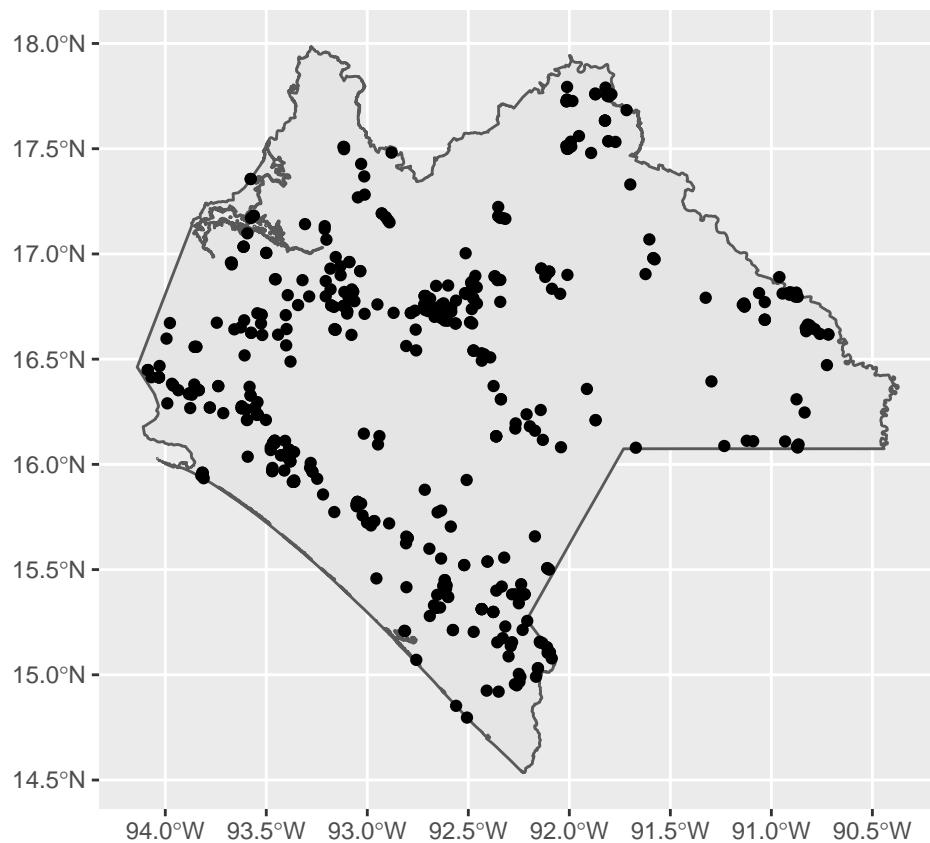
Debido a que ya se cuenta con el objeto `commelina_mexico_sf`, únicamente se realiza la intersección que filtra los datos dentro del polígono. Al ser un área más pequeña, esta operación no debería tardar demasiado:

```
commelina_chiapas <- st_intersection(x = commelina_mexico_sf,
                                         y = mapa_chiapas)

## Warning: attribute variables are assumed to be spatially constant throughout all
## geometries
```

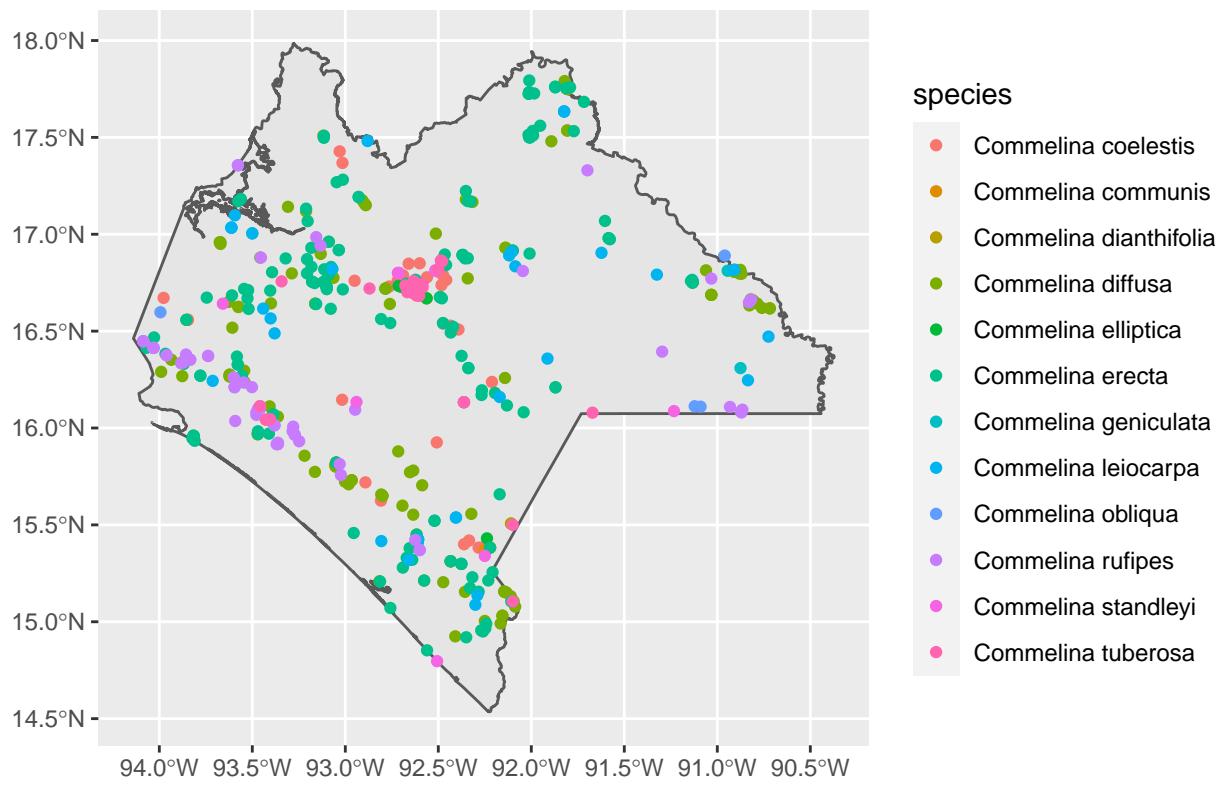
Ahora se puede usar este nuevo conjunto para realizar el mapa:

```
ggplot() +
  geom_sf(data = mapa_chiapas, aes(geometry = geometry)) +
  geom_sf(data = commelina_chiapas, aes(geometry = geometry))
```



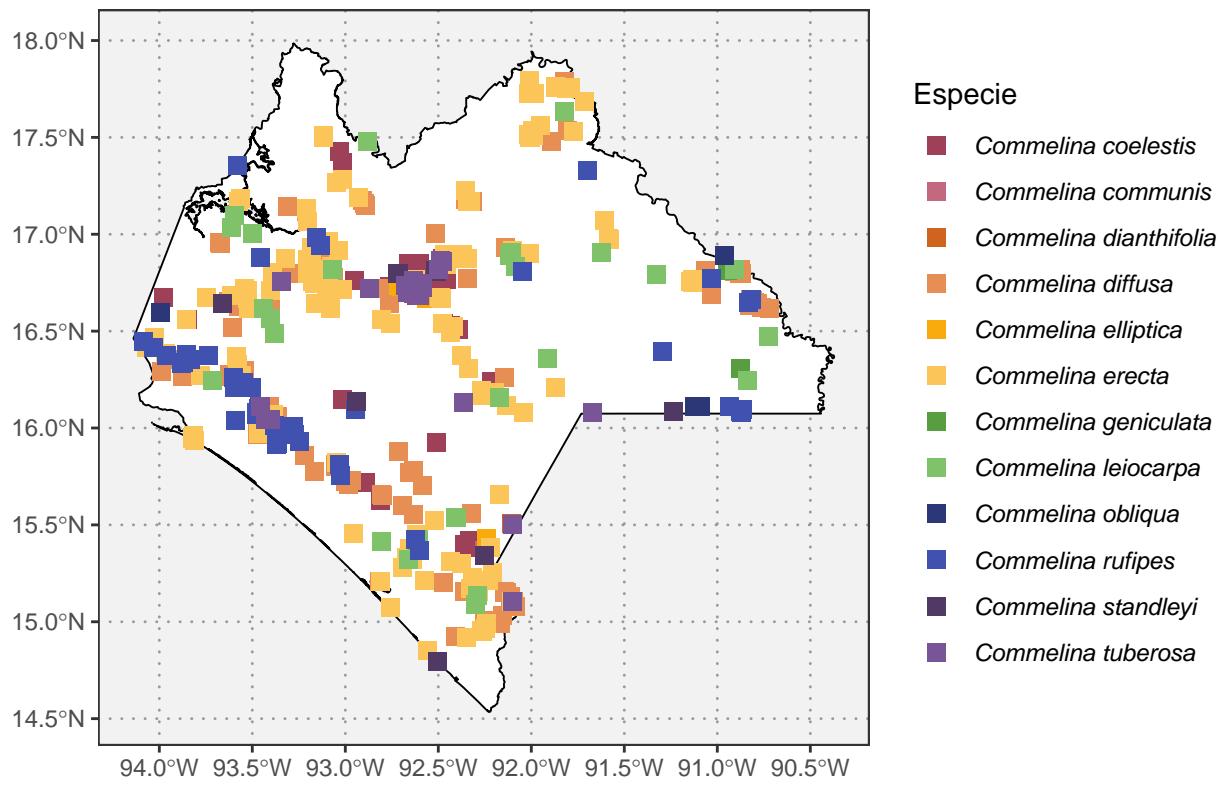
Como se mencionó antes, el objeto `sf` que resulta de la intersección cuenta con la información sobre la especie a la que corresponde cada observación. Es por esto que se puede usar `aes()` para obtener la distribución de cada especie:

```
ggplot() +
  geom_sf(data = mapa_chiapas, aes(geometry = geometry)) +
  geom_sf(data = commelina_chiapas, aes(geometry = geometry,
                                         color = species))
```



El mapa terminado:

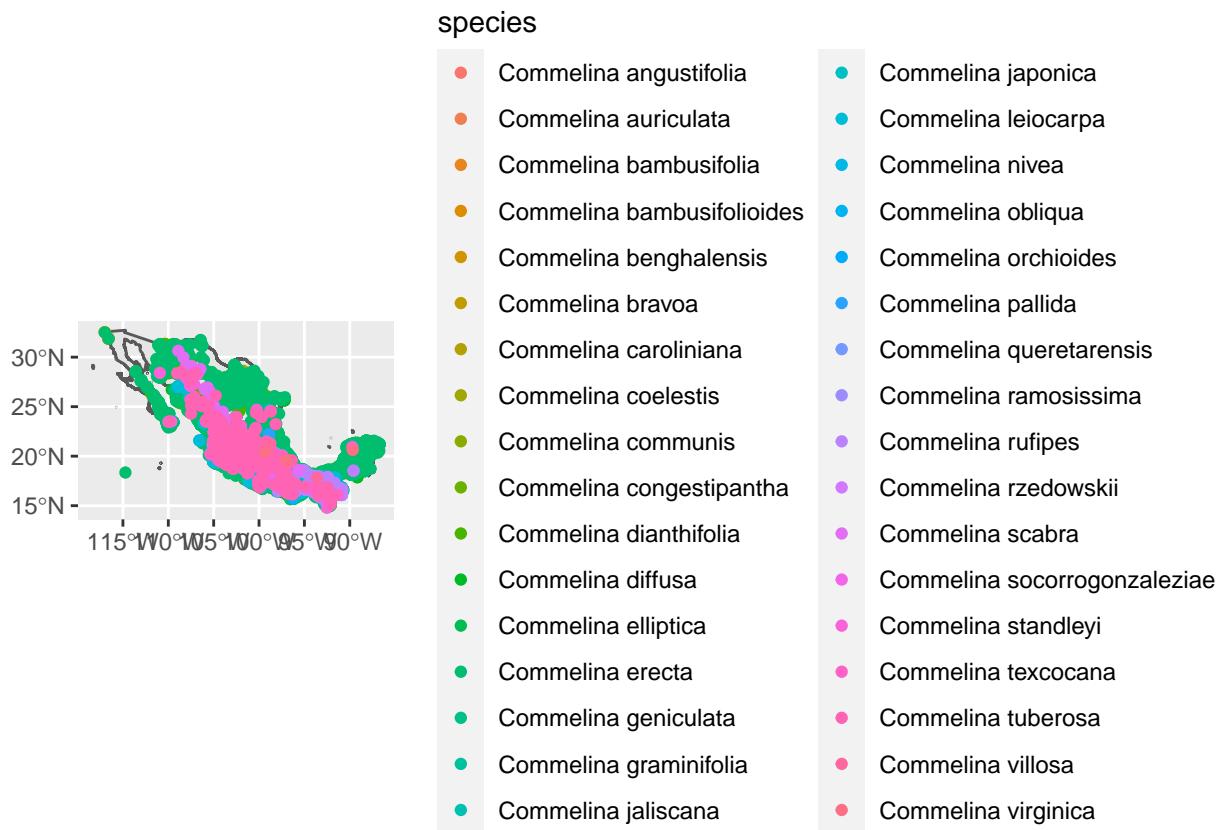
```
ggplot() +
  geom_sf(data = mapa_chiapas, aes(geometry = geometry),
          fill = "white", color = "black", size = 1/3) +
  geom_sf(data = commelina_chiapas, aes(geometry = geometry,
                                         color = species),
          pch = 15, size = 3) +
  labs(
    x = NULL,
    y = NULL,
    color = "Especie"
  ) +
  theme_bw() +
  theme(panel.background = element_rect(fill = "grey95"),
        panel.grid = element_line(linetype = 3,
                                  color = "grey60"),
        legend.text = element_text(face = "italic")) +
  scale_color_palatteer_d("DresdenColor::paired")
```



6.4 Modificación de la leyenda

Realizar este proceso para el mapa completo de México presenta algunos problemas. Debido a que hay una gran cantidad de especies, el espacio que ocupa la leyenda dificulta la visualización del mapa:

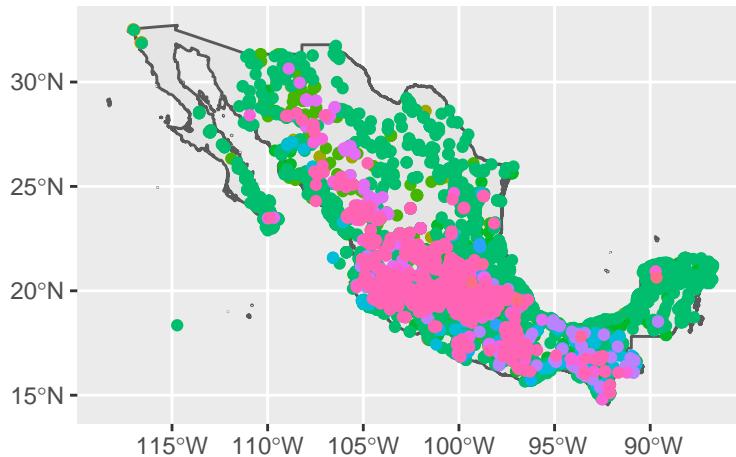
```
ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry)) +
  geom_sf(data = inter_commelina, aes(geometry = geometry,
                                       color = species))
```



En esta sección se describen los pasos necesarios para ajustar el tamaño y distribución de la leyenda. El mapa resultante requiere un gran nivel de detalle para apreciar la información representada, por lo cual sus dimensiones serán mayores a las de todas las gráficas y mapas mostrados anteriormente. Las imágenes de esta sección se verán distorsionadas debido a que las dimensiones usadas exceden el espacio de cada imagen. En RStudio ocurre una situación similar en la ventana **Plots**, pero al exportar el mapa todas las partes tendrán las proporciones adecuadas. El mapa resultante tendrá un ancho de 28 cm y una altura de 15 cm, de acuerdo con estas dimensiones se asignará el tamaño de letra, tipo y tamaño de carácter, posición y tamaño de leyenda.

El primer paso consiste en eliminar los nombres de los ejes del mapa y colocar la leyenda en la parte inferior:

```
ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry)) +
  geom_sf(data = inter_commelina, aes(geometry = geometry,
                                         color = species)) +
  labs(x = NULL,
       y = NULL,
       color = NULL) +
  theme(legend.position = "bottom")
```



stifolia	● Commelina coelestis	● Commelina geniculata	● Commelina orchoides	● C
ulata	● Commelina communis	● Commelina graminifolia	● Commelina pallida	● C
usifolia	● Commelina congestipantha	● Commelina jaliscana	● Commelina queretarensis	● C
usifolioides	● Commelina dianthifolia	● Commelina japonica	● Commelina ramosissima	● C
halensis	● Commelina diffusa	● Commelina leiocarpa	● Commelina rufipes	● C
ia	● Commelina elliptica	● Commelina nivea	● Commelina rzedowskii	● C
niana	● Commelina erecta	● Commelina obliqua	● Commelina scabra	

Para disminuir la extensión horizontal de la leyenda es posible cambiar en el objeto `inter_commelina` el nombre del género para que aparezca abreviado. Para cambiar los nombres se usa el paquete `stringr` que es cargado con el resto de las librerías de `tidyverse`. Con la función `str_replace()` se sustituye un patrón de caracteres (el nombre del género) dentro de una columna. Esta función se utiliza dentro de `mutate()` para crear una nueva versión de la columna `species`. En el argumento `string` se coloca el nombre de la columna en donde se realiza la sustitución, en `pattern` el nombre del género y en `replacement` la abreviación. Para esta última es necesario colocar un espacio después del punto:

```
commelina_abr <- inter_commelina %>%
  mutate(sp_abr = str_replace(string = species,
                             pattern = "Commelina",
                             replacement = "C. "))

commelina_abr

## Simple feature collection with 7991 features and 9 fields
## Geometry type: POINT
## Dimension:      XY
## Bounding box:  xmin: -117.031 ymin: 14.79667 xmax: -86.7157 ymax: 32.52555
## Geodetic CRS:  WGS 84
## # A tibble: 7,991 x 10
##       family     genus   species      count~1    AREA PERIM~2    COV_ COV_ID
##       <chr>      <chr>   <chr>        <chr>     <dbl>    <dbl> <dbl>    <dbl>
## 1 Commelinaceae Commelina Commelina erecta    MX      173.    185.     2    404
## 2 Commelinaceae Commelina Commelina diffusa    MX      173.    185.     2    404
## 3 Commelinaceae Commelina Commelina erecta    MX      173.    185.     2    404
## 4 Commelinaceae Commelina Commelina dianthi~    MX      173.    185.     2    404
## 5 Commelinaceae Commelina Commelina diffusa    MX      173.    185.     2    404
```

```

## 6 Commelinaceae Commelina Commelina diffusa MX      173.    185.    2    404
## 7 Commelinaceae Commelina Commelina erecta MX      173.    185.    2    404
## 8 Commelinaceae Commelina Commelina erecta MX      173.    185.    2    404
## 9 Commelinaceae Commelina Commelina erecta MX      173.    185.    2    404
## 10 Commelinaceae Commelina Commelina diffusa MX     173.    185.    2    404
## # ... with 7,981 more rows, 2 more variables: geometry <POINT [°]>,
## #   sp_abr <chr>, and abbreviated variable names 1: countryCode, 2: PERIMETER

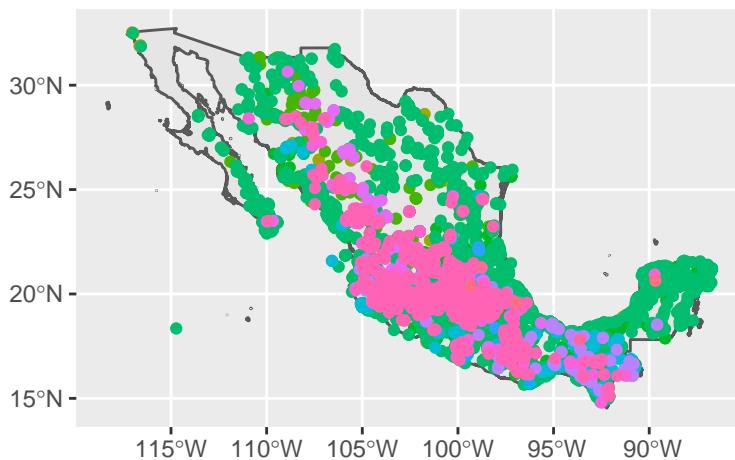
```

Usando el objeto `commelina_abr` la leyenda es más pequeña:

```

ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry)) +
  geom_sf(data = commelina_abr, aes(geometry = geometry,
                                      color = sp_abr)) +
  labs(x = NULL,
       y = NULL,
       color = NULL) +
  theme(legend.position = "bottom")

```

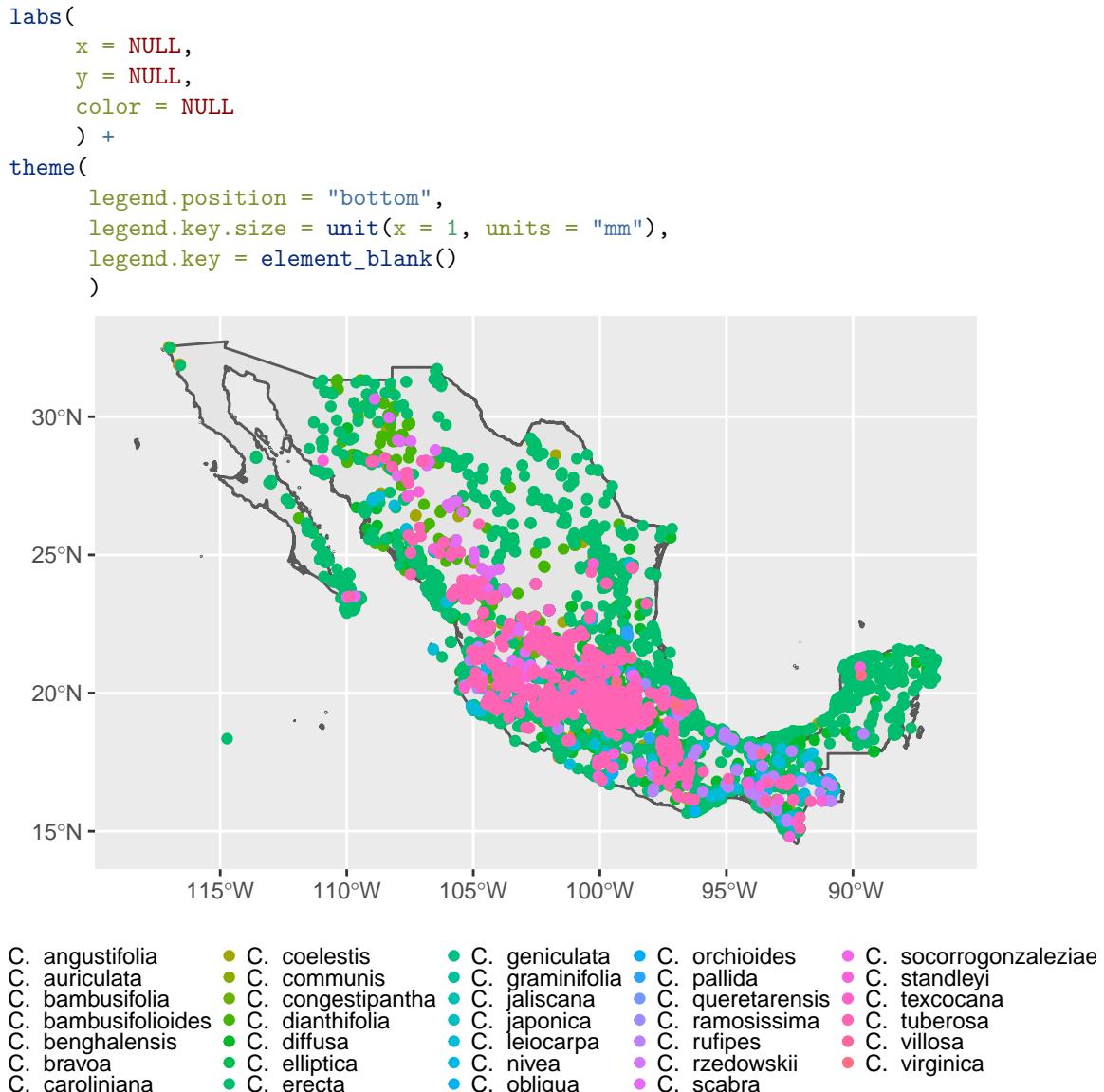


Otras modificaciones que permiten reducir el tamaño de la leyenda pueden hacerse en `theme()`. Modificando `legend.key.size` se puede disminuir la separación horizontal y vertical de cada uno de los elementos mediante la función `unit()`, que especifica la magnitud y dimensiones que tomará la nueva separación horizontal y vertical. Adicionalmente, `legend.key` permite eliminar el color de fondo de la leyenda al especificar `element_blank()`:

```

ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry)) +
  geom_sf(data = commelina_abr, aes(geometry = geometry,
                                      color = sp_abr)) +

```



Debido a que el tamaño de la imagen exportada será mayor a lo que se observa aquí, es posible expandir aún más la leyenda horizontalmente. Estas opciones se encuentran en la función `guides()`, en la cual se modifican las leyendas que están asociadas a una variable mediante una de las estéticas, en este caso `color`. Debido a ello se utiliza `color` como argumento y se especifica que las modificaciones corresponden a la leyenda con `guide_legend()`. Con esta función es posible modificar la manera en la que se distribuyen los elementos de la leyenda, para este mapa se acomodarán en cuatro filas usando `nrow`:

```

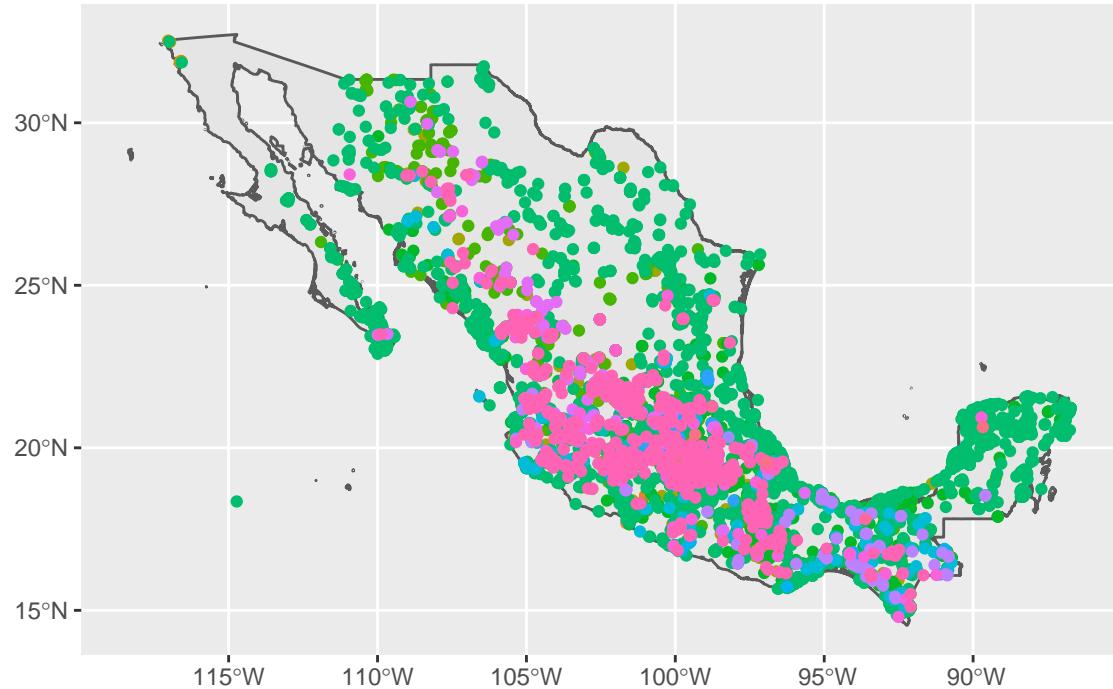
ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry)) +
  geom_sf(data = commelina_abr, aes(geometry = geometry,
                                    color = sp_abr)) +
  labs(
    x = NULL,
    y = NULL,
    color = NULL
  ) +
  theme(

```

```

    legend.position = "bottom",
    legend.key.size = unit(x = 1, units = "mm"),
    legend.key = element_blank()
) +
guides(color = guide_legend(nrow = 4))

```



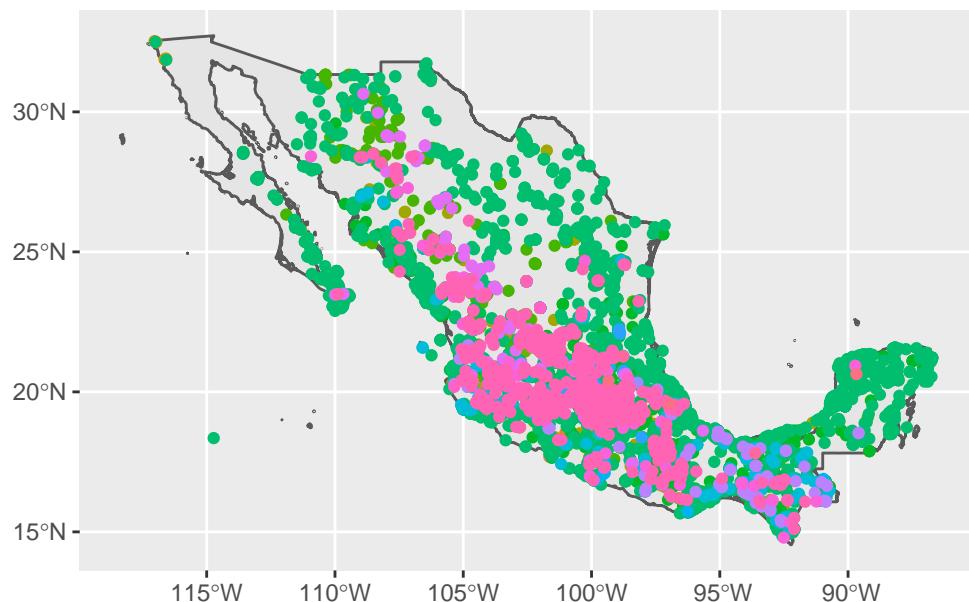
benghalensis	● <i>C. communis</i>	● <i>C. elliptica</i>	● <i>C. jaliscana</i>	● <i>C. obliqua</i>	● <i>C. ramosissima</i>	● <i>C.</i>
bravae	● <i>C. congestipantha</i>	● <i>C. erecta</i>	● <i>C. japonica</i>	● <i>C. orchoides</i>	● <i>C. rufipes</i>	● <i>C.</i>
caroliniana	● <i>C. dianthifolia</i>	● <i>C. geniculata</i>	● <i>C. leiocarpa</i>	● <i>C. pallida</i>	● <i>C. rzedowskii</i>	● <i>C.</i>
coelestis	● <i>C. diffusa</i>	● <i>C. graminifolia</i>	● <i>C. nivea</i>	● <i>C. queretarensis</i>	● <i>C. scabra</i>	● <i>C.</i>

Otro aspecto a considerar es el tamaño de la letra. Para que se ajuste adecuadamente a las dimensiones de la imagen exportada se usará un tamaño de 18 puntos. En el argumento `legend.text` dentro de `theme()`, se usa la función `element_text()` para especificar el tamaño. Si únicamente se coloca un número, las dimensiones serán automáticamente en puntos. En la misma función se puede ajustar el tipo de letra para que sean itálicas con el argumento `face`:

```

ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry)) +
  geom_sf(data = commelina_abr, aes(geometry = geometry,
                                      color = sp_abr)) +
  labs(
    x = NULL,
    y = NULL,
    color = NULL
  ) +
  theme(
    legend.position = "bottom",
    legend.key.size = unit(x = 1, units = "mm"),
    legend.key = element_blank(),
    legend.text = element_text(size = 18, face = "italic")
  ) +
  guides(color = guide_legend(nrow = 4))

```



<i>inis</i>	• <i>C. elliptica</i>	• <i>C. jaliscana</i>	• <i>C. obliqua</i>
<i>stipantha</i>	• <i>C. erecta</i>	• <i>C. japonica</i>	• <i>C. orchioides</i>
<i>folia</i>	• <i>C. geniculata</i>	• <i>C. leiocarpa</i>	• <i>C. pallida</i>
	• <i>C. graminifolia</i>	• <i>C. nivea</i>	• <i>C. queretarensis</i>

Las últimas modificaciones corresponden a otros aspectos visuales del mapa, como es el tipo y tamaño de carácter (de acuerdo con la imagen exportada), el color de fondo, tamaño de letra de los ejes y paleta de colores. La paleta será construida manualmente debido a que la cantidad de especies dificulta la elección de una paleta dentro de las que son provistas por el paquete `palettereer`:

```

paleta_commelina <- c("#2A6041", "#084C61", "#603140", "#FFC857",
                      "#323031", "#F15152", "#360568", "#7785AC",
                      "#A5E6BA", "#7D451B", "#D19C1D", "#9AC6C5",
                      "#65334D", "#A50104", "#272D2D", "#D782BA",
                      "#FCBA04", "#41521F", "#2D3141", "#037171",
                      "#315C2B", "#D782BA", "#012622", "#F4B860",
                      "#5C1A1B", "#C2E812", "#8C2155", "#DB4C40",
                      "#8B1E3F", "#525C55", "#F374AE", "#607196",
                      "#DF9B6D", "#0A1128")

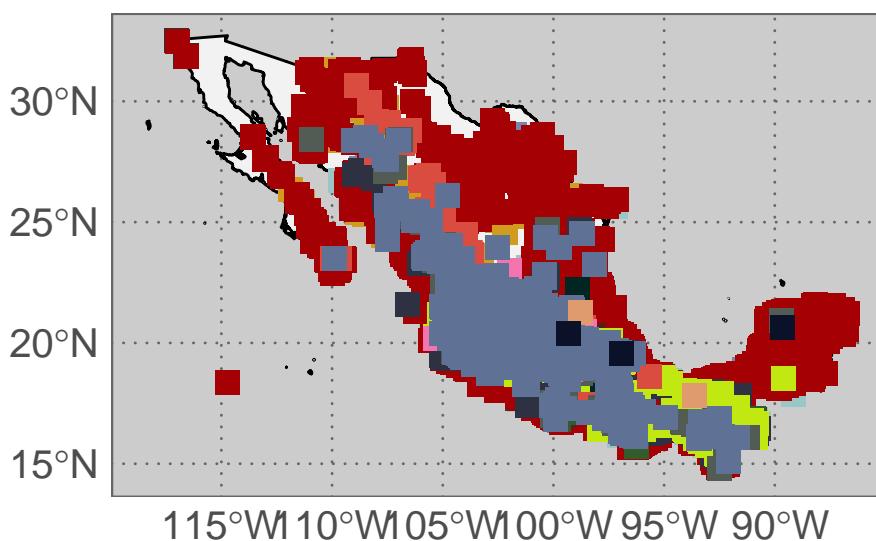
ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry),
          fill = "grey95", color = "black", size = 1/2) +
  geom_sf(data = commelina_abr, aes(geometry = geometry,
                                     color = sp_abr),
          pch = 15, size = 4) +
  scale_color_manual(values = paleta_commelina) +
  labs(
    x = NULL,
    y = NULL,
    color = NULL
  ) +
  theme_minimal() +

```

```

theme(
  legend.position = "bottom",
  legend.key.size = unit(x = 1, units = "mm"),
  legend.key = element_blank(),
  legend.text = element_text(size = 18, face = "italic"),
  axis.text = element_text(size = 15),
  legend.margin = margin(t = 6, r = 6, b = 6, l = 6, unit = "mm"),
  legend.background = element_rect(fill = "grey95",
                                    color = "grey40"),
  panel.background = element_rect(fill = "grey80",
                                    color = "grey40"),
  panel.grid = element_line(linetype = 3,
                            color = "grey40")
) +
guides(color = guide_legend(nrow = 4))

```



<i>nis</i>	<i>C. elliptica</i>	<i>C. jaliscana</i>	<i>C. obliqua</i>
<i>tipantha</i>	<i>C. erecta</i>	<i>C. japonica</i>	<i>C. orchiodes</i>
<i>olia</i>	<i>C. geniculata</i>	<i>C. leiocarpa</i>	<i>C. pallida</i>
	<i>C. graminifolia</i>	<i>C. nivea</i>	<i>C. queretarensis</i>

Finalmente, el mapa se exporta en formato .png, con las dimensiones previamente mencionadas y con una resolución de 300 pixeles por pulgada:

```
ggsave(filename = "mapas/mapa_commelina_mexico.png",
       plot = last_plot(), width = 28, height = 15, dpi = 300)
```

6.5 Mapa de tipos de vegetación

No todos los mapas utilizan coordenadas que pueden ser representadas como puntos. También se pueden usar polígonos dentro del área de un país para mostrar información. Un ejemplo de este caso es la distribución de los tipos de vegetación en México, la cual también se encuentra disponible en el portal de CONABIO bajo el apartado de Vegetación y uso de suelo>INEGI. El conjunto de datos usado aquí corresponde a la serie VII,

la descarga debe de ser en formato shapefile con coordenadas geográficas.

Esta capa también debe ser introducida con `st_read()`. Debido a que se trata de un archivo muy grande esta función puede tardar unos minutos:

```
vegetacion <- st_read("datos_manual/mapas/tipos_vegetacion/usv250s7gw.shp")  
## Reading layer `usv250s7gw' from data source  
##   `/home/leot/Documents/programacion/R/introduccion_R_biotologia/datos_manual/mapas/tipos_vegetacion/usv250s7gw.shp'  
##   using driver 'ESRI Shapefile'  
## Simple feature collection with 179064 features and 6 fields  
## Geometry type: MULTIPOLYGON  
## Dimension:      XY  
## Bounding box:  xmin: -118.3648 ymin: 14.53239 xmax: -86.71069 ymax: 32.71871  
## Geodetic CRS:  WGS 84
```

Las primeras dos columnas de este objeto describen el tipo de vegetación que corresponde a a cada polígono y al final se encuentra la columna geometry.

```
str(vegetacion)  
## Classes 'sf' and 'data.frame':  179064 obs. of  7 variables:  
## $ CLAVE      : chr  "ACUI" "ACUI" "ACUI" ...  
## $ DESCRIPCIO: chr  "ACUÍCOLA" "ACUÍCOLA" "ACUÍCOLA" ...  
## $ AREA       : num  334013 1989898 381081 289517 110040 ...  
## $ usv_svii  : num  1.04e+10 1.04e+10 1.04e+10 1.04e+10 1.04e+10 ...  
## $ cov_       : num  11 12 13 14 15 16 17 18 19 20 ...  
## $ cov_id    : num  12 13 14 15 16 17 18 19 20 21 ...  
## $ geometry   :sfct MULTIPOLYGON of length 179064; first list element: List of 1  
##   ..$ :List of 1  
##     ... .$: num [1:50, 1:2] -102 -102 -102 -102 -102 ...  
##     ..- attr(*, "class")= chr [1:3] "MULTIPOLYGON" "sfg"  
##     - attr(*, "sf_column")= chr "geometry"  
##     - attr(*, "agr")= Factor w/ 3 levels "constant","aggregate",...: NA NA NA NA NA  
##     ..- attr(*, "names")= chr [1:6] "CLAVE" "DESCRIPCIO" "AREA" "usv_svii" ...
```

Para evitar problemas relacionados con el rendimiento de R, se usará únicamente una fracción de todos estos datos, particularmente aquellos relacionados con el los diferentes tipos de matorral. Al explorar la columna DESCRIPCIO podemos ver todos los tipos de vegetación:

```
unique(vegetacion$DESCRIPCIO)  
## [1] "ACUÍCOLA"  
## [2] "DESPROVISTO DE VEGETACIÓN"  
## [3] "ASENTAMIENTOS HUMANOS"  
## [4] "BOSQUE DE OYAMEL"  
## [5] "BOSQUE CULTIVADO"  
## [6] "BOSQUE DE GALERÍA"  
## [7] "BOSQUE DE TÁSCATE"  
## [8] "BOSQUE MESÓFILO DE MONTAÑA"  
## [9] "BOSQUE DE PINO"  
## [10] "BOSQUE DE PINO-ENCINO"  
## [11] "BOSQUE DE ENCINO"  
## [12] "BOSQUE DE ENCINO-PINO"  
## [13] "BOSQUE DE AYARÍN"  
## [14] "SIN VEGETACIÓN APARENTE"  
## [15] "CUERPO DE AGUA"  
## [16] "AGRICULTURA DE HUMEDAD ANUAL"
```

```
## [17] "AGRICULTURA DE HUMEDAD ANUAL Y PERMANENTE"
## [18] "AGRICULTURA DE HUMEDAD ANUAL Y SEMIPERMANENTE"
## [19] "AGRICULTURA DE HUMEDAD SEMIPERMANENTE"
## [20] "MATORRAL CRASICAULE"
## [21] "MATORRAL DESÉRTICO MICRÓFILO"
## [22] "MATORRAL DESÉRTICO ROSETÓFILO"
## [23] "BOSQUE DE MEZQUITE"
## [24] "MEZQUITAL TROPICAL"
## [25] "MEZQUITAL XERÓFILO"
## [26] "MATORRAL SARCOCAUDE"
## [27] "MATORRAL SARCO-CRASICAULE"
## [28] "MATORRAL SARCO-CRASICAULE DE NEBLINA"
## [29] "PASTIZAL CULTIVADO"
## [30] "PASTIZAL HALÓFILO"
## [31] "PASTIZAL INDUCIDO"
## [32] "PASTIZAL NATURAL"
## [33] "PASTIZAL GIPSÓFILO"
## [34] "AGRICULTURA DE RIEGO ANUAL"
## [35] "AGRICULTURA DE RIEGO ANUAL Y PERMANENTE"
## [36] "AGRICULTURA DE RIEGO ANUAL Y SEMIPERMANENTE"
## [37] "AGRICULTURA DE RIEGO PERMANENTE"
## [38] "AGRICULTURA DE RIEGO SEMIPERMANENTE"
## [39] "AGRICULTURA DE RIEGO SEMIPERMANENTE Y PERMANENTE"
## [40] "SELVA BAJA CADUCIFOLIA"
## [41] "SELVA BAJA ESPINOSA CADUCIFOLIA"
## [42] "SELVA BAJA SUBCADUCIFOLIA"
## [43] "SELVA DE GALERÍA"
## [44] "SELVA MEDIANA CADUCIFOLIA"
## [45] "SELVA MEDIANA SUBPERENNIFOLIA"
## [46] "SELVA MEDIANA SUBCADUCIFOLIA"
## [47] "AGRICULTURA DE TEMPORAL ANUAL"
## [48] "AGRICULTURA DE TEMPORAL ANUAL Y PERMANENTE"
## [49] "AGRICULTURA DE TEMPORAL ANUAL Y SEMIPERMANENTE"
## [50] "AGRICULTURA DE TEMPORAL PERMANENTE"
## [51] "AGRICULTURA DE TEMPORAL SEMIPERMANENTE"
## [52] "AGRICULTURA DE TEMPORAL SEMIPERMANENTE Y PERMANENTE"
## [53] "POPAL"
## [54] "VEGETACIÓN DE GALERÍA"
## [55] "VEGETACIÓN HALÓFILA XERÓFILA"
## [56] "VEGETACIÓN HALÓFILA HIDRÓFILA"
## [57] "MANGLAR"
## [58] "PALMAR INDUCIDO"
## [59] "PALMAR NATURAL"
## [60] "VEGETACIÓN SECUNDARIA ARBÓREA DE BOSQUE DE GALERÍA"
## [61] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE BOSQUE DE TÁSCATE"
## [62] "VEGETACIÓN SECUNDARIA ARBÓREA DE BOSQUE DE TÁSCATE"
## [63] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE BOSQUE MESÓFILO DE MONTAÑA"
## [64] "VEGETACIÓN SECUNDARIA ARBÓREA DE BOSQUE MESÓFILO DE MONTAÑA"
## [65] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE BOSQUE DE PINO"
## [66] "VEGETACIÓN SECUNDARIA ARBÓREA DE BOSQUE DE PINO"
## [67] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE BOSQUE DE PINO-ENCINO"
## [68] "VEGETACIÓN SECUNDARIA ARBÓREA DE BOSQUE DE PINO-ENCINO"
## [69] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE BOSQUE DE ENCINO"
## [70] "VEGETACIÓN SECUNDARIA ARBÓREA DE BOSQUE DE ENCINO"
```

[71] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE BOSQUE DE ENCINO-PINO"
[72] "VEGETACIÓN SECUNDARIA ARBÓREA DE BOSQUE DE ENCINO-PINO"
[73] "VEGETACIÓN SECUNDARIA ARBÓREA DE BOSQUE DE AYARÍN"
[74] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE MATORRAL CRASICAULE"
[75] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE MATORRAL DESÉRTICO MICRÓFILO"
[76] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE MATORRAL DESÉRTICO ROSETÓFILO"
[77] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE BOSQUE DE MEZQUITE"
[78] "VEGETACIÓN SECUNDARIA ARBÓREA DE BOSQUE DE MEZQUITE"
[79] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE MEZQUITAL XERÓFILO"
[80] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE MATORRAL SARCOCAULE"
[81] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE MATORRAL SARCO-CRASICAULE"
[82] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE MATORRAL SUBTROPICAL"
[83] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE PASTIZAL NATURAL"
[84] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE SELVA BAJA CADUCIFOLIA"
[85] "VEGETACIÓN SECUNDARIA ARBÓREA DE SELVA BAJA CADUCIFOLIA"
[86] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE SELVA BAJA ESPINOSA CADUCIFOLIA"
[87] "VEGETACIÓN SECUNDARIA ARBÓREA DE SELVA BAJA ESPINOSA CADUCIFOLIA"
[88] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE SELVA BAJA SUBCADUCIFOLIA"
[89] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE SELVA MEDIANA CADUCIFOLIA"
[90] "VEGETACIÓN SECUNDARIA ARBÓREA DE SELVA MEDIANA CADUCIFOLIA"
[91] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE SELVA MEDIANA SUBPERENNIFOLIA"
[92] "VEGETACIÓN SECUNDARIA ARBÓREA DE SELVA MEDIANA SUBPERENNIFOLIA"
[93] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE SELVA MEDIANA SUBCADUCIFOLIA"
[94] "VEGETACIÓN SECUNDARIA ARBÓREA DE SELVA MEDIANA SUBCADUCIFOLIA"
[95] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE MANGLAR"
[96] "VEGETACIÓN SECUNDARIA ARBÓREA DE MANGLAR"
[97] "VEGETACIÓN SECUNDARIA HERBÁcea DE BOSQUE MESÓFILO DE MONTAÑA"
[98] "VEGETACIÓN SECUNDARIA HERBÁcea DE BOSQUE DE PINO"
[99] "VEGETACIÓN SECUNDARIA HERBÁcea DE BOSQUE DE PINO-ENCINO"
[100] "VEGETACIÓN SECUNDARIA HERBÁcea DE BOSQUE DE ENCINO"
[101] "VEGETACIÓN SECUNDARIA HERBÁcea DE BOSQUE DE ENCINO-PINO"
[102] "VEGETACIÓN SECUNDARIA HERBÁcea DE PASTIZAL NATURAL"
[103] "VEGETACIÓN SECUNDARIA HERBÁcea DE SELVA BAJA CADUCIFOLIA"
[104] "VEGETACIÓN SECUNDARIA HERBÁcea DE SELVA MEDIANA SUBPERENNIFOLIA"
[105] "VEGETACIÓN SECUNDARIA HERBÁcea DE SELVA MEDIANA SUBCADUCIFOLIA"
[106] "VEGETACIÓN SECUNDARIA HERBÁcea DE MANGLAR"
[107] "SABANOIDE"
[108] "TULAR"
[109] "VEGETACIÓN DE DUNAS COSTERAS"
[110] "PRADERA DE ALTA MONTAÑA"
[111] "BOSQUE DE CEDRO"
[112] "BOSQUE INDUCIDO"
[113] "AGRICULTURA DE HUMEDAD PERMANENTE"
[114] "AGRICULTURA DE HUMEDAD SEMIPERMANENTE Y PERMANENTE"
[115] "MATORRAL ESPINOSO TAMAULIPECO"
[116] "CHAPARRAL"
[117] "MATORRAL SUBMONTANO"
[118] "VEGETACIÓN DE PETÉN"
[119] "SELVA ALTA PERENNIFOLIA"
[120] "SELVA ALTA SUBPERENNIFOLIA"
[121] "SELVA BAJA PERENNIFOLIA"
[122] "SELVA BAJA ESPINOSA SUBPERENNIFOLIA"
[123] "SELVA BAJA SUBPERENNIFOLIA"
[124] "SABANA"

[125] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE BOSQUE DE OYAMEL"
[126] "VEGETACIÓN SECUNDARIA ARBÓREA DE BOSQUE DE OYAMEL"
[127] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE BOSQUE DE CEDRO"
[128] "VEGETACIÓN SECUNDARIA ARBÓREA DE BOSQUE DE CEDRO"
[129] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE MATORRAL ESPINOSO TAMAULIPECO"
[130] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE MEZQUITAL TROPICAL"
[131] "VEGETACIÓN SECUNDARIA ARBÓREA DE MEZQUITAL TROPICAL"
[132] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE CHAPARRAL"
[133] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE MATORRAL SUBMONTANO"
[134] "VEGETACIÓN SECUNDARIA ARBÓREA DE VEGETACIÓN DE PETÉN"
[135] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE SELVA ALTA PERENNIFOLIA"
[136] "VEGETACIÓN SECUNDARIA ARBÓREA DE SELVA ALTA PERENNIFOLIA"
[137] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE SELVA ALTA SUBPERENNIFOLIA"
[138] "VEGETACIÓN SECUNDARIA ARBÓREA DE SELVA ALTA SUBPERENNIFOLIA"
[139] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE SELVA BAJA PERENNIFOLIA"
[140] "VEGETACIÓN SECUNDARIA ARBÓREA DE SELVA BAJA PERENNIFOLIA"
[141] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE SELVA BAJA ESPINOSA SUBPERENNIFOLI"
[142] "VEGETACIÓN SECUNDARIA ARBÓREA DE SELVA BAJA ESPINOSA SUBPERENNIFOLIA"
[143] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE SELVA BAJA SUBPERENNIFOLIA"
[144] "VEGETACIÓN SECUNDARIA ARBÓREA DE SELVA BAJA SUBPERENNIFOLIA"
[145] "VEGETACIÓN SECUNDARIA ARBÓREA DE SELVA BAJA SUBCADUCIFOLIA"
[146] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE SELVA DE GALERÍA"
[147] "VEGETACIÓN SECUNDARIA ARBÓREA DE SELVA DE GALERÍA"
[148] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE SELVA MEDIANA PERENNIFOLIA"
[149] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE VEGETACIÓN DE GALERÍA"
[150] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE VEGETACIÓN HALÓFILA XERÓFILA"
[151] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE VEGETACIÓN HALÓFILA HIDRÓFILA"
[152] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE VEGETACIÓN DE DUNAS COSTERAS"
[153] "VEGETACIÓN SECUNDARIA HERBÁcea DE BOSQUE DE OYAMEL"
[154] "VEGETACIÓN SECUNDARIA HERBÁcea DE BOSQUE DE TÁSCATE"
[155] "VEGETACIÓN SECUNDARIA HERBÁcea DE MATORRAL CRASICAULE"
[156] "VEGETACIÓN SECUNDARIA HERBÁcea DE MATORRAL DESÉRTICO ROSETÓFILO"
[157] "VEGETACIÓN SECUNDARIA HERBÁcea DE SELVA ALTA PERENNIFOLIA"
[158] "VEGETACIÓN SECUNDARIA HERBÁcea DE SELVA ALTA SUBPERENNIFOLIA"
[159] "VEGETACIÓN SECUNDARIA HERBÁcea DE SELVA BAJA ESPINOSA CADUCIFOLIA"
[160] "VEGETACIÓN SECUNDARIA HERBÁcea DE SELVA BAJA ESPINOSA SUBPERENNIFOLI"
[161] "VEGETACIÓN SECUNDARIA HERBÁcea DE SELVA MEDIANA CADUCIFOLIA"
[162] "VEGETACIÓN SECUNDARIA HERBÁcea DE VEGETACIÓN HALÓFILA XERÓFILA"
[163] "VEGETACIÓN SECUNDARIA HERBÁcea DE PALMAR NATURAL"
[164] "VEGETACIÓN SECUNDARIA HERBÁcea DE VEGETACIÓN DE DUNAS COSTERAS"
[165] "MATORRAL ROSETÓFILO COSTERO"
[166] "MATORRAL SUBTROPICAL"
[167] "VEGETACIÓN DE DESIERTOS ARENOSOS"
[168] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE BOSQUE DE GALERÍA"
[169] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE MATORRAL ROSETÓFILO COSTERO"
[170] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE MATORRAL SARCO-CRASICAULE DE NEBLI"
[171] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE PASTIZAL HALÓFILO"
[172] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE VEGETACIÓN DE DESIERTOS ARENOSOS"
[173] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE PALMAR NATURAL"
[174] "VEGETACIÓN SECUNDARIA ARBÓREA DE PALMAR NATURAL"
[175] "VEGETACIÓN SECUNDARIA HERBÁcea DE MATORRAL DESÉRTICO MICRÓFILO"
[176] "VEGETACIÓN SECUNDARIA HERBÁcea DE MATORRAL ROSETÓFILO COSTERO"
[177] "VEGETACIÓN SECUNDARIA HERBÁcea DE MATORRAL SARCO-CRASICAULE"
[178] "VEGETACIÓN SECUNDARIA HERBÁcea DE MATORRAL SARCO-CRASICAULE DE NEBLI"

```

## [179] "VEGETACIÓN GIPSÓFILA"
## [180] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE BOSQUE DE AYARÍN"
## [181] "VEGETACIÓN SECUNDARIA ARBUSTIVA DE PASTIZAL GIPSÓFILO"
## [182] "VEGETACIÓN SECUNDARIA HERBÁcea DE MATORRAL ESPINOSO TAMAULIPECO"
## [183] "VEGETACIÓN SECUNDARIA HERBÁcea DE MATORRAL DE CONIFERAS"

```

Para empezar con algo muy sencillo, se usará únicamente un tipo de vegetación, el matorral desértico micrófilo. Primero se use `filter()` para crear un subconjunto que contenga únicamente los datos de este tipo de matorral:

```

matorral_micro <- vegetacion %>%
  filter(DESCRIPCIO == "MATORRAL DESÉRTICO MICRÓFILO")

head(matorral_micro)

## Simple feature collection with 6 features and 6 fields
## Geometry type: MULTIPOLYGON
## Dimension:      XY
## Bounding box:  xmin: -102.2803 ymin: 22.38139 xmax: -102.0517 ymax: 22.82362
## Geodetic CRS:  WGS 84
##   CLAVE           DESCRIPCIO     AREA    usv_svii cov_ cov_id
## 1  MDM MATORRAL DESÉRTICO MICRÓFILO 3911690 20904010400 39533 39534
## 2  MDM MATORRAL DESÉRTICO MICRÓFILO 11411484 20904010400 39557 39558
## 3  MDM MATORRAL DESÉRTICO MICRÓFILO 6177874 20904010400 39598 39599
## 4  MDM MATORRAL DESÉRTICO MICRÓFILO 3849344 20904010400 39599 39600
## 5  MDM MATORRAL DESÉRTICO MICRÓFILO 2711052 20904010400 39608 39609
## 6  MDM MATORRAL DESÉRTICO MICRÓFILO 9019459 20904010400 39609 39610
##   geometry
## 1 MULTIPOLYGON (((-102.1147 2...
## 2 MULTIPOLYGON (((-102.0651 2...
## 3 MULTIPOLYGON (((-102.1096 2...
## 4 MULTIPOLYGON (((-102.277 22...
## 5 MULTIPOLYGON (((-102.081 22...
## 6 MULTIPOLYGON (((-102.1202 2...

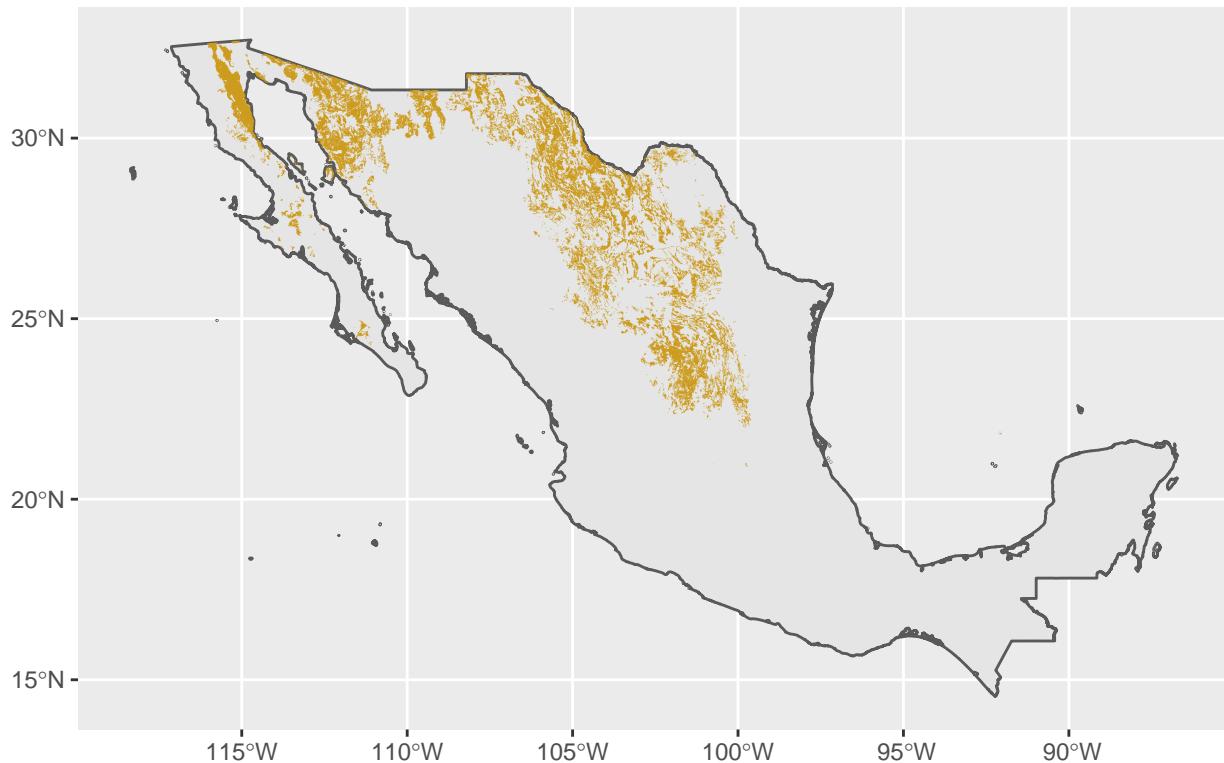
```

Usando el mapa de México de la sección anterior, es posible mostrar la distribución de este tipo de matorral en todo el país:

```

ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry)) +
  geom_sf(data = matorral_micro, aes(geometry = geometry),
         fill = "goldenrod3", color = "transparent")

```



El color del borde (`color`) se escribió como transparente para que sea más clara el área ocupada por el matorral.

Para graficar todos los tipos de matorral, por ejemplo, es necesario primero conocer todos los tipos de matorral incluidos en `vegetacion` para después indicar con `filter()` que se deben seleccionar las observaciones correspondientes solo a esas categorías.

Debido a que hay un total de 183 tipos de vegetación, revisar todo el vector para identificar aquellos que correspondan a diferentes tipos de matorral puede ser algo problemático ya que es una tarea propensa a errores además de larga. Usando las funciones básicas de R y de `stringr` (paquete de `tidyverse`) es posible darle a R la instrucciones adecuadas para hacer este proceso sin errores, aunque la secuencia de instrucciones puede ser algo complicada, por lo que se describirá a detalle.

Lo primero que hacer es guardar el vector anterior como un objeto para facilitar su uso en los pasos siguientes:

```
nombres_vegetacion <- unique(vegetacion$DESCRIPCION)
```

Después es necesario identificar todos aquellos elementos que corresponden a tipos de matorral dentro de `nombres_vegetacion`. En lugar de buscar en todo el vector, se usará la función `str_starts()` de `stringr`, en la cual se introduce una cadena de caracteres para buscar todas las observaciones de una columna o vector que empiezan con esa cadena. En este caso, se desea buscar todas aquellas que empiezan con “MATORRAL”. En el argumento `string` se coloca el vector construido anteriormente y en `pattern` la cadena de caracteres que se desea buscar.

El resultado de esta función es un vector de la misma longitud que el introducido en `string`, con valores de `TRUE` si es que el elemento contiene los primeros caracteres indicados o `FALSE` cuando no:

```
str_starts(string = nombres_vegetacion, pattern = "MATORRAL")
```

```
## [1] FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE FALSE FALSE
## [25] FALSE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [37] FALSE FALSE
```

```

## [49] FALSE FALSE
## [61] FALSE FALSE
## [73] FALSE FALSE
## [85] FALSE FALSE
## [97] FALSE FALSE
## [109] FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE FALSE FALSE FALSE
## [121] FALSE FALSE
## [133] FALSE FALSE
## [145] FALSE FALSE
## [157] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE FALSE
## [169] FALSE FALSE
## [181] FALSE FALSE FALSE

```

Por si mismo este vector no permite seleccionar los elementos que buscamos, pero ya sabemos que aquellos valores que son verdaderos corresponden a los tipos del matorral. Si podemos conocer qué posiciones dentro del vector son verdaderas, podemos obtener un nuevo vector con los tipos de los matorrales al usar [] para acceder los elementos adecuados en `nombres_vegetacion`. La función `which()`, permite hacer justo eso, ya que regresa un vector con las posiciones que son verdaderas:

```

str_starts(string = nombres_vegetacion, pattern = "MATORRAL") %>%
  which()

## [1] 20 21 22 26 27 28 115 117 165 166

```

Otra manera de llevar a cabo esta función sería colocar todos los contenidos de la primera línea dentro de `which()`, pero es más sencillo usar `%>%`. Ahora que tenemos el número de posiciones, podemos guardarlas en un objeto para facilitar el uso de []:

```

posiciones <- str_starts(string = nombres_vegetacion, pattern = "MATORRAL") %>%
  which()

```

De este modo, los nombres de los matorrales se obtienen de la siguiente manera:

```

nombres_vegetacion[posiciones]

## [1] "MATORRAL CRASICAULE"
## [2] "MATORRAL DESÉRTICO MICRÓFILO"
## [3] "MATORRAL DESÉRTICO ROSETÓFILO"
## [4] "MATORRAL SARCOCAULE"
## [5] "MATORRAL SARCO-CRASICAULE"
## [6] "MATORRAL SARCO-CRASICAULE DE NEBLINA"
## [7] "MATORRAL ESPINOSO TAMAULIPECO"
## [8] "MATORRAL SUBMONTANO"
## [9] "MATORRAL ROSETÓFILO COSTERO"
## [10] "MATORRAL SUBTROPICAL"

```

Ahora es necesario usar estos nombres para crear un subconjunto del objeto `vegetacion` que seleccione únicamente aquellas observaciones de los matorrales usando `filter()`:

```

matorrales <- vegetacion %>%
  filter(DESCRIPCION %in% nombres_vegetacion[posiciones])

head(matorrales)

## Simple feature collection with 6 features and 6 fields
## Geometry type: MULTIPOLYGON
## Dimension:      XY
## Bounding box:  xmin: -102.384 ymin: 20.95211 xmax: -102.2301 ymax: 21.97692
## Geodetic CRS:  WGS 84

```

```

##   CLAVE      DESCRIPCIO     AREA    usv_svii cov_ cov_id
## 1 MC MATORRAL CRASICAULE 952203.7 20906010400 38859 38860
## 2 MC MATORRAL CRASICAULE 820200.4 20906010400 38863 38864
## 3 MC MATORRAL CRASICAULE 5341788.0 20906010400 38865 38866
## 4 MC MATORRAL CRASICAULE 1152438.4 20906010400 38866 38867
## 5 MC MATORRAL CRASICAULE 939249.2 20906010400 38963 38964
## 6 MC MATORRAL CRASICAULE 2224267.8 20906010400 38965 38966
##
##                                     geometry
## 1 MULTIPOLYGON (((-102.3596 2...
## 2 MULTIPOLYGON (((-102.3718 2...
## 3 MULTIPOLYGON (((-102.3093 2...
## 4 MULTIPOLYGON (((-102.3331 2...
## 5 MULTIPOLYGON (((-102.2392 2...
## 6 MULTIPOLYGON (((-102.2499 2...

```

Para verificar que se tienen únicamente los datos requeridos podemos revisar las categorías en DESCRIPCIO:

```

unique(matorrales$DESCRIPCIO)

## [1] "MATORRAL CRASICAULE"
## [2] "MATORRAL DESÉRTICO MICRÓFILO"
## [3] "MATORRAL DESÉRTICO ROSETÓFILO"
## [4] "MATORRAL SARCOCAULE"
## [5] "MATORRAL SARCO-CRASICAULE"
## [6] "MATORRAL SARCO-CRASICAULE DE NEBLINA"
## [7] "MATORRAL ESPINOSO TAMAULIPECO"
## [8] "MATORRAL SUBMONTANO"
## [9] "MATORRAL ROSETÓFILO COSTERO"
## [10] "MATORRAL SUBTROPICAL"

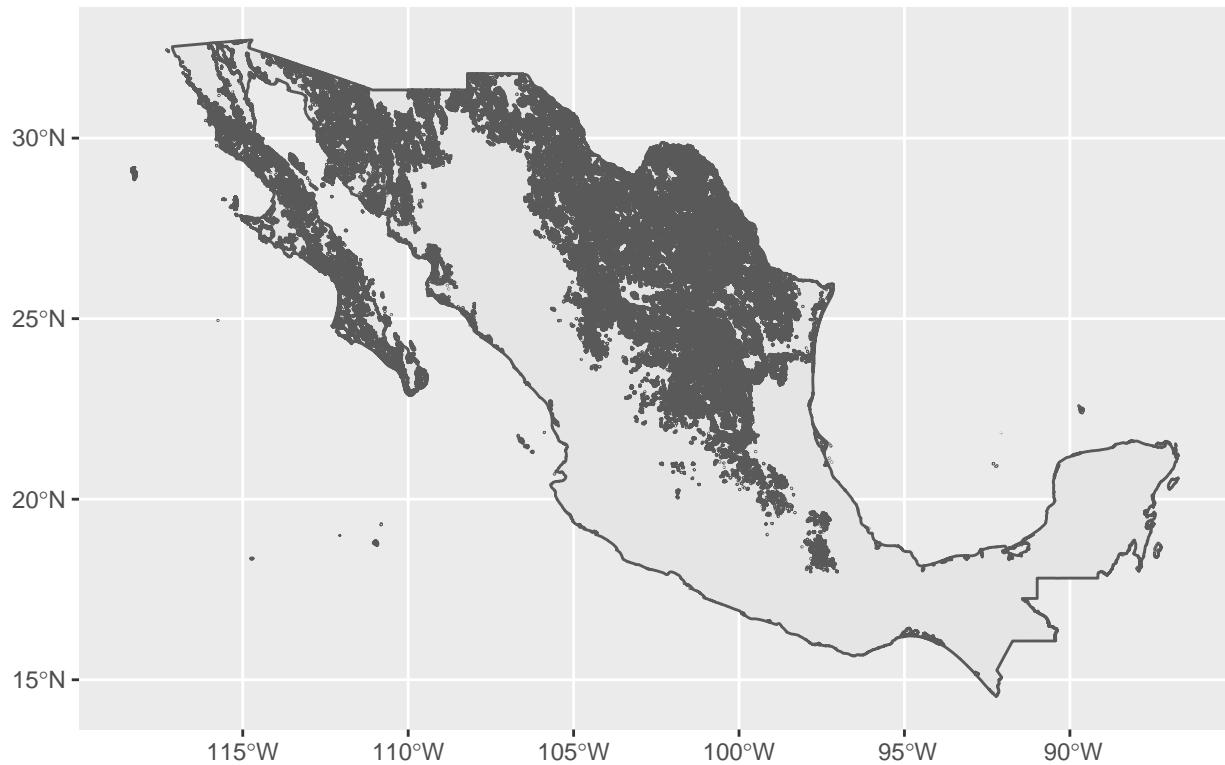
```

Con este conjunto de datos podemos ver la distribución de los matorrales:

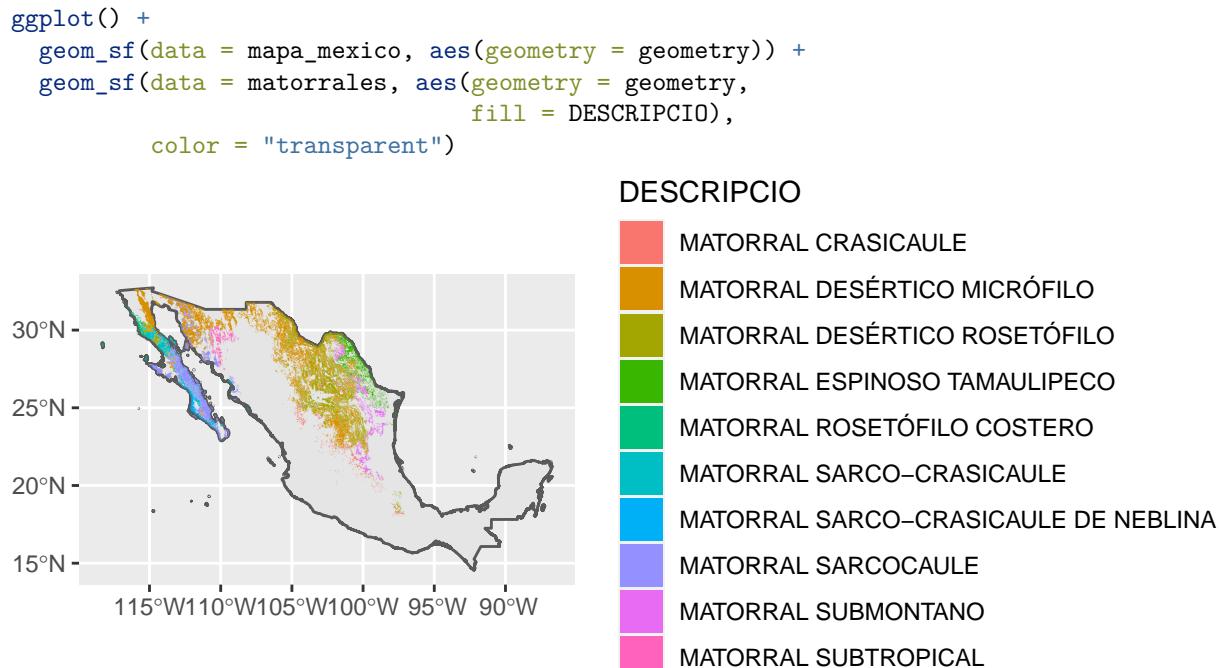
```

ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry)) +
  geom_sf(data = matorrales, aes(geometry = geometry))

```



Para distinguir entre cada tipo de matorral se debe indicar en `aes` que el atributo `fill` debe asociarse a la variable `DESCRIPCIO`:



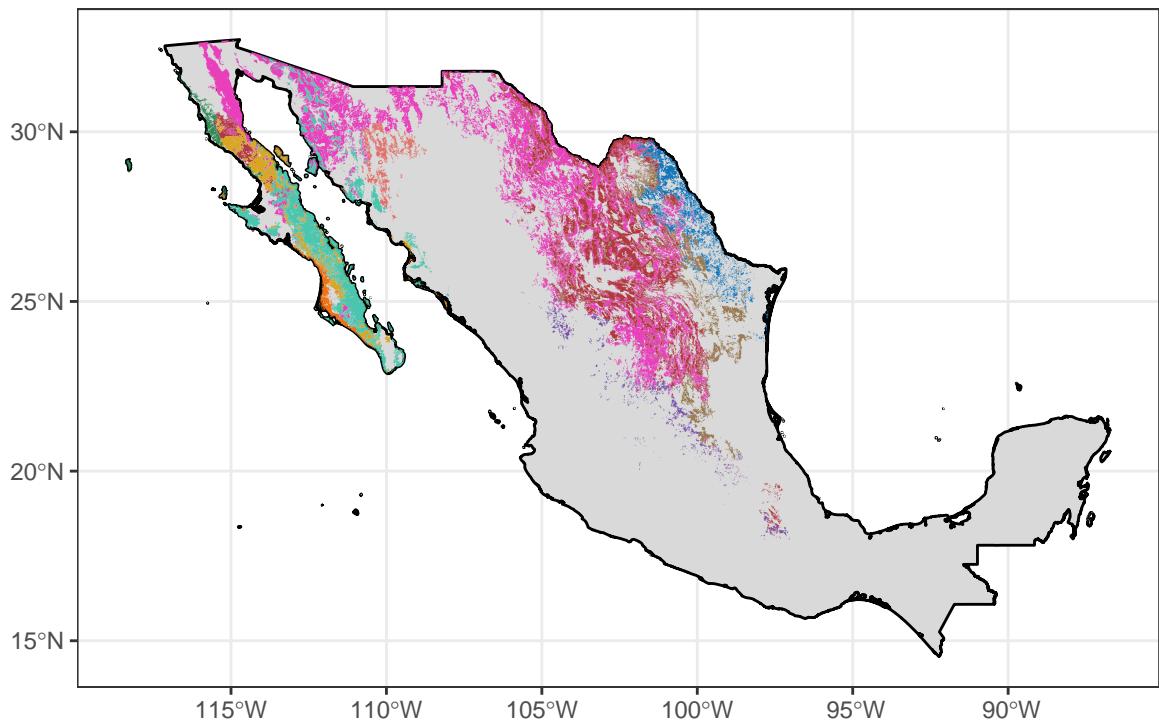
Recreando los mismos pasos usados en el mapa de distribución de la familia Commelinaceae este mapa queda de la siguiente manera:

```
ggplot() +
  geom_sf(data = mapa_mexico, aes(geometry = geometry),
```

```

        fill = "grey85", color = "black", size = .5) +
geom_sf(data = matorrales, aes(geometry =
                                    fill = DESCRIPCIO),
         color = "transparent") +
labs(fill = "Matorral") +
scale_fill_paletteer_d(palette = "basetheme::royal",
                       labels = c("Crasicaule", "Desértico micrófilo",
                                 "Desértico rosetófilo",
                                 "Epinoso tamaulipeco",
                                 "Rosetófilo costero", "Sarco-crasicaule",
                                 "Sarco-crasicaule de neblina",
                                 "Sarcocaula", "Submontano",
                                 "Subtropical")) +
theme_bw() +
theme(legend.position = "bottom",
      legend.text = element_text(size = 8),
      legend.key.size = unit(3, "mm"),
      legend.key = element_blank(),
      legend.margin = margin(3, 3, 3, 3, unit = "mm"),
      legend.background = element_rect(fill = "grey85",
                                       color = "black",
                                       size = .5)) +
guides(fill = guide_legend(nrow = 2))

```



Matorral	Crasicaule	Desértico rosetófilo	Rosetófilo costero	Sarco-crasicaule de neblina	Subm
	Desértico micrófilo	Epinoso tamaulipeco	Y	Sarcocaula	Subtrop

```

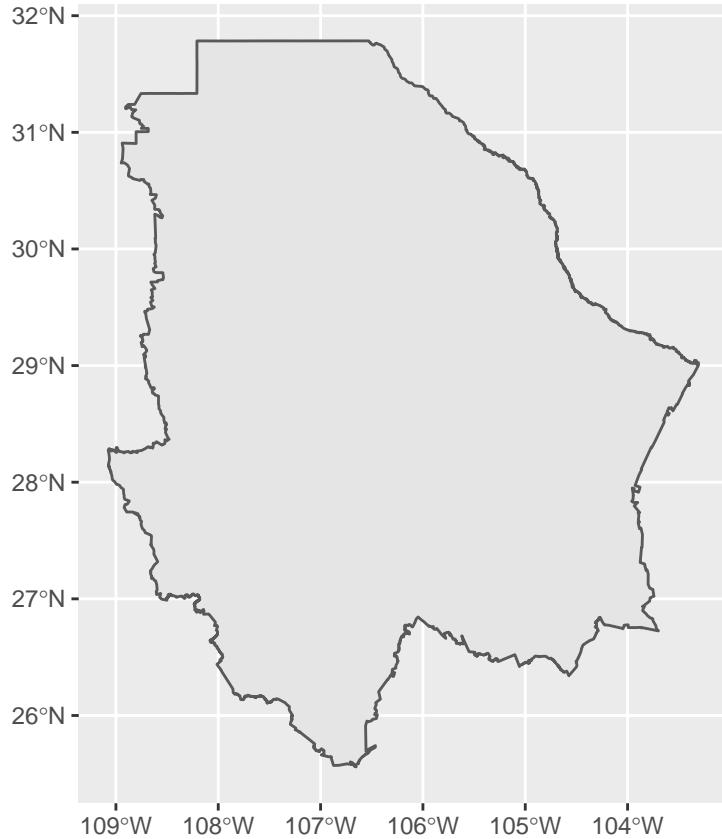
ggsave(filename = "mapas/mapa_matorrales.png", height = 15, width = 28,
       dpi = 300, units = "cm", plot = last_plot())

```

Finalmente, es posible combinar los diferentes métodos usados en estos mapas y en los anteriores para realizar

un mapa de Chihuahua en donde se muestren los matorrales presentes en el estado. El primer paso es crear el mapa de Chihuahua:

```
mapa_chihuahua <- mapa_estados %>%
  filter(NOM_ENT == "Chihuahua")  
  
ggplot() +
  geom_sf(data = mapa_chihuahua, aes(geometry = geometry))
```



Usando el objeto `matorrales` es posible hacer la intersección necesaria para este mapa, pero se muestra un error que no se había presentado antes:

```
matorrales_chihuahua <- st_intersection(x = matorrales,
                                         y = mapa_chihuahua)  
  
## Error in wk_handle.wk_wkb(wkb, s2_geography_writer(oriented = oriented, : Loop 0 is not valid: Edge
```

Para solucionar este error que tiene que ver con errores en aspectos de la geometría esférica. Con la función `st_make_valid()` se evita este error al convertir la geometría inválida en válida. Esta función debe aplicarse a `matorrales`. Para evitar llenar la memoria de objetos redundantes se sobrescribe el mismo objeto:

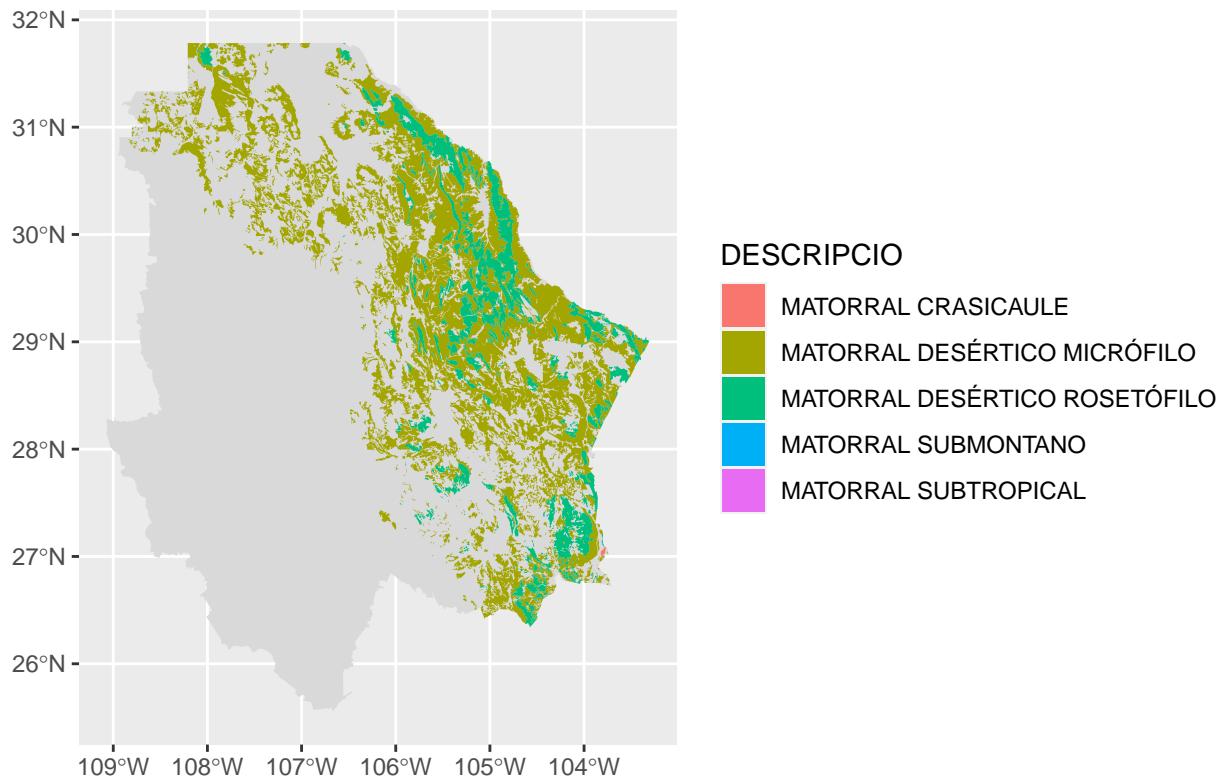
```
matorrales <- st_make_valid(matorrales)
```

Ahora al correr el código de la intersección no debería de haber ningún problema:

```
matorrales_chihuahua <- st_intersection(x = matorrales,
                                         y = mapa_chihuahua)  
  
## Warning: attribute variables are assumed to be spatially constant throughout all
## geometries
```

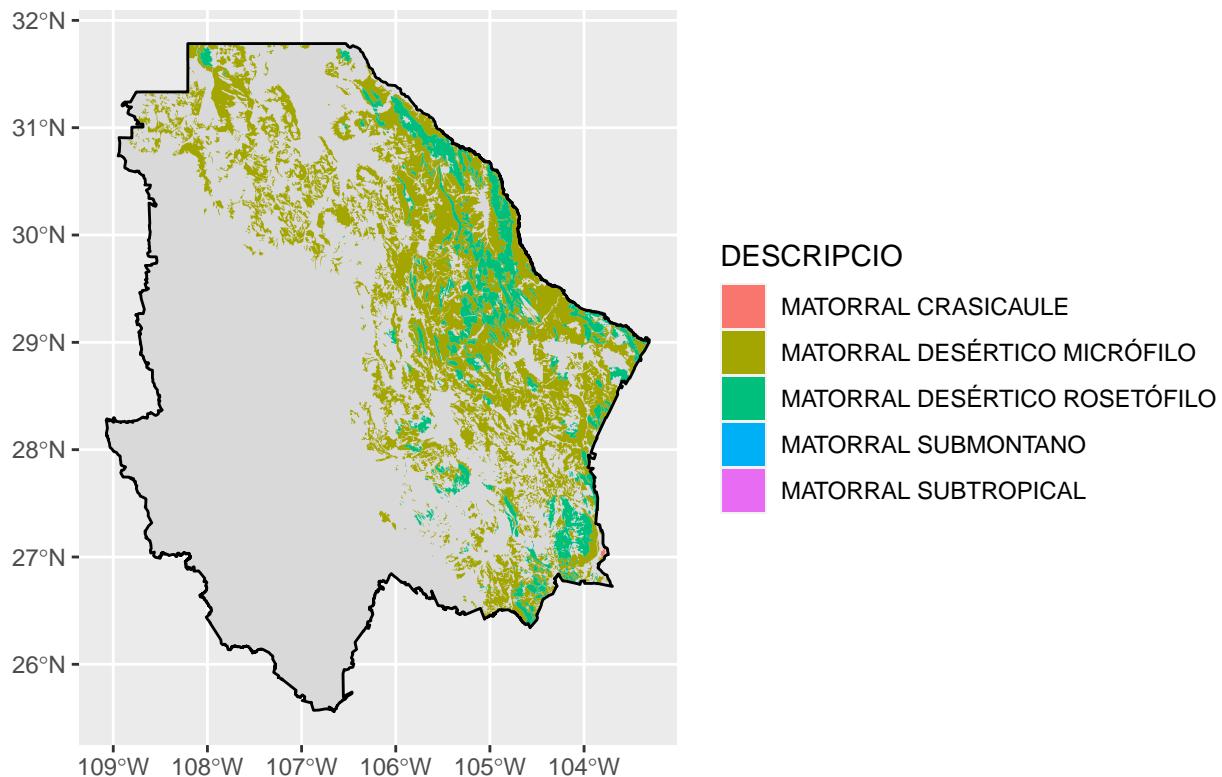
Usando este nuevo objeto es posible construir el mapa de los matorrales para Chihuahua:

```
ggplot() +
  geom_sf(data = mapa_chihuahua, aes(geometry = geometry),
          fill = "grey85", color = "transparent") +
  geom_sf(data = matorrales_chihuahua, aes(geometry = geometry,
                                             fill = DESCRIPCIO),
          color = "transparent")
```



Debido a que los tipos de matorral se encuentran por encima del mapa de Chihuahua los límites del estado no se aprecian claramente. Para solucionar esto, en la primera capa se elimina el color del contorno y después de la capa de los matorrales se coloca nuevamente el mapa de Chihuahua en donde únicamente se especifica el contorno, el cual queda por encima y se distingue claramente:

```
ggplot() +
  geom_sf(data = mapa_chihuahua, aes(geometry = geometry),
          fill = "grey85", color = "transparent") +
  geom_sf(data = matorrales_chihuahua, aes(geometry = geometry,
                                             fill = DESCRIPCIO),
          color = "transparent") +
  geom_sf(data = mapa_chihuahua, aes(geometry = geometry),
          fill = "transparent", color = "black", size = .5)
```



Aquí se observa que Chihuahua posee principalmente matorral desértico micrófilo y desértico rosetófilo. Los otros tipos de matorral no son apreciables mas que en muy pequeñas áreas, por lo que podrían eliminarse si es que se cuenta con suficiente información para realizar este cambio.

Si es que este es el caso, se puede modificar el objeto `matorrales_chihuahua` para solo incluir los dos tipos de matorral:

```
matorrales_chihuahua <- matorrales_chihuahua %>%
  filter(DESCRIPCIO %in% c("MATORRAL DESÉRTICO MICRÓFILO",
                           "MATORRAL DESÉRTICO ROSETÓFILO"))
head(matorrales_chihuahua)

## Simple feature collection with 6 features and 14 fields
## Geometry type: GEOMETRY
## Dimension: XY
## Bounding box: xmin: -105.0686 ymin: 26.34846 xmax: -104.4954 ymax: 26.49956
## Geodetic CRS: WGS 84
## CLAVE          DESCRICPIO      AREA    usv_svii cov_ cov_id
## 1   MDM MATORRAL DESÉRTICO MICRÓFILO 2388340.2 20904010400 40462 40463
## 2   MDM MATORRAL DESÉRTICO MICRÓFILO 102710146.3 20904010400 40506 40507
## 3   MDM MATORRAL DESÉRTICO MICRÓFILO 498803.2 20904010400 40512 40513
## 4   MDM MATORRAL DESÉRTICO MICRÓFILO 9299455.5 20904010400 40518 40519
## 5   MDM MATORRAL DESÉRTICO MICRÓFILO 1109431.9 20904010400 40521 40522
## 6   MDM MATORRAL DESÉRTICO MICRÓFILO 1081426.5 20904010400 40522 40523
## CVE_ENT    NOM_ENT CVE_CAP    NOM_CAP COV_ COV_ID AREA.1 PERIMETER
## 1     08 Chihuahua 190001 Chihuahua    7      8 24697336 3120.466
## 2     08 Chihuahua 190001 Chihuahua    7      8 24697336 3120.466
## 3     08 Chihuahua 190001 Chihuahua    7      8 24697336 3120.466
## 4     08 Chihuahua 190001 Chihuahua    7      8 24697336 3120.466
## 5     08 Chihuahua 190001 Chihuahua    7      8 24697336 3120.466
## 6     08 Chihuahua 190001 Chihuahua    7      8 24697336 3120.466
```

```

##                                     geometry
## 1 POLYGON ((-104.5623 26.3527...
## 2 MULTIPOLYGON (((-104.503 26...
## 3 POLYGON ((-104.5072 26.4734...
## 4 POLYGON ((-104.5005 26.4836...
## 5 POLYGON ((-105.0542 26.488, ...
## 6 POLYGON ((-105.0304 26.4995...

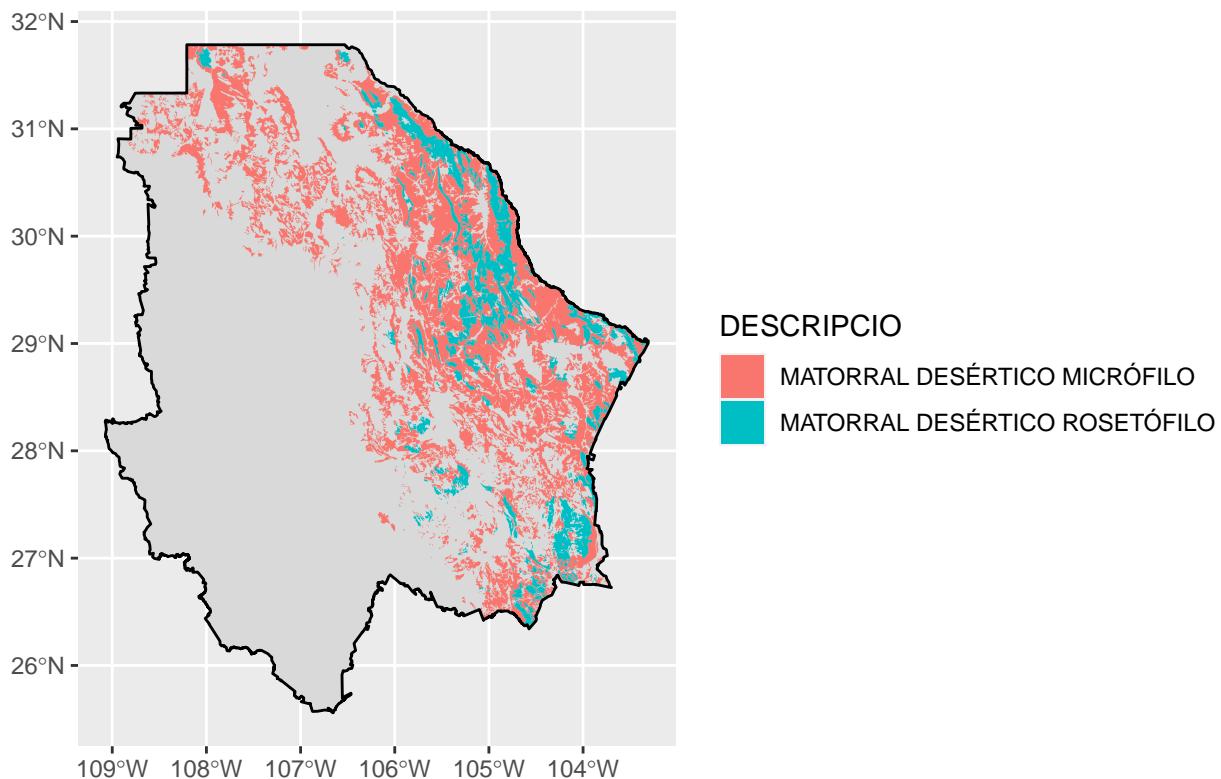
```

El nuevo mapa contiene únicamente estas dos categorías:

```

ggplot() +
  geom_sf(data = mapa_chihuahua, aes(geometry = geometry),
          fill = "grey85", color = "transparent") +
  geom_sf(data = matorrales_chihuahua, aes(geometry = geometry,
                                             fill = DESCRIPCIO),
          color = "transparent") +
  geom_sf(data = mapa_chihuahua, aes(geometry = geometry),
          fill = "transparent", color = "black", size = .5)

```

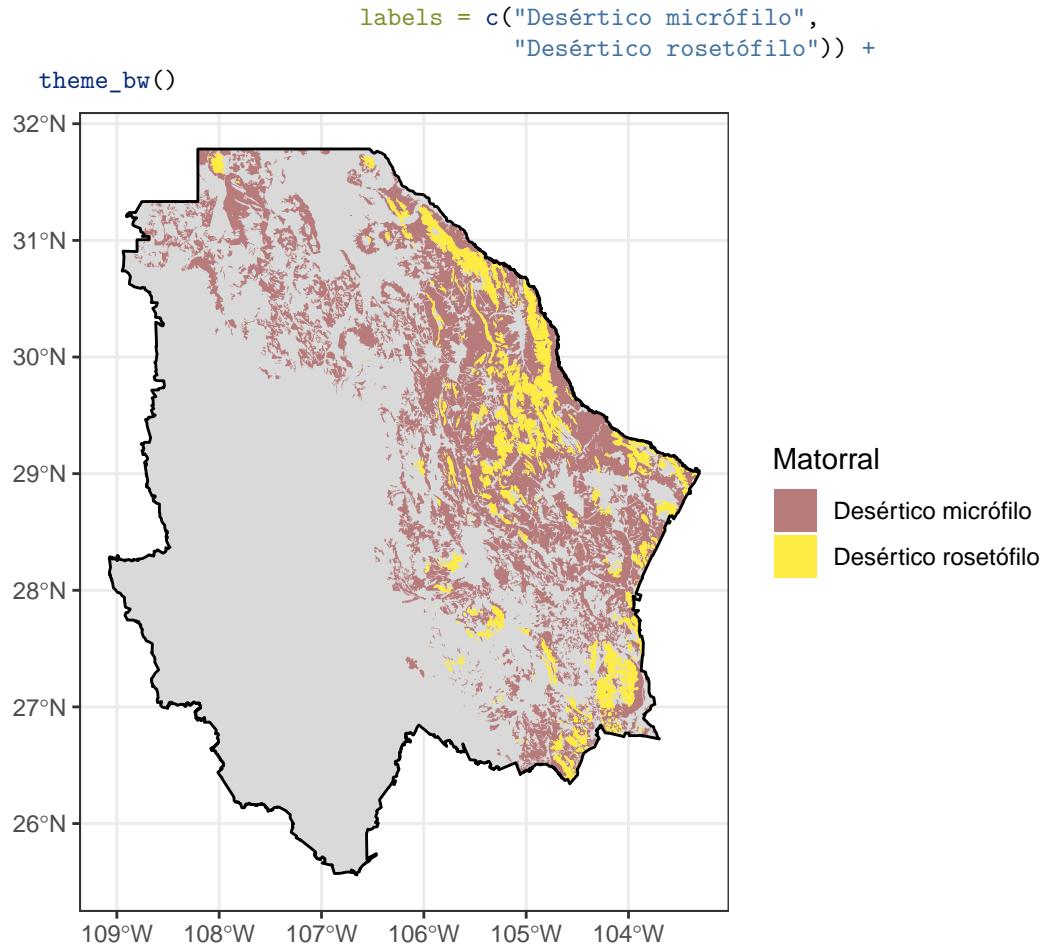


Ajustando algunos aspectos visuales, el mapa terminado queda así:

```

ggplot() +
  geom_sf(data = mapa_chihuahua, aes(geometry = geometry),
          fill = "grey85", color = "transparent") +
  geom_sf(data = matorrales_chihuahua, aes(geometry = geometry,
                                             fill = DESCRIPCIO),
          color = "transparent") +
  geom_sf(data = mapa_chihuahua, aes(geometry = geometry),
          fill = "transparent", color = "black", size = .5) +
  labs(fill = "Matorral") +
  scale_fill_palatteer_d(palette = "calecopal::fire",

```



7 Pruebas de hipótesis en R

Usualmente es necesario decidir si es posible realizar una prueba paramétrica o no debido a que son más robustas, debido a que requieren que los datos se ajusten a la distribución normal y que los grupos a comparar tengan varianzas iguales (homocedasticidad), es necesario primero probar estos supuestos. Antes de empezar con este procedimiento, debe tenerse en cuenta que solo se puede realizar una prueba paramétrica cuando se tienen numéricos continuos, de lo contrario debe optarse por una prueba no paramétrica sin necesidad de probar los supuestos.

7.1 Prueba de normalidad

En R se puede realizar la prueba de Shapiro para probar normalidad mediante la función `shapiro.test()`. Lo que se busca de esta prueba es que el valor de p sea mayor a 0.05, debido a que la hipótesis nula establece que los datos se distribuyen normalmente, mientras que la alternativa es que no:

$$H_0 : \text{distribución} = \text{normal}$$

$$H_A : \text{distribución} \neq \text{normal}$$

En la función solo se debe introducir en el argumento `x` el vector con los datos cuya distribución se quiere evaluar:

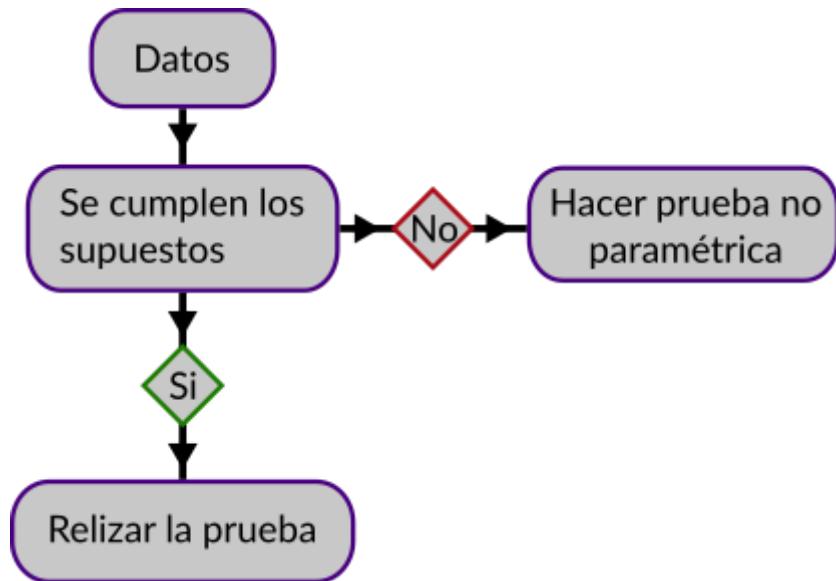


Figure 4: Decisión entre pruebas paramétricas y no paramétricas

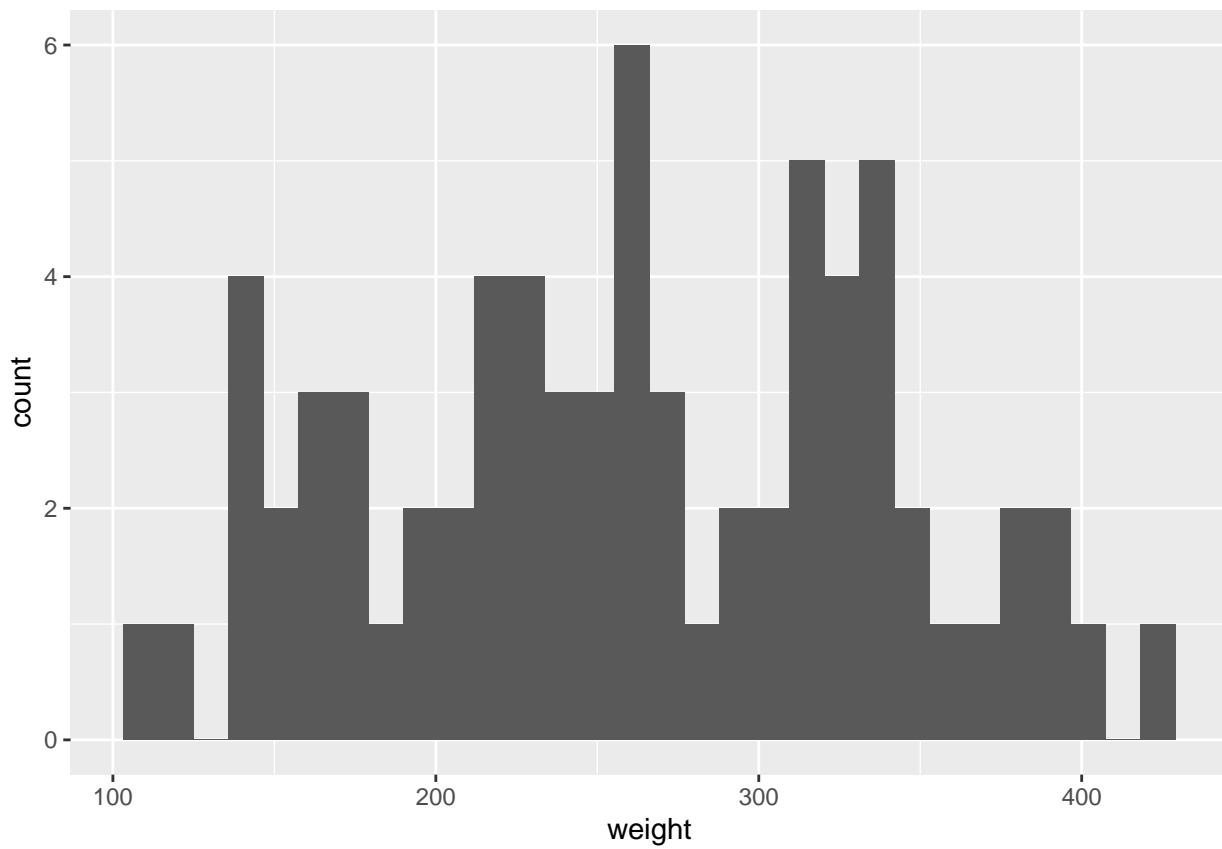
```

shapiro.test(x = chickwts$weight)
##
##  Shapiro-Wilk normality test
##
## data: chickwts$weight
## W = 0.97674, p-value = 0.2101
  
```

En este caso, el valor de p es mayor a 0.05, entonces la distribución es aproximadamente normal. La normalidad también puede evaluarse visualmente construyendo un histograma:

```

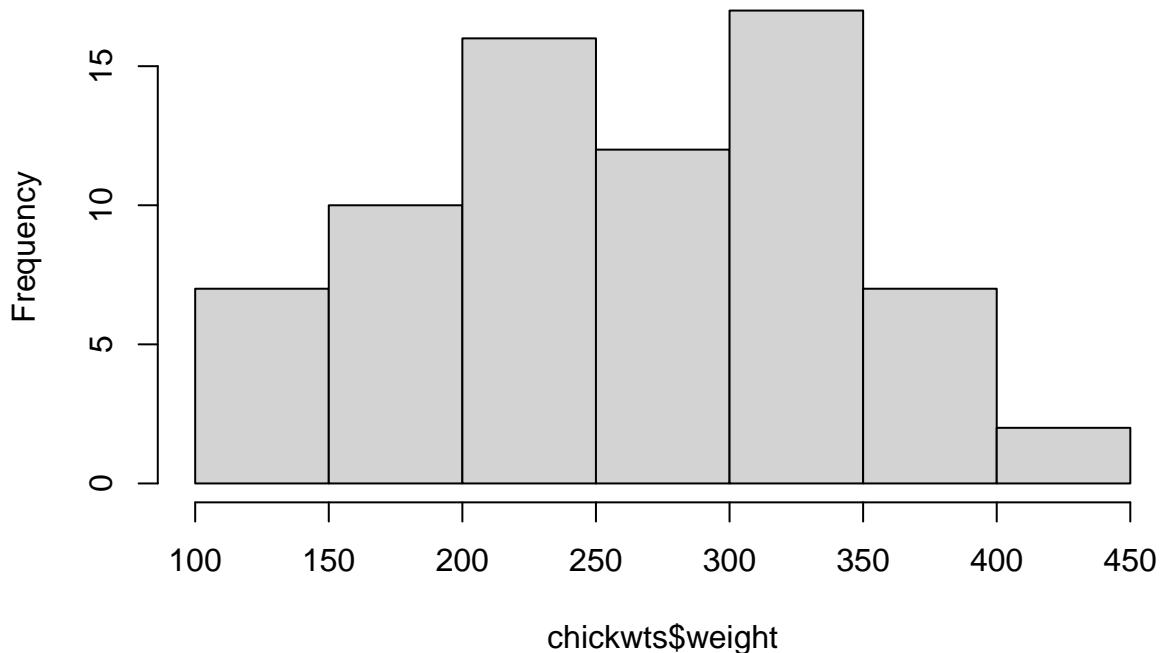
ggplot(data = chickwts, aes(x = weight)) +
  geom_histogram()
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
  
```



Otra manera de hacer el histograma es usando la paquetería básica de R:

```
hist(x = chickwts$weight)
```

Histogram of chickwts\$weight



Ambos métodos difieren en el tamaño de las barras. Independientemente de qué gráfica se decida usar, es esencial realizar la prueba de Shapiro mientras que la visualización de la distribución es algo complementario.

7.2 Prueba de homocedasticidad

La prueba de Bartlett para evaluar igualdad de varianzas se realiza con `bartlett.test()`, en la cual se deben poner los datos de la variable numérica en `x` y en `g` el vector con las categorías. Usando nuevamente `chickwts`, las categorías de cada observación se especifican en la variable `feed`:

```
head(chickwts)
```

```
##   weight      feed
## 1    179 horsebean
## 2    160 horsebean
## 3    136 horsebean
## 4    227 horsebean
## 5    217 horsebean
## 6    168 horsebean
```

Por lo tanto, la función debe escribirse así:

```
bartlett.test(x = chickwts$weight, g = chickwts$feed)

##
##  Bartlett test of homogeneity of variances
##
##  data:  chickwts$weight and chickwts$feed
##  Bartlett's K-squared = 3.2597, df = 5, p-value = 0.66
```

Para esta prueba igualmente se busca que el valor de `p` sea mayor a 0.05, debido a que la hipótesis nula es que las varianzas entre los grupos sea igual, mientras que la alternativa es que al menos la varianza de un grupo es diferente a la de los demás:

$$H_0 : \sigma_1^2 = \sigma_2^2 = \dots = \sigma_n^2$$

$$H_A : \sigma_1^2 \neq \sigma_2^2 = \sigma_3^2 \dots = \sigma_n^2$$

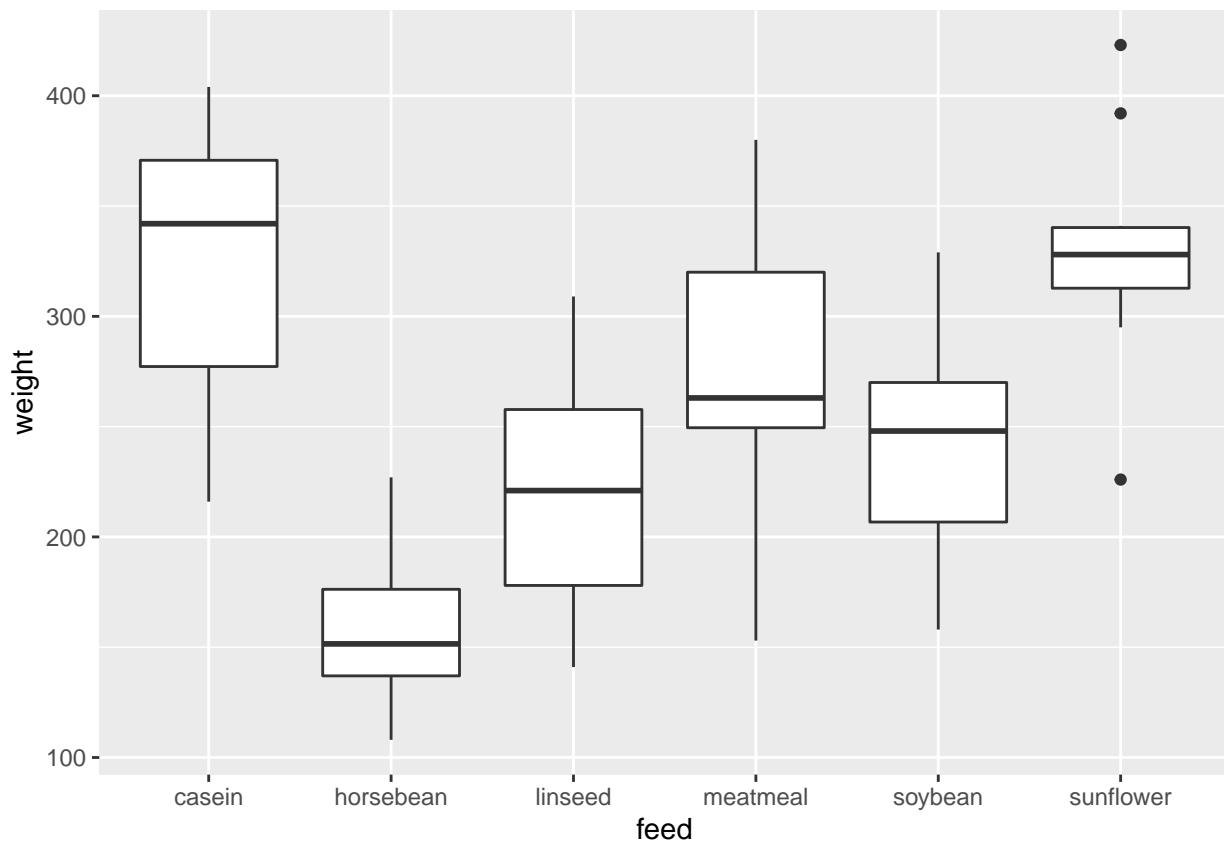
La función provee otra manera de introducir la información, en la cual se usan los argumentos `formula` y `data`, en el primero se escriben los nombres de las columnas del data frame que se va a utilizar, indicando los datos numéricos y las categorías de la siguiente manera: `datos_numericos ~ grupos`, donde el operador `~` se lee “en función de”. En el segundo argumento únicamente se indica el nombre del data frame:

```
bartlett.test(formula = weight ~ feed, data = chickwts)

##
##  Bartlett test of homogeneity of variances
##
##  data:  weight by feed
##  Bartlett's K-squared = 3.2597, df = 5, p-value = 0.66
```

Para corroborar la igualdad de varianzas visualmente se puede usar una gráfica de cajas y bigotes:

```
ggplot(data = chickwts, aes(x = feed, y = weight)) +
  geom_boxplot()
```



Usando esta gráfica no es posible ver el valor de la varianza, pero al comparar el rango de los datos de cada categoría se puede tener una idea preliminar de si hay o no igualdad de varianzas. Si el rango de todas las categorías es comparable, es decir, la distancia de un extremo a otro de los bigotes es similar entre todas las categorías, es más probable que exista homocedasticidad.

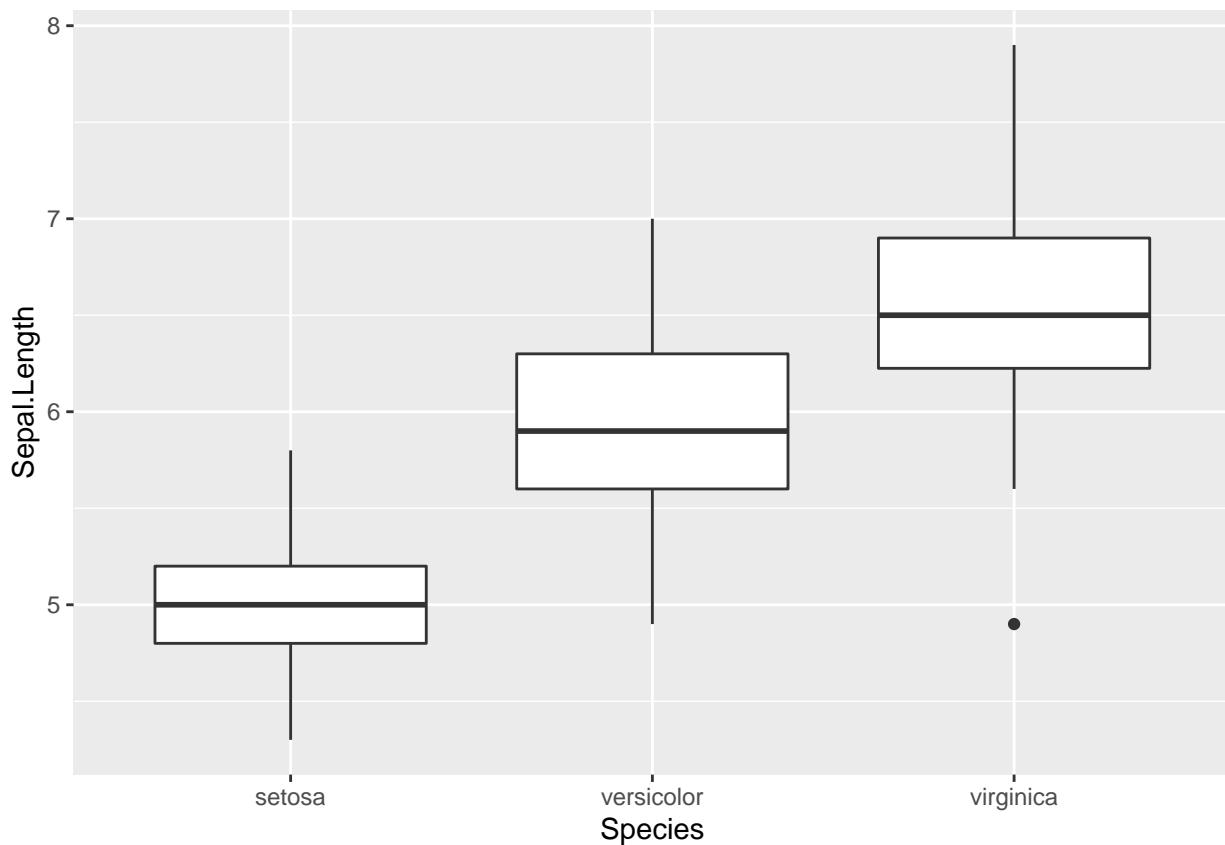
Sin embargo, la similitud visual que podamos encontrar no es una manera confiable de evaluar igualdad de varianzas. En el data frame `iris`, al realizar la prueba usando la longitud de sépalo de acuerdo con la especie, obtenemos que no hay igualdad de varianzas:

```
bartlett.test(formula = Sepal.Length ~ Species, data = iris)

##
##  Bartlett test of homogeneity of variances
##
## data: Sepal.Length by Species
## Bartlett's K-squared = 16.006, df = 2, p-value = 0.0003345
```

Visualmente, no es posible determinar si el rango entre una categoría y la otra es suficientemente diferente como para que las varianzas no sean iguales:

```
ggplot(data = iris, aes(x = Species, y = Sepal.Length)) +
  geom_boxplot()
```



Sin embargo, al realizar la gráfica es posible especular cuales podrían ser las razones de que no haya homocedasticidad. En este caso podrían ser los datos de *setosa* que tienen un rango más estrecho, o el dato extremo en *virginica*. De este modo podría identificarse algún error con el muestreo o alguna explicación de los datos atípicos y corregirlo, de ser posible, para obtener igualdad de varianzas. Pero este no siempre tiene que ser el caso, no todas los datos van a ajustarse a este y otros supuestos, pero al explorar los datos es posible obtener información que permita tomar decisiones para mejorar el diseño experimental o el muestreo.

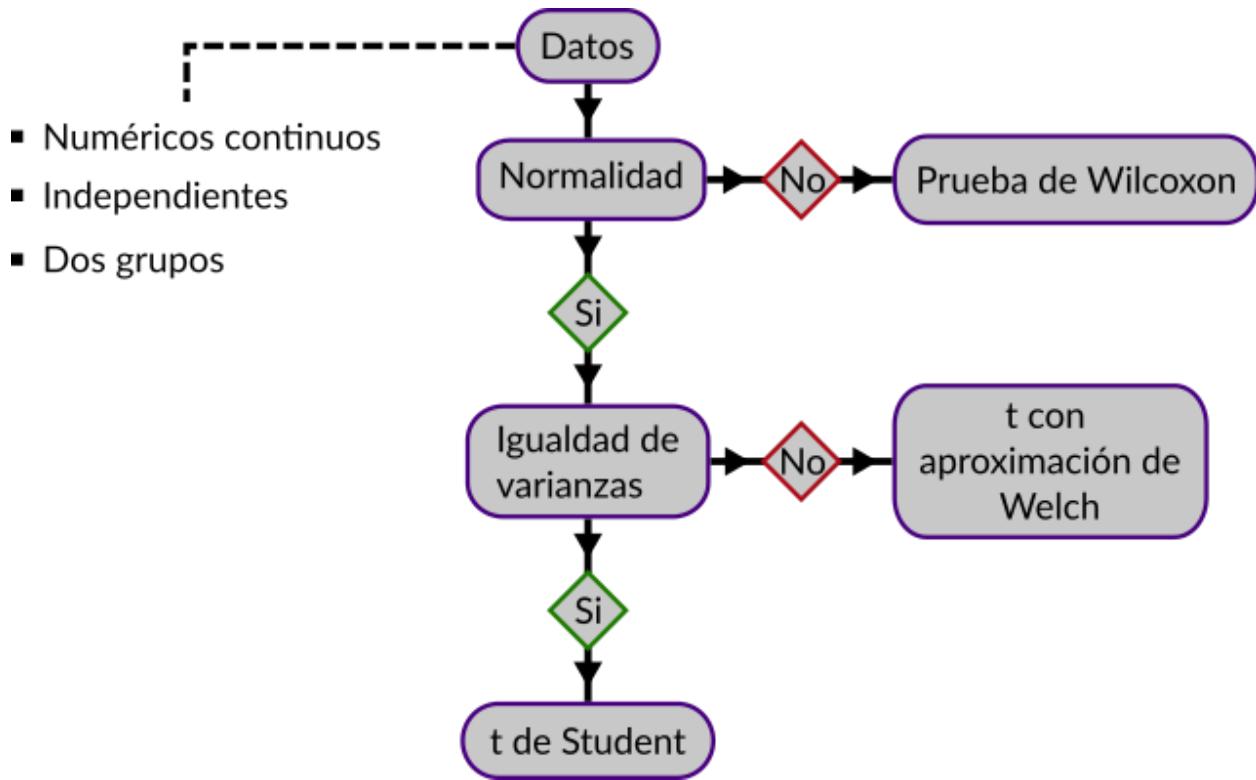
7.3 Comparación entre dos grupos

La prueba paramétrica para comparar dos grupos es la prueba de t. Su contraparte no paramétrica es la prueba de Wilcoxon (también conocida como la prueba de Mann-Whitney si es que se usa para dos grupos).

En esta sección se mostrarán las funciones paramétricas y no paramétricas para comparar dos grupos independientes, datos pareados y para realizar una comparación con la media de la población.

7.3.1 Prueba de t para grupos independientes

Asumiendo que se busca determinar si los datos cumplen con los supuestos para hacer una prueba de t, los pasos a seguir se muestran en el siguiente diagrama de flujo:



Se debe evaluar tanto normalidad como homocedasticidad, en caso de que no haya normalidad debe optarse por la prueba no paramétrica. Sin embargo, cuando los datos se distribuyen normalmente pero no hay igualdad de varianzas se puede realizar la prueba de t con aproximación de Welch. En caso de que se cumplan ambos supuestos se realiza la t de Student.

El siguiente conjunto de datos muestra la longitud de granos de maíz en milímetros para dos variedades (cónico y chapalote):

```

granos_maiz <- read_csv(file = "datos_manual/granos_maiz.csv",
                        col_types = "nf")
str(granos_maiz)

## spec_tbl_df [160 x 2] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
## $ longitud: num [1:160] 5.09 7.48 5.47 6.68 6.47 ...
## $ maiz     : Factor w/ 2 levels "conico","chapalote": 1 1 1 1 1 1 1 1 1 1 ...
## - attr(*, "spec")=
##   .. cols(
##     ..   longitud = col_number(),
##     ..   maiz = col_factor(levels = NULL, ordered = FALSE, include_na = FALSE)
##   .. )
## - attr(*, "problems")=<externalptr>

```

Para determinar si es que existe diferencia significativa entre los granos de ambas variedades, primero es necesario probar normalidad:

```

shapiro.test(x = granos_maiz$longitud)

##
## Shapiro-Wilk normality test
##
## data:  granos_maiz$longitud
## W = 0.99282, p-value = 0.6108

```

Debido a que el valor de p es mayor a 0.05, la distribución de los datos es aproximadamente normal. El siguiente paso consiste en determinar si hay igualdad de varianzas:

```
bartlett.test(formula = longitud~maiz, data = granos_maiz)

##
##  Bartlett test of homogeneity of variances
##
## data:  longitud by maiz
## Bartlett's K-squared = 18.335, df = 1, p-value = 1.852e-05
```

Debido a que no hay igualdad de varianza deberá usarse la aproximación de Welch. Para realizar la prueba de t se usa `t.test()`, la cual tiene los argumentos `formula`, `data`, `alternative` y `var.equal`. En `formula` y `data` se especifican los datos que se van a usar para la prueba. En `alternative` se especifica si la prueba será de una o dos colas y en `var.equal` se indica si se debe usar la aproximación de Welch. Por defecto, el nivel de confianza es 0.95, pero puede cambiarse al valor deseado modificando el argumento `conf.level`.

Por defecto, `alternative` siempre tomará el valor `two.sided` y se realizará una prueba de dos colas, en la cual únicamente se determina que hay diferencia significativa entre ambos grupos, sin especificar si uno es menor o mayor que el otro. Para hacer una prueba de una cola debe usarse `greater` o `less` para evaluar si la media de una de las poblaciones es mayor o menor, respectivamente. La elección de una prueba de una o dos colas determina las hipótesis nulas y alternativas.

Para una prueba de dos colas, la hipótesis nula es que la media del grupo 1 es igual a la media del grupo 2, mientras que la hipótesis nula es las medias son significativamente diferentes:

$$H_0 : \bar{x}_1 = \bar{x}_2$$

$$H_A : \bar{x}_1 \neq \bar{x}_2$$

Las pruebas de una cola tienen la misma hipótesis nula, solo cambian las hipótesis alternativas, en las cuales la media de la población 1 puede ser mayor o menor a la de la población 2.

Para la cola derecha (`greater`):

$$H_A : \bar{x}_1 > \bar{x}_2$$

Para la cola izquierda (`less`):

$$H_A : \bar{x}_1 < \bar{x}_2$$

Para una prueba de dos colas, la función se escribe así:

```
t.test(formula = longitud~maiz, data = granos_maiz,
       alternative = "two.sided", var.equal = FALSE)

##
##  Welch Two Sample t-test
##
## data:  longitud by maiz
## t = 4.5981, df = 130.76, p-value = 9.937e-06
## alternative hypothesis: true difference in means between group conico and group chapalote is not equal to zero
## 95 percent confidence interval:
##  0.4331836 1.0874008
## sample estimates:
##   mean in group conico mean in group chapalote
```

```
##                      5.973974                      5.213682
```

En la consola se imprime un pequeño reporte en el cual se indica que se trata de una prueba de t con aproximación de Welch. En el segundo renglón se muestra el valor del estadístico t, los grados de confianza y el valor de p, el cual en este caso es menor a 0.05, por lo cual se cumple la hipótesis alternativa y si existe diferencia significativa entre las medias de ambos grupos. En el siguiente renglón se muestra la hipótesis alternativa, la cual siempre se mostrará allí independientemente de que el valor de p sea o no significativo. Finalmente se muestran el intervalo de confianza y la media calculadas para cada grupo

Para realizar una prueba de una cola no es recomendable introducir los datos usando **formula** y **data**, ya que R automáticamente asigna los grupos 1 y 2 para la comparación realizada por **greater** y **less** de acuerdo con los nombres de las categorías. Esto implica menos control por parte del usuario y se presta a confusiones en cuanto a la dirección de la prueba ya que no hay ninguna indicación explícita de cómo está ocurriendo la comparación

Una prueba de t de una cola usando **greater** da el siguiente resultado:

```
t.test(formula = longitud_maiz, data = granos_maiz,
       alternative = "greater", var.equal = FALSE)

##
##  Welch Two Sample t-test
##
## data:  longitud by maiz
## t = 4.5981, df = 130.76, p-value = 4.968e-06
## alternative hypothesis: true difference in means between group conico and group chapalote is greater
## 95 percent confidence interval:
##  0.4863739      Inf
## sample estimates:
##   mean in group conico mean in group chapalote
##                 5.973974                  5.213682
```

La función no aclara qué grupo fue comparado respecto a cual, solo se sabe que el valor de p es significativo y por lo tanto la media del grupo 1 es mayor respecto a la del grupo 2. Lo mismo ocurre cuando se realiza la prueba en la dirección opuesta:

```
t.test(formula = longitud_maiz, data = granos_maiz,
       alternative = "less", var.equal = FALSE)

##
##  Welch Two Sample t-test
##
## data:  longitud by maiz
## t = 4.5981, df = 130.76, p-value = 1
## alternative hypothesis: true difference in means between group conico and group chapalote is less than
## 95 percent confidence interval:
##      -Inf 1.03421
## sample estimates:
##   mean in group conico mean in group chapalote
##                 5.973974                  5.213682
```

La única diferencia consiste en que el valor de p no es significativo, pero nuevamente no es claro cómo es que se están comparando ambos grupos. Sin embargo, las medias calculadas por las pruebas dejan claro que la función tomó como grupo 1 a los datos de **conico** y grupo 2 a los de **chapalote**.

Este comportamiento complica innecesariamente el procedimiento para realizar la prueba, ya que en ningún momento hay una indicación explícita sobre cómo se establece la hipótesis alternativa y solo puede conocerse el orden de la comparación con información que se obtiene hasta después de realizar la prueba. Una manera de solucionar esto es usando otro método que provee **t.test()**, en el cual se introducen los datos numéricos

de los dos grupos como vectores en los argumentos `x` y `y`, lo cual requiere que los datos estén en formato ancho para tener los datos de cada tipo de maíz en vectores separados:

```
granos_maiz_wide <- granos_maiz %>%
  group_by(maiz) %>%
  mutate(filas = row_number()) %>%
  pivot_wider(values_from = longitud, names_from = maiz)
granos_maiz_wide

## # A tibble: 80 x 3
##   filas conico chapalote
##   <int>   <dbl>     <dbl>
## 1     1     5.09     5.85
## 2     2     7.48     7.03
## 3     3     5.47     7.92
## 4     4     6.68     7.41
## 5     5     6.47     4.72
## 6     6     5.43     6.30
## 7     7     5.36     7.47
## 8     8     6.51     6.06
## 9     9     6.50     6.00
## 10    10    5.68     3.76
## # ... with 70 more rows
```

De este modo los datos pueden accederse de acuerdo con el tipo de maíz usando el operador `$`. Con la necesidad de esta transformación adicional podría uno cuestionarse cual es la mejor manera de introducir los datos. Mi recomendación es siempre empezar con los datos en formato largo y después hacer las transformaciones necesarias en R. Sobre todo si se busca realizar una gráfica, ya que `ggplot()` requiere los datos en formato largo.

De este modo, si se desea probar la hipótesis alterna de que la longitud de los granos de la variedad chapalote es menor a los de la variedad cónico:

$$H_A : \bar{t}_{\text{chapalote}} < \bar{t}_{\text{conico}}$$

La función es la siguiente:

```
t.test(x = granos_maiz_wide$chapalote, y = granos_maiz_wide$conico,
       alternative = "less", var.equal = FALSE)

##
## Welch Two Sample t-test
##
## data:  granos_maiz_wide$chapalote and granos_maiz_wide$conico
## t = -4.5981, df = 130.76, p-value = 4.968e-06
## alternative hypothesis: true difference in means is less than 0
## 95 percent confidence interval:
##       -Inf -0.4863739
## sample estimates:
## mean of x mean of y
## 5.213682 5.973974
```

El resultado de la función tiene un formato ligeramente diferente, pero todavía muestra los mismos componentes. Debido a que el valor de `p` es menor a 0.05, se cumple la hipótesis alternativa y la longitud de grano del maíz chapalote es menor que la de cónico.

Usando los argumentos `x` y `y` es más sencillo establecer los argumentos de acuerdo con la hipótesis que tenemos en mente, en lugar de ajustarnos al comportamiento automático de la función.

En caso de que los datos obtenidos si tengan igualdad de varianzas, lo único que se debe cambiar de la función es `var.equal`:

```
t.test(x = granos_maiz_wide$chapalote, y = granos_maiz_wide$conico,
       alternative = "less", var.equal = TRUE)

##
## Two Sample t-test
##
## data:  granos_maiz_wide$chapalote and granos_maiz_wide$conico
## t = -4.5981, df = 158, p-value = 4.344e-06
## alternative hypothesis: true difference in means is less than 0
## 95 percent confidence interval:
##       -Inf -0.4867104
## sample estimates:
## mean of x mean of y
## 5.213682 5.973974
```

Por defecto, `var.equal` siempre es falso y se realiza la prueba con el ajuste de Welch. De cualquier modo, es recomendable siempre mostrarlo explícitamente para evitar ambigüedades en el método que se sigue para cada paso del análisis.

Si se desea acceder a los diferentes valores calculados por la prueba, es posible guardar el resultado de la función como un objeto y usar el operador `$` para acceder a los parámetros. Los componentes de la prueba pueden consultarse en la documentación (`?t.test()`), bajo el apartado Value:

```
prueba_t <- t.test(x = maiz_chapalote$longitud, y = maiz_conico$longitud,
                     alternative = "less", var.equal = FALSE)

## Error in t.test(x = maiz_chapalote$longitud, y = maiz_conico$longitud, : object 'maiz_chapalote' not
prueba_t$conf.int

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

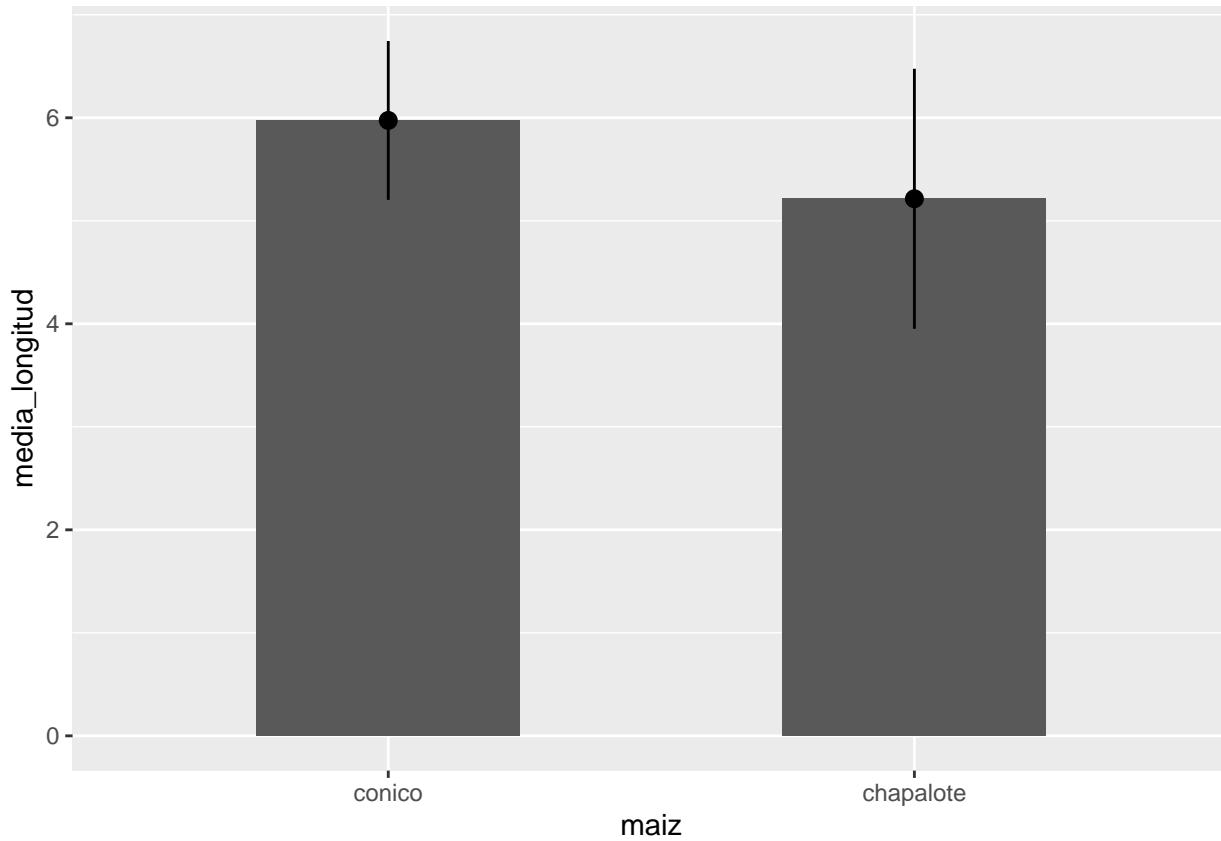
Debido a que la prueba de t es una comparación de medias, los resultados de la prueba pueden acompañarse visualmente con una gráfica de barras que muestre la media de cada grupo con barras de error que indiquen la desviación estándar.

```
medias_maiz <- granos_maiz %>%
  group_by(maiz) %>%
  summarize(media_longitud = mean(longitud),
            desv_longitud = sd(longitud))

grafica_maiz

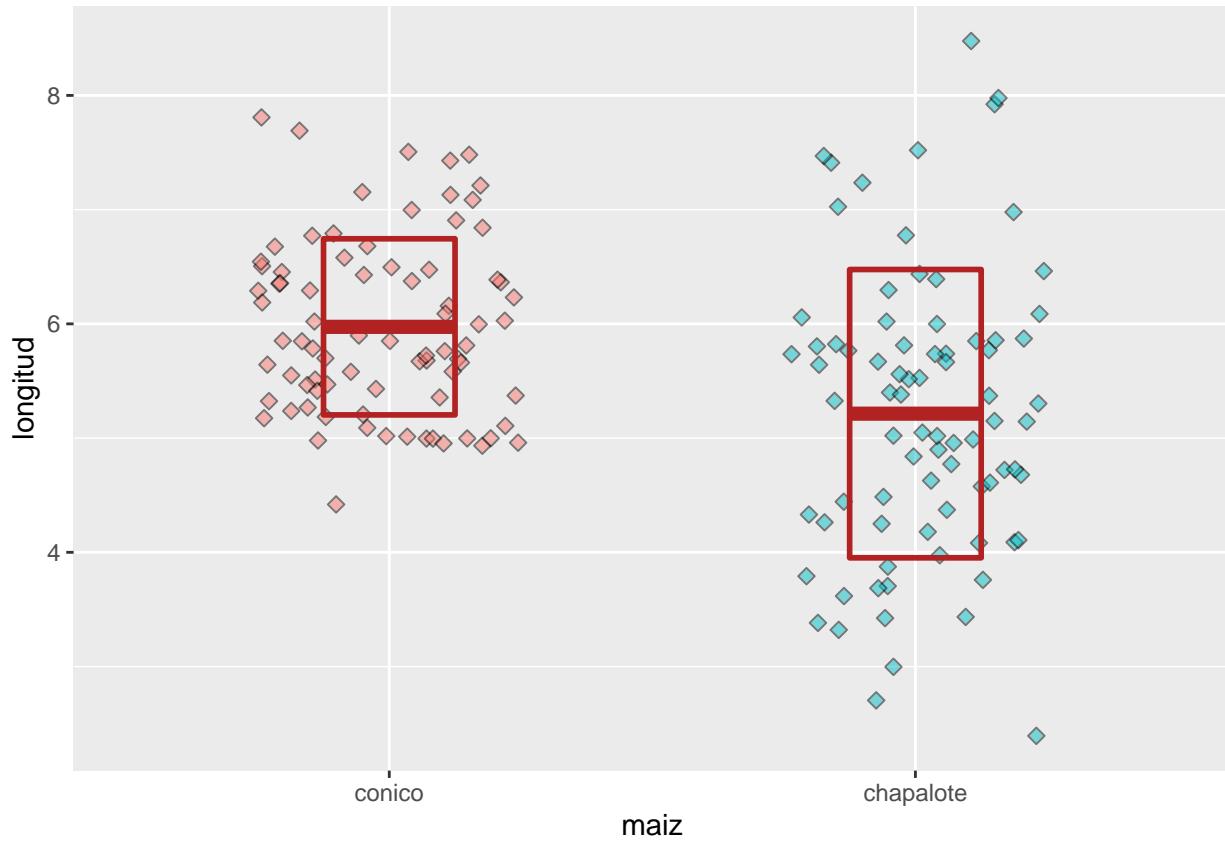
## Error in eval(expr, envir, enclos): object 'grafica_maiz' not found

ggplot(data = medias_maiz, aes(x = maiz, y = media_longitud)) +
  geom_col(position = "dodge", width = 1/2) +
  geom_pointrange(aes(ymin = media_longitud - desv_longitud,
                        ymax = media_longitud + desv_longitud))
```



Esta gráfica tiene la ventaja de mostrar los valores calculados por el análisis, particularmente la media e indirectamente la desviaciones estándar ya que esta última se usa para calcular el error estándar. Sin embargo, no permite observar la distribución de los datos, lo cual puede realizarse usando `geom_jitter()` y para preservar la media y la desviación estándar se usa `geom_crossbar()`:

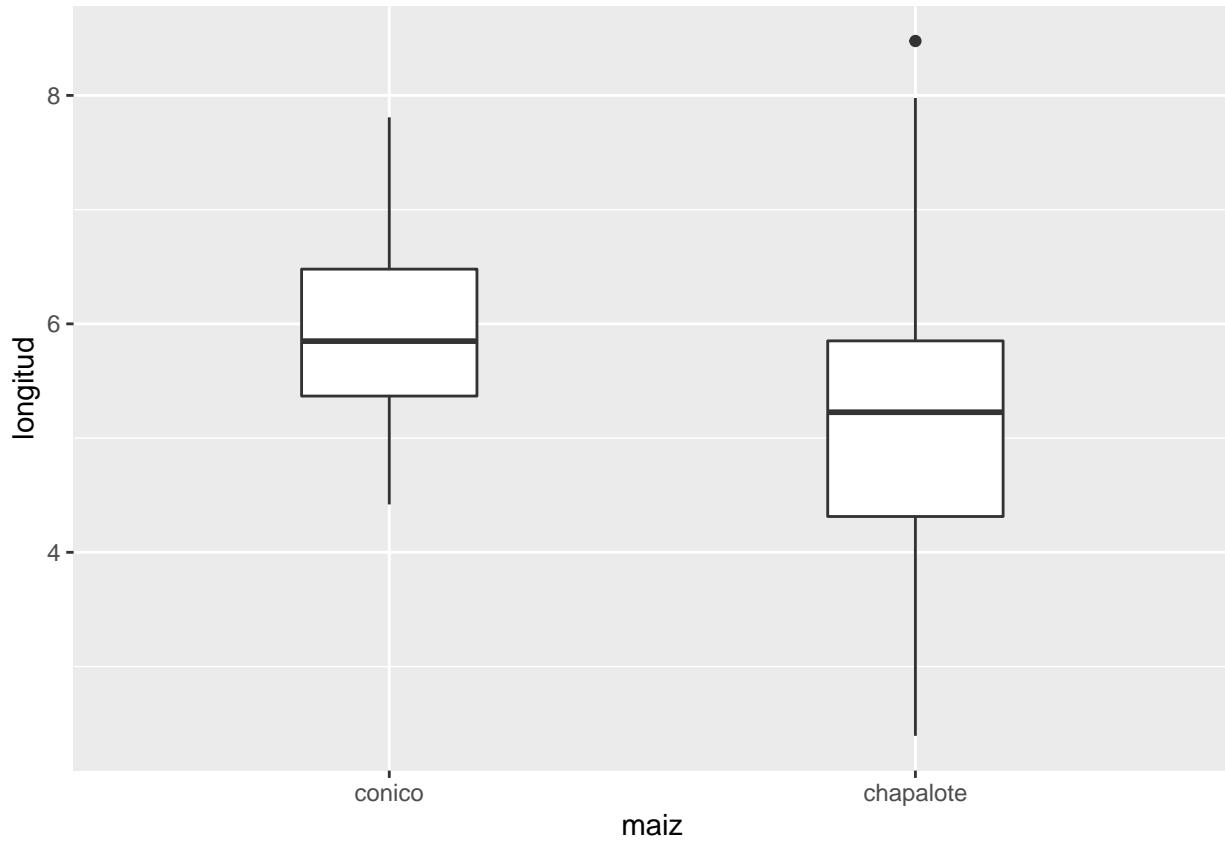
```
ggplot() +
  geom_jitter(data = granos_maiz, aes(x = maiz, y = longitud, fill = maiz),
              width = 1/4, pch = 23, size = 2, alpha = 1/2) +
  geom_crossbar(data = medias_maiz, aes(x = maiz, y = media_longitud,
                                         ymin = media_longitud - desv_longitud,
                                         ymax = media_longitud + desv_longitud),
                width = 1/4, color = "firebrick", size = 1) +
  theme(legend.position = "none")
```



Una ventaja de usar `geom_jitter()` es que es que al mostrar observaciones individuales es posible determinar si es que la media de uno de los grupos es resultado de muy pocas observaciones. En este caso, ambos grupos cuentan con 80.

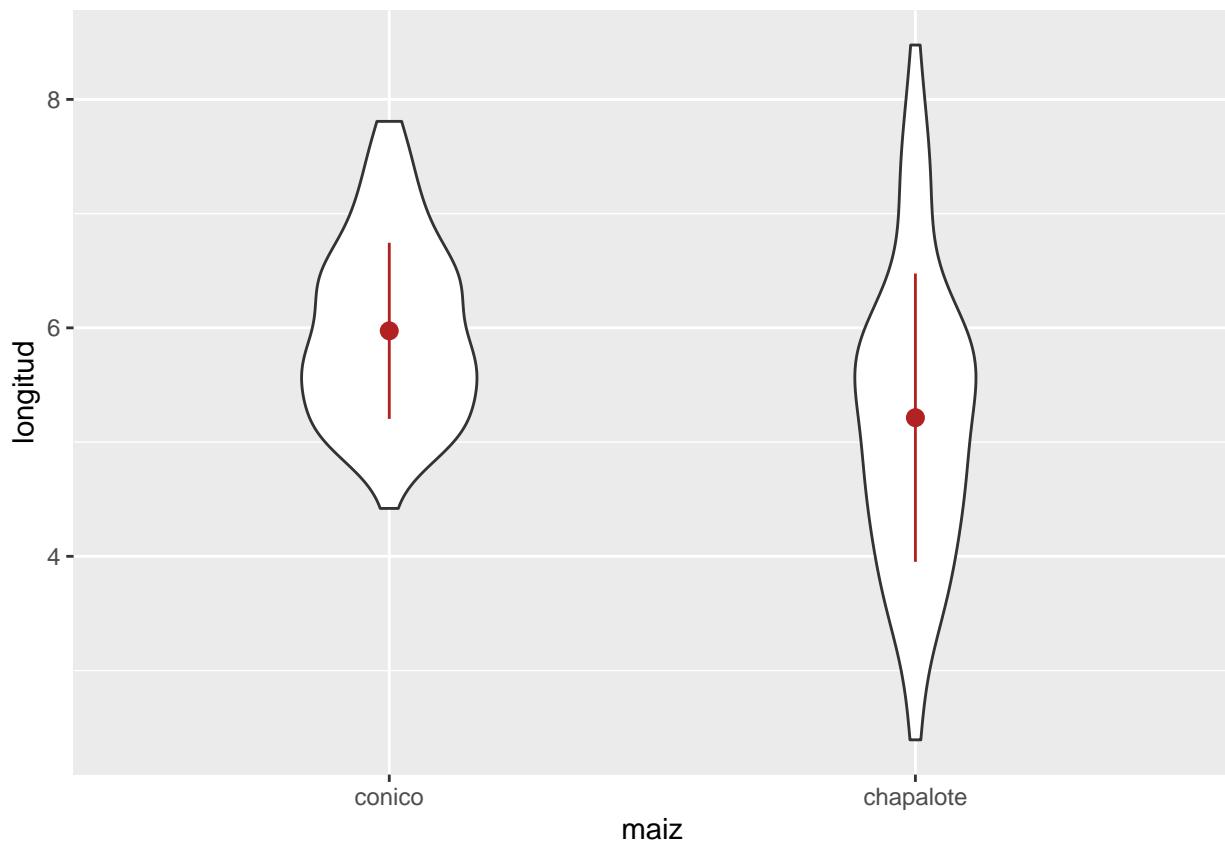
Otra manera podría ser usando una gráfica de cajas y bigotes:

```
ggplot(data = granos_maiz, aes(x = maiz, y = longitud)) +
  geom_boxplot(width = 1/3)
```



El problema con esta gráfica consiste en que no se muestran los valores relevantes para el análisis, solo es posible ver el rango y la mediana. Una gráfica de violín puede ser más adecuada ya que da una representación gráfica de la distribución de los datos (sin mostrar observaciones individuales), y puede ser complementada con `geom_pointrange()` para mostrar las medias y desviaciones estándar:

```
ggplot() +
  geom_violin(data = granos_maiz, aes(x = maiz, y = longitud), width = 1/3) +
  geom_pointrange(data = medias_maiz, aes(x = maiz, y = media_longitud,
                                           ymin = media_longitud - desv_longitud,
                                           ymax = media_longitud + desv_longitud),
                  color = "firebrick")
```



La elección de la gráfica dependerá de lo que se quiera comunicar y los requerimientos de la audiencia a quienes se le vaya a presentar.

7.3.2 Prueba de Wilcoxon para grupos independientes

El siguiente conjunto de datos es similar al anterior, pero no cumple con los supuestos requeridos para la prueba de t:

```
granos_maiz_2 <- read_csv(file = "datos_manual/granos_maiz_2.csv",
                           col_types = "nf")

## Error: 'datos_manual/granos_maiz_2.csv' does not exist in current working directory ('/home/leot/Docu
granos_maiz_2

## Error in eval(expr, envir, enclos): object 'granos_maiz_2' not found
shapiro.test(x = granos_maiz_2$longitud)

## Error in stopifnot(is.numeric(x)): object 'granos_maiz_2' not found
```

Debido a que el valor de p es menor a .05, la distribución de los datos no es aproximadamente normal.

```
hist(x = granos_maiz_2$longitud)

## Error in hist(x = granos_maiz_2$longitud): object 'granos_maiz_2' not found
```

La prueba de Wilcoxon se realiza con la función `wilcox.test` del mismo modo que la de t ya que cuentan con argumentos similares. Se puede usar tanto `formula` y `data` como `x` y `y` para introducir los datos. El argumento `alternative` permite seleccionar entre una y dos colas. El nivel de confianza por defecto es 0.95 y puede ser modificado mediante `conf.level`.

La prueba de Wilcoxon no compara medias sino medianas, por lo tanto las hipótesis son las siguientes para las pruebas de dos y una cola:

$$H_0 : Md_1 = Md_2$$

$$H_A : Md_1 \neq Md_2$$

$$H_A : Md_1 > Md_2$$

$$H_A : Md_1 < Md_2$$

Para una prueba de dos colas, el código necesario es el siguiente:

```
wilcox.test(formula = longitud ~ maiz, data = granos_maiz_2,
            alternative = "two.sided")
## Error in eval(m$data, parent.frame()): object 'granos_maiz_2' not found
```

En el resultado se puede observar que solo se calcula el estadístico W y el valor de p. Debido a que el valor de p es menor a 0.05, la hipótesis alternativa de que ambos grupos tienen medianas diferentes se cumple. La función describe esta hipótesis como un cambio en posición entre los grupos.

Las pruebas de una cola también requieren los datos en formato ancho:

```
granos_maiz_2_wide <- granos_maiz_2 %>%
  group_by(maiz) %>%
  mutate(fila = row_number()) %>%
  pivot_wider(names_from = maiz, values_from = longitud)
## Error in group_by(., maiz): object 'granos_maiz_2' not found
granos_maiz_2_wide
## Error in eval(expr, envir, enclos): object 'granos_maiz_2_wide' not found
```

Para probar la hipótesis de que los granos del maíz cónico son menores a los del maíz chapalote:

```
wilcox.test(x = granos_maiz_2_wide$conico, y = granos_maiz_2_wide$chapalote,
            alternative = "less")
## Error in wilcox.test(x = granos_maiz_2_wide$conico, y = granos_maiz_2_wide$chapalote, : object 'gran
```

En este caso la hipótesis alternativa no se cumple porque el valor de p no es significativo. La prueba en la dirección opuesta si es significativa:

```
wilcox.test(x = granos_maiz_2_wide$conico, y = granos_maiz_2_wide$chapalote,
            alternative = "greater")
## Error in wilcox.test(x = granos_maiz_2_wide$conico, y = granos_maiz_2_wide$chapalote, : object 'gran
```

Para mostrar los resultados de esta comparación no es adecuado usar una gráfica de barras con las medias y desviación estándar, es más conveniente usar una gráfica de cajas y bigotes para claramente ver las medianas que están siendo comparadas por las pruebas:

```
ggplot(data = granos_maiz_2, aes(x = maiz, y = longitud)) +
  geom_boxplot(width = 1/3)
## Error in ggplot(data = granos_maiz_2, aes(x = maiz, y = longitud)): object 'granos_maiz_2' not found
```

```

ggplot(data = granos_maiz_2, aes(x = maiz, y = longitud)) +
  geom_jitter(width = 1/5, aes(fill = maiz), pch = 23, size = 2,
              alpha = 1/2) +
  geom_boxplot(width = 1/3, size = 1, fill = "transparent", color = "sienna4") +
  theme(legend.position = "none")

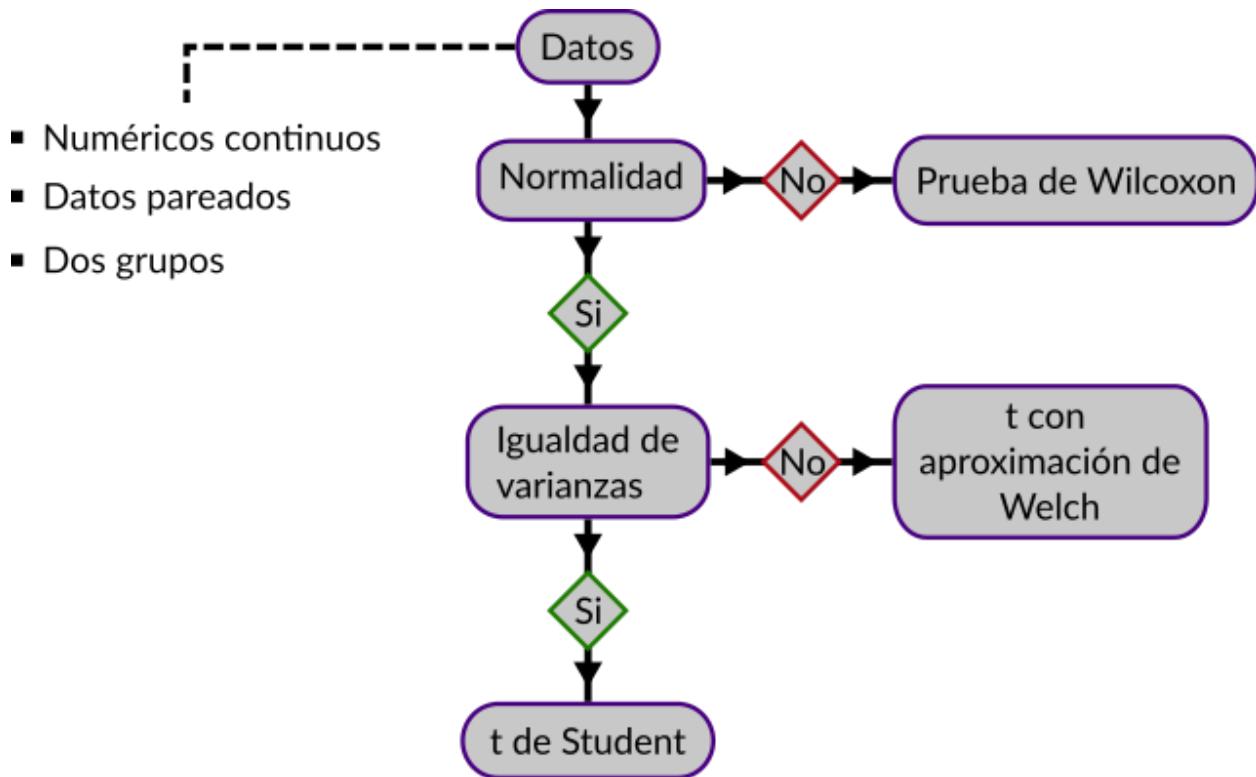
## Error in ggplot(data = granos_maiz_2, aes(x = maiz, y = longitud)): object 'granos_maiz_2' not found

```

7.3.3 Prueba de t para muestras pareadas

Cuando los datos son pareados, es decir, provienen de la misma unidad de estudio pero en momentos diferentes, los datos no son independientes, pero pueden ser comparados para determinar si alguna la variable medida muestra un cambio significativo entre el tiempo inicial y el final.

El diagrama de flujo para esta prueba es el mismo que para muestras independientes, solo cambian las características de los datos.



Un ejemplo de datos pareados es el siguiente data frame, en el cual se muestran los pesos de 110 conejos antes y después de ser expuestos a un tratamiento:

```

pesos_conejos <- read_csv(file = "datos_manual/pesos_conejos.csv",
                           col_types = "fnn")
pesos_conejos

## # A tibble: 220 x 3
##   conejo tiempo     peso
##   <fct>  <fct>   <dbl>
## 1 1      antes    4.13
## 2 1      despues  4.85
## 3 2      antes    3.77
## 4 2      despues  5.01
## 5 3      antes    4.00

```

```

## 6 3      despues 4.28
## 7 4      antes   4.04
## 8 4      despues 4.67
## 9 5      antes   3.5
## 10 5     despues 5.00
## # ... with 210 more rows

```

La prueba de normalidad para datos pareados no se realiza sobre la totalidad de los datos, sino sobre las diferencias entre ambos momentos del muestreo. En este caso es conveniente tener los datos en formato ancho:

```

pesos_conejos_wide <- pesos_conejos %>%
  pivot_wider(names_from = tiempo, values_from = peso)
pesos_conejos_wide

## # A tibble: 110 x 3
##   conejo antes despues
##   <fct>  <dbl>   <dbl>
## 1 1       4.13    4.85
## 2 2       3.77    5.01
## 3 3       4.00    4.28
## 4 4       4.04    4.67
## 5 5       3.5     5.00
## 6 6       3.01    4.42
## 7 7       3.70    4.85
## 8 8       3.95    4.54
## 9 9       1.95    4.73
## 10 10    3.15    4.81
## # ... with 100 more rows

```

Para realizar la conversión no es necesario crear una nueva variable con el número de filas porque cada observación es una combinación única de variables. Usando `mutate()` se puede crear una nueva columna con la diferencia entre ambos tiempos:

```

pesos_conejos_wide <- pesos_conejos_wide %>%
  mutate(diferencia = antes - despues)
pesos_conejos_wide

## # A tibble: 110 x 4
##   conejo antes despues diferencia
##   <fct>  <dbl>   <dbl>     <dbl>
## 1 1       4.13    4.85    -0.728
## 2 2       3.77    5.01    -1.24
## 3 3       4.00    4.28    -0.286
## 4 4       4.04    4.67    -0.628
## 5 5       3.5     5.00    -1.50
## 6 6       3.01    4.42    -1.42
## 7 7       3.70    4.85    -1.15
## 8 8       3.95    4.54    -0.594
## 9 9       1.95    4.73    -2.78
## 10 10    3.15    4.81    -1.66
## # ... with 100 more rows

```

Usando este data frame se puede probar normalidad:

```

shapiro.test(x = pesos_conejos_wide$diferencia)

##

```

```

## Shapiro-Wilk normality test
##
## data: pesos_conejos_wide$diferencia
## W = 0.98951, p-value = 0.5569

```

La prueba de Bartlett requiere los datos en formato largo ya que es necesario una columna que identifique a cada una de las observaciones de manera individual:

```

bartlett.test(formula = peso ~ tiempo, data = pesos_conejos)

##
## Bartlett test of homogeneity of variances
##
## data: peso by tiempo
## Bartlett's K-squared = 83.454, df = 1, p-value < 2.2e-16

```

Debido a que no hay homogeneidad de varianzas es necesario usar indicar el ajuste de Welch en la función. Para hacer la prueba con datos pareados solo se debe indicar en `t.test()` que el valor del argumento `paired` es verdadero:

```

t.test(formula = peso ~ tiempo, data = pesos_conejos_2, alternative = "two.sided",
       var.equal = FALSE, paired = TRUE)

## Error in eval(m$data, parent.frame()): object 'pesos_conejos_2' not found

```

El valor de `p` es significativo y como la prueba es de dos colas, se prueba que hay diferencia significativa en el peso entre tiempo inicial y el final. Para las pruebas de una cola es más conveniente usar el formato ancho e igualmente solo se debe modificar `paired`:

```

t.test(x = pesos_conejos_wide$antes, y = pesos_conejos_wide$despues,
       alternative = "less", var.equal = FALSE, paired = TRUE)

##
## Paired t-test
##
## data: pesos_conejos_wide$antes and pesos_conejos_wide$despues
## t = -18.005, df = 109, p-value < 2.2e-16
## alternative hypothesis: true mean difference is less than 0
## 95 percent confidence interval:
##      -Inf -1.330734
## sample estimates:
## mean difference
##              -1.465791

```

De acuerdo con esta prueba, el peso de los conejos antes del tratamiento es significativamente menor que después de él. Un ejemplo de una gráfica para mostrar los datos es el siguiente:

```

pesos_conejos_medias <- pesos_conejos %>%
  group_by(tiempo) %>%
  summarize(media_peso = mean(peso),
            desv_peso = sd(peso))

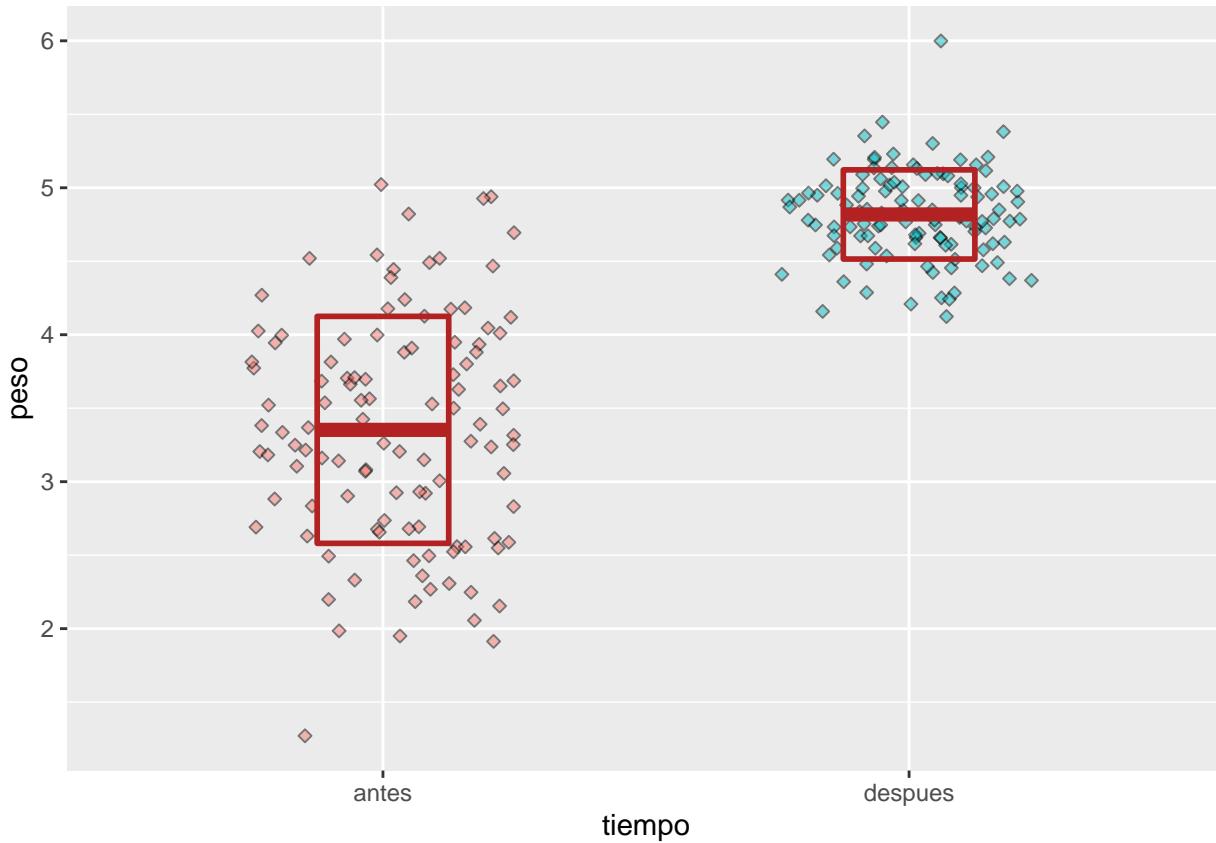
ggplot() +
  geom_jitter(data = pesos_conejos, aes(x = tiempo, y = peso,
                                         fill = tiempo),
              pch = 23, width = 1/4, alpha = 1/2) +
  geom_crossbar(data = pesos_conejos_medias, aes(x = tiempo, y = media_peso,
                                                 ymin = media_peso - desv_peso,
                                                 ymax = media_peso + desv_peso),
                color = "black")

```

```

width = 1/4, color = "firebrick", size = 1) +
theme(legend.position = "none")

```



7.3.4 Prueba de Wilcoxon para muestras pareadas

Para realizar esta prueba solo es necesario indicar que `paired` es verdadero.

```

wilcox.test(formula = peso ~ tiempo, data = pesos_conejos, paired = TRUE,
            alternative = "two.sided")

##
##  Wilcoxon signed rank test with continuity correction
##
##  data:  peso by tiempo
##  V = 26, p-value < 2.2e-16
##  alternative hypothesis: true location shift is not equal to 0

```

7.3.5 Prueba de t para una muestra

En estos casos solo se tienen los datos de un grupo y la media del grupo con el que se desea comparar. Para esta prueba solo es necesario probar normalidad debido a que no hay grupo con el cual comparar la varianza:

El data frame `trees` contiene información relacionada con la diámetro (in) (incorrectamente indicado como Girth (circunferencia) de acuerdo con la documentación), altura (ft) y volumen (ft3) de árboles de cereza. Asumiendo que otro conjunto de árboles tiene un diámetro promedio de 14.7 in y se desea determinar si es que los datos contenidos en `trees` son significativamente diferentes a esa media, primero es necesario probar normalidad:

```
shapiro.test(x = trees$Girth)
```

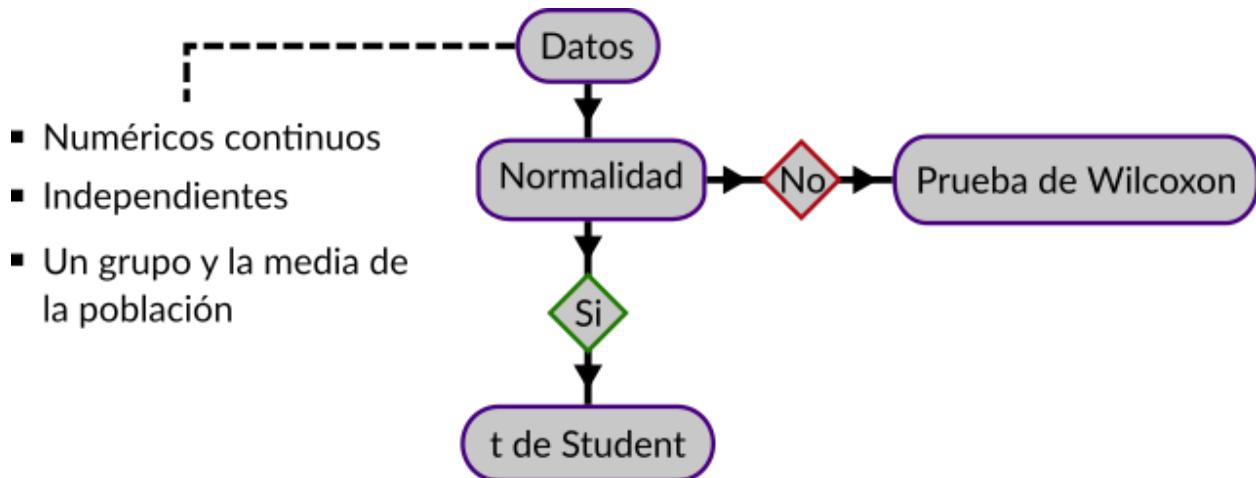


Figure 5: Vista inicial de RStudio

```

##  

## Shapiro-Wilk normality test  

##  

## data: trees$Girth  

## W = 0.94117, p-value = 0.08893

```

En `t.test()` solo debe de usarse el argumento `x` para introducir los datos y se debe especificar el valor de la media con el que se va a comparar usando `mu`:

```

t.test(x = trees$Girth, mu = 14.7, alternative = "two.sided")  

##  

## One Sample t-test  

##  

## data: trees$Girth  

## t = -2.5755, df = 30, p-value = 0.01518  

## alternative hypothesis: true mean is not equal to 14.7  

## 95 percent confidence interval:  

## 12.09731 14.39947  

## sample estimates:  

## mean of x  

## 13.24839

```

En el caso de las pruebas de una cola, `x` siempre será tratado como el grupo 1 y `mu` como el grupo 2:

```

t.test(x = trees$Girth, mu = 14.7, alternative = "less")  

##  

## One Sample t-test  

##  

## data: trees$Girth  

## t = -2.5755, df = 30, p-value = 0.007591  

## alternative hypothesis: true mean is less than 14.7  

## 95 percent confidence interval:  

## -Inf 14.20501  

## sample estimates:  

## mean of x  

## 13.24839

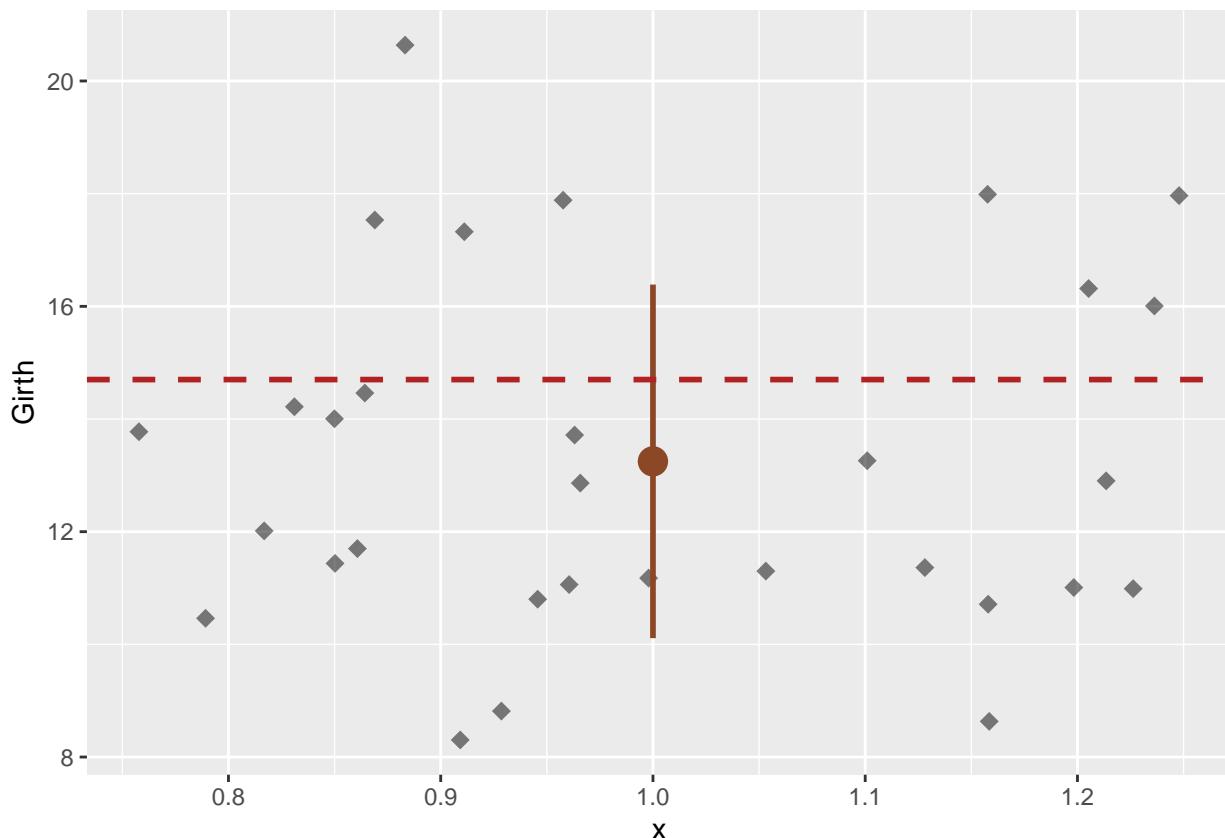
```

Esta prueba puede ser graficada de manera similar a las anteriores, pero debido a que el grupo con el que se está comparando es solo un valor (la media) se puede usar una linea horizontal para indicar este valor único:

```
trees_medias <- trees %>%
  summarize(media_diam = mean(Girth),
            desv_diam = sd(Girth))
trees_medias

##   media_diam desv_diam
## 1    13.24839  3.138139

ggplot() +
  geom_jitter(data = trees, aes(x = 1, y = Girth), width = 1/4,
              pch = 18, alpha = 1/2, size = 3) +
  geom_pointrange(data = trees_medias, aes(y = media_diam, x = 1,
                                             ymin = media_diam - desv_diam,
                                             ymax = media_diam + desv_diam),
                  color = "sienna4", size = 1) +
  geom_hline(yintercept = 14.7, color = "firebrick", size = 1, linetype = 2)
```



7.3.6 Prueba de Wilcoxon para una muestra

El procedimiento es el mismo, solo que existe una pequeña ambigüedad en los argumentos, ya que la prueba de Wilcoxon compara medianas, por lo que debe usarse una mediana para hacer la comparación, sin embargo el nombre del argumento sigue siendo `mu`.

Los datos de `trees` relacionados con el volumen no se distribuyen normalmente:

```
shapiro.test(trees$Volume)
```

```

##  

## Shapiro-Wilk normality test  

##  

## data: trees$Volume  

## W = 0.88757, p-value = 0.003579

```

Debido a esto es necesario hacer una prueba de Wilcoxon. Asumiendo que la mediana del volumen de otro grupo de árboles es 19.4 ft³, la función para determinar si es que la mediana de los datos de `trees` es mayor es la siguiente:

```

wilcox.test(x = trees$Volume, mu = 19.4, alternative = "greater")  

## Warning in wilcox.test.default(x = trees$Volume, mu = 19.4, alternative =  

## "greater"): cannot compute exact p-value with ties  

##  

## Wilcoxon signed rank test with continuity correction  

##  

## data: trees$Volume  

## V = 410, p-value = 0.0007756  

## alternative hypothesis: true location is greater than 19.4

```

En este caso el valor de p es significativo, por lo cual la mediana de los valores de `trees` es mayor. Para graficar esta prueba se puede hacer algo similar a la prueba de t, pero usando cajas y bigotes:

```

ggplot(data = trees, aes(x = 1, y = Volume)) +  

  geom_boxplot(width = 1/5, fill = "transparent",  

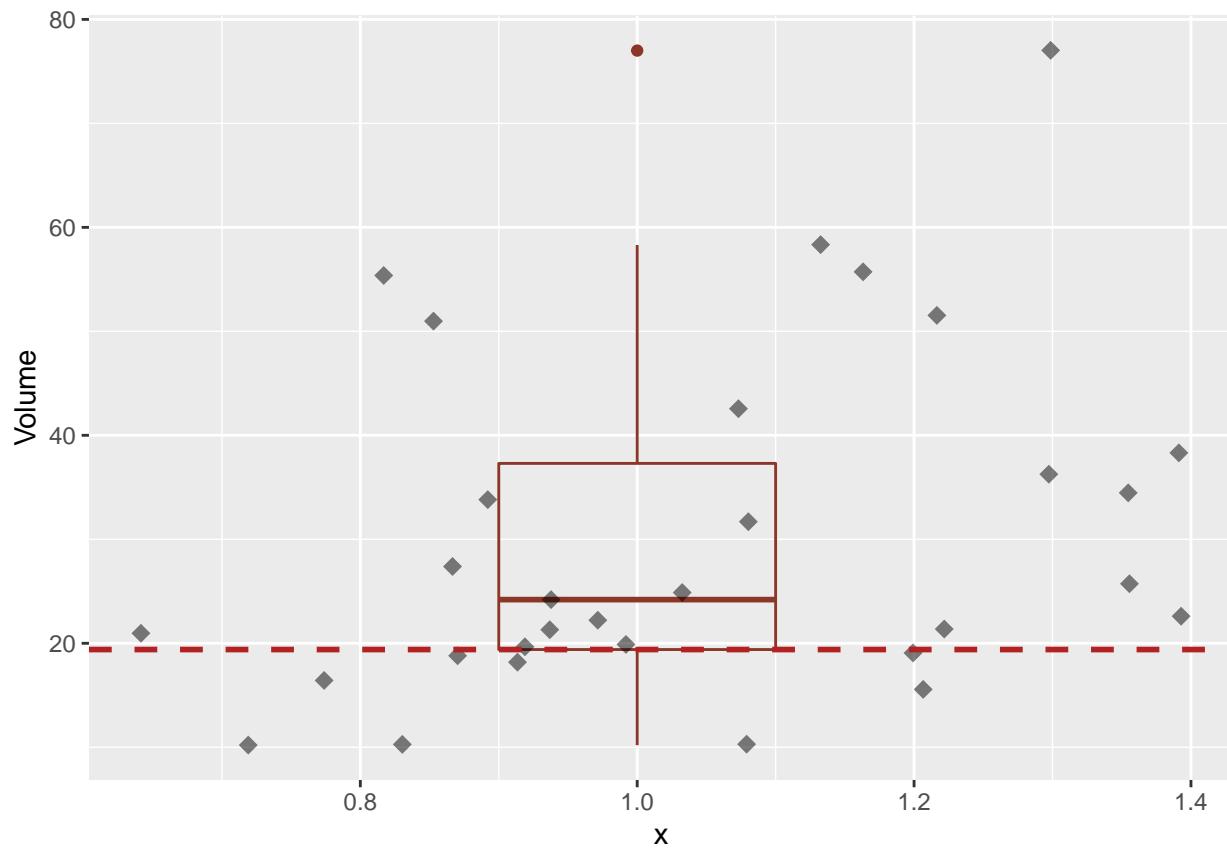
               color = "tomato4") +  

  geom_jitter(pch = 18, size = 3, alpha = 1/2) +  

  geom_hline(yintercept = 19.4, linetype = 2, color = "firebrick",  

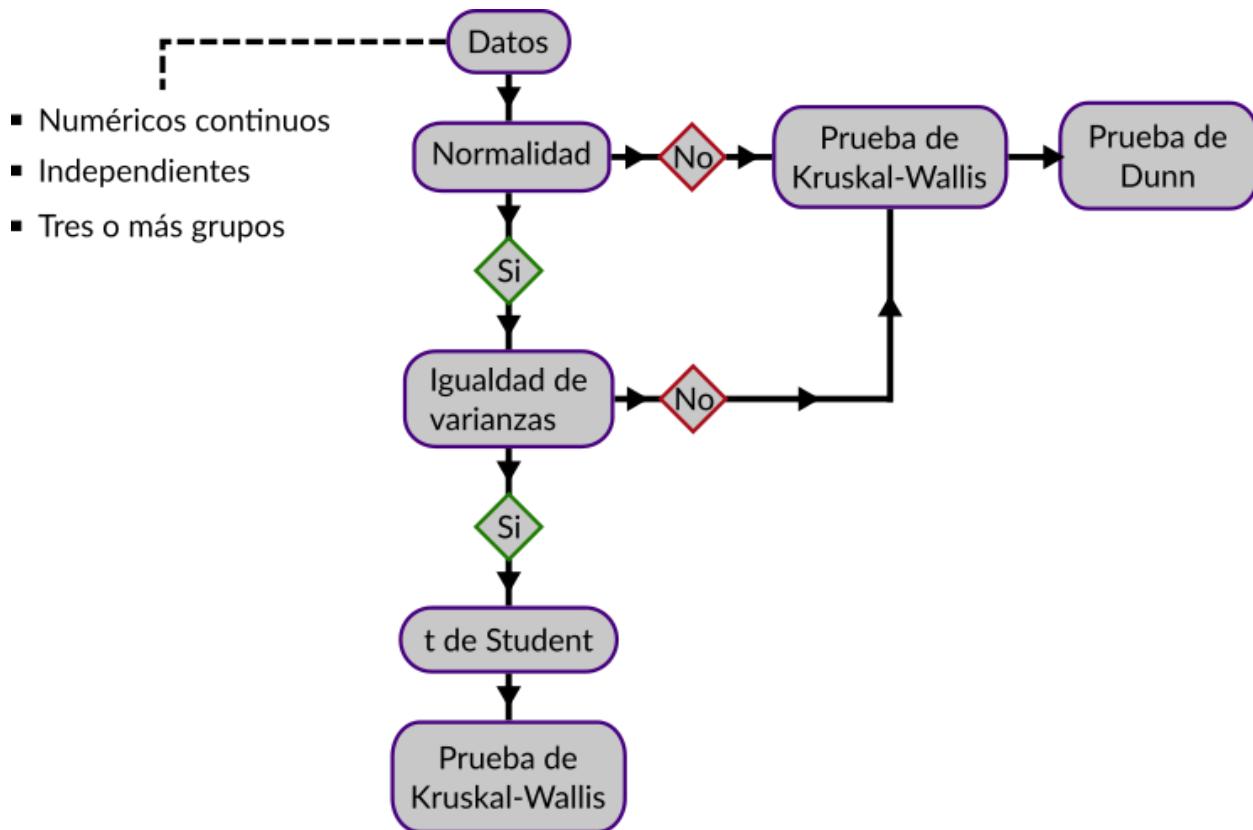
             size = 1)

```



7.4 Comparación entre tres o más grupos

Cuando se tienen tres o más grupos que se desean comparar es necesario realizar un ANOVA o una prueba de Kruskal-Wallis como la alternativa no paramétrica:



7.4.1 ANOVA

Es importante que se cumplan ambos supuestos, a diferencia de la prueba de t no es posible realizar el análisis si no hay homocedasticidad. Una consideración importante es que la normalidad se debe probar sobre los residuos del análisis, no sobre los datos originales.

Para el ANOVA se usará el conjunto de datos `chickwts`, el cual cuenta con pesos de pollos de acuerdo con seis tipos de alimentos:

```

str(chickwts)

## 'data.frame':    71 obs. of  2 variables:
## $ weight: num  179 160 136 227 217 168 108 124 143 140 ...
## $ feed   : Factor w/ 6 levels "casein","horsebean",...: 2 2 2 2 2 2 2 2 2 2 ...

```

Para evaluar la normalidad de los residuos primero es necesario realizar el análisis con la función `aov()` (analysis of variance), en donde los datos tienen que introducirse usando `formula` y `data`. En el primer argumento se indican los datos y las categorías usando el operador `~`:

```

aov(formula = weight ~ feed, data = chickwts)

## Call:
##   aov(formula = weight ~ feed, data = chickwts)
## 
## Terms:
##           feed Residuals
## Sum of Squares 231129.2 195556.0
## Deg. of Freedom      5       65
## 
## Residual standard error: 54.85029

```

```
## Estimated effects may be unbalanced
```

Los valores calculados por la función no se muestran en el resultado obtenido, para acceder a ellos es necesario primero guardar el análisis como un objeto:

```
modelo_pollos <- aov(formula = weight ~ feed, data = chickwts)
str(modelo_pollos)

## List of 13
## $ coefficients : Named num [1:6] 323.6 -163.4 -104.8 -46.7 -77.2 ...
##   ..- attr(*, "names")= chr [1:6] "(Intercept)" "feedhorsebean" "feedlinseed" "feedmeatmeal" ...
## $ residuals      : Named num [1:71] 18.8 -0.2 -24.2 66.8 56.8 ...
##   ..- attr(*, "names")= chr [1:71] "1" "2" "3" "4" ...
## $ effects       : Named num [1:71] -2201.8 345 228.6 -58.2 -237.4 ...
##   ..- attr(*, "names")= chr [1:71] "(Intercept)" "feedhorsebean" "feedlinseed" "feedmeatmeal" ...
## $ rank          : int 6
## $ fitted.values: Named num [1:71] 160 160 160 160 160 ...
##   ..- attr(*, "names")= chr [1:71] "1" "2" "3" "4" ...
## $ assign         : int [1:6] 0 1 1 1 1 1
## $ qr             :List of 5
##   ..$ qr    : num [1:71, 1:6] -8.426 0.119 0.119 0.119 0.119 ...
##   ... ..- attr(*, "dimnames")=List of 2
##   ... ... .$. : chr [1:71] "1" "2" "3" "4" ...
##   ... ... .$. : chr [1:6] "(Intercept)" "feedhorsebean" "feedlinseed" "feedmeatmeal" ...
##   ... ..- attr(*, "assign")= int [1:6] 0 1 1 1 1 1
##   ... ..- attr(*, "contrasts")=List of 1
##   ... ... .$. feed: chr "contr.treatment"
##   ..$ qraux: num [1:6] 1.12 1.26 1 1 1 ...
##   ..$ pivot: int [1:6] 1 2 3 4 5 6
##   ..$ tol  : num 1e-07
##   ..$ rank : int 6
##   ..- attr(*, "class")= chr "qr"
## $ df.residual   : int 65
## $ contrasts     :List of 1
##   ..$ feed: chr "contr.treatment"
## $ xlevels       :List of 1
##   ..$ feed: chr [1:6] "casein" "horsebean" "linseed" "meatmeal" ...
## $ call          : language aov(formula = weight ~ feed, data = chickwts)
## $ terms         :Classes 'terms', 'formula' language weight ~ feed
##   .. ..- attr(*, "variables")= language list(weight, feed)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. ..- attr(*, "dimnames")=List of 2
##   ... ... .$. : chr [1:2] "weight" "feed"
##   ... ... .$. : chr "feed"
##   .. ..- attr(*, "term.labels")= chr "feed"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(weight, feed)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "factor"
##   .. ..- attr(*, "names")= chr [1:2] "weight" "feed"
## $ model         :'data.frame': 71 obs. of 2 variables:
##   ..$ weight: num [1:71] 179 160 136 227 217 168 108 124 143 140 ...
##   ..$ feed   : Factor w/ 6 levels "casein","horsebean",...: 2 2 2 2 2 2 2 2 2 2 ...
```

```

##  ..- attr(*, "terms")=Classes 'terms', 'formula' language weight ~ feed
##  ... .- attr(*, "variables")= language list(weight, feed)
##  ... .- attr(*, "factors")= int [1:2, 1] 0 1
##  ... .- attr(*, "dimnames")=List of 2
##  ... .- .$. : chr [1:2] "weight" "feed"
##  ... .- .$. : chr "feed"
##  ... .- attr(*, "term.labels")= chr "feed"
##  ... .- attr(*, "order")= int 1
##  ... .- attr(*, "intercept")= int 1
##  ... .- attr(*, "response")= int 1
##  ... .- attr(*, ".Environment")=<environment: R_GlobalEnv>
##  ... .- attr(*, "predvars")= language list(weight, feed)
##  ... .- attr(*, "dataClasses")= Named chr [1:2] "numeric" "factor"
##  ... .- attr(*, "names")= chr [1:2] "weight" "feed"
## - attr(*, "class")= chr [1:2] "aov" "lm"

```

El objeto es una lista que tiene 13 elementos, el que es de interés es `residuals`, con el cual se realiza la prueba de Shapiro:

```

shapiro.test(x = modelo_pollos$residuals)

##
##  Shapiro-Wilk normality test
##
## data:  modelo_pollos$residuals
## W = 0.98616, p-value = 0.6272

```

Debido a que la distribución de los residuos es aproximadamente normal, evalúa la igualdad de varianzas usando la prueba de Bartlett con el data frame original:

```

bartlett.test(formula = weight ~ feed, data = chickwts)

##
##  Bartlett test of homogeneity of variances
##
## data:  weight by feed
## Bartlett's K-squared = 3.2597, df = 5, p-value = 0.66

```

Ambos supuestos se cumplen, por lo que los resultados del análisis son válidos. Se pueden volver a mostrar usando el nombre del objeto:

```

modelo_pollos

## Call:
##   aov(formula = weight ~ feed, data = chickwts)
##
## Terms:
##           feed Residuals
## Sum of Squares 231129.2 195556.0
## Deg. of Freedom      5       65
##
## Residual standard error: 54.85029
## Estimated effects may be unbalanced

```

Sin embargo, aquí no se muestra el valor de p. Para ello es necesario usar la función `summary()`:

```

summary(modelo_pollos)

##          Df Sum Sq Mean Sq F value    Pr(>F)
## feed      5 231129   46226   15.37 5.94e-10 ***

```

```

## Residuals   65 195556    3009
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

Aquí se muestran los grados de confianza, la suma de cuadrados, la media de cuadrados, el valor de F y el valor de p (Pr). En la parte inferior se muestra una leyenda que indica los valores de significancia, en este caso al valor de p es significativo y debido a que es muy cercano a 0 tiene tres asteriscos, lo cual indica que el valor es altamente significativo.

Las hipótesis nula del ANOVA establece que las medias de todos los grupos son iguales, mientras que la hipótesis alterna indica que al menos una de las medias es diferente al resto:

$$H_0 : \bar{t}_1 = \bar{t}_2 = \bar{t}_3 \dots = \bar{t}_4$$

$$H_A : \bar{t}_1 \neq \bar{t}_2 \neq \bar{t}_3 \dots \neq \bar{t}_4$$

Con el valor de p obtenido solo es posible afirmar que al menos una de las medias es diferente al resto, pero no se sabe cuál. En este caso, no es posible aún saber cuál es el alimento que ocasiona un peso significativamente diferente. Para ello se deben realizar comparaciones entre los grupos usando una prueba post hoc, como la prueba de Tukey con la función TukeyHSD, la cual solo requiere en x el modelo realizado:

```

TukeyHSD(x = modelo_pollos)

##      Tukey multiple comparisons of means
##      95% family-wise confidence level
##
## Fit: aov(formula = weight ~ feed, data = chickwts)
##
## $feed
##              diff          lwr          upr      p adj
## horsebean-casein -163.383333 -232.346876 -94.41979 0.0000000
## linseed-casein   -104.833333 -170.587491 -39.07918 0.0002100
## meatmeal-casein   -46.674242 -113.906207  20.55772 0.3324584
## soybean-casein    -77.154762 -140.517054 -13.79247 0.0083653
## sunflower-casein     5.333333  -60.420825  71.08749 0.9998902
## linseed-horsebean  58.550000 -10.413543 127.51354 0.1413329
## meatmeal-horsebean 116.709091  46.335105 187.08308 0.0001062
## soybean-horsebean  86.228571  19.541684 152.91546 0.0042167
## sunflower-horsebean 168.716667  99.753124 237.68021 0.0000000
## meatmeal-linseed    58.159091 -9.072873 125.39106 0.1276965
## soybean-linseed     27.678571 -35.683721  91.04086 0.7932853
## sunflower-linseed   110.166667  44.412509 175.92082 0.0000884
## soybean-meatmeal   -30.480519 -95.375109  34.41407 0.7391356
## sunflower-meatmeal  52.007576 -15.224388 119.23954 0.2206962
## sunflower-soybean   82.488095  19.125803 145.85039 0.0038845

```

El resultado muestra las comparaciones entre pares de todas las categorías, aquellas que son significativamente diferentes tiene un valor de p (padj) menor a 0.05. Si se quisieran separar solo aquellas comparaciones que son significativas primero es necesario convertir la matriz resultante a un tibble. Para acceder a la matriz se usa el operador \$ y el nombre vector que contiene las categorías:

```

prueba_tukey <- TukeyHSD(x = modelo_pollos)
prueba_tukey$feed %>%
  class()
  
## [1] "matrix" "array"

```

```

prueba_tukey$feed %>%
  as_tibble()

## # A tibble: 15 x 4
##      diff     lwr     upr   `p adj`
##      <dbl>   <dbl>   <dbl>   <dbl>
## 1 -163.    -232.   -94.4 0.0000000307
## 2 -105.    -171.   -39.1 0.000210
## 3 -46.7    -114.   20.6  0.332
## 4 -77.2    -141.   -13.8 0.00837
## 5  5.33    -60.4   71.1  1.00
## 6  58.6    -10.4   128.  0.141
## 7  117.     46.3   187.  0.000106
## 8  86.2    19.5   153.  0.00422
## 9  169.     99.8   238.  0.0000000122
## 10 58.2    -9.07   125.  0.128
## 11 27.7    -35.7   91.0  0.793
## 12 110.     44.4   176.  0.0000884
## 13 -30.5   -95.4   34.4  0.739
## 14 52.0    -15.2   119.  0.221
## 15 82.5    19.1   146.  0.00388

```

Usando el tibble es posible usar `filter()` en la cuarta columna para seleccionar solo los datos menores a 0.05. Para mantener los nombres de las filas también es necesario crear una nueva variable con `mutate`:

```

prueba_tukey$feed %>%
  as_tibble() %>%
  mutate(comparacion = row.names(TukeyHSD(x = modelo_pollos)$feed)) %>%
  filter(`p adj` < .05) %>%
  select(comparacion, `p adj`)

## # A tibble: 8 x 2
##   comparacion   `p adj`
##   <chr>          <dbl>
## 1 horsebean-casein 0.0000000307
## 2 linseed-casein  0.000210
## 3 soybean-casein  0.00837
## 4 meatmeal-horsebean 0.000106
## 5 soybean-horsebean 0.00422
## 6 sunflower-horsebean 0.0000000122
## 7 sunflower-linseed  0.0000884
## 8 sunflower-soybean  0.00388

```

Es importante usar “ para indicar el nombre de la columna `p adj` debido a que el espacio entre ambos términos lo convierte en un nombre no sintáctico. Con el último data frame obtenido es posible tener una idea más clara sobre los grupos que difieren significativamente unos de otros.

Para graficar los resultados del ANOVA se usa la media y la desviación estándar, por ejemplo con una gráfica de barras:

```

medias_pollos <- chickwts %>%
  group_by(feed) %>%
  summarize(media_peso = mean(weight),
            desv_peso = sd(weight))
medias_pollos

## # A tibble: 6 x 3
##   feed      media_peso  desv_peso
##   <fct>     <dbl>       <dbl>
## 1 chicken   377.        19.5
## 2 corn      373.        22.5
## 3 linseed   364.        18.0
## 4 milo      364.        17.5
## 5 soybean   364.        17.5
## 6 sunflower 364.        17.5

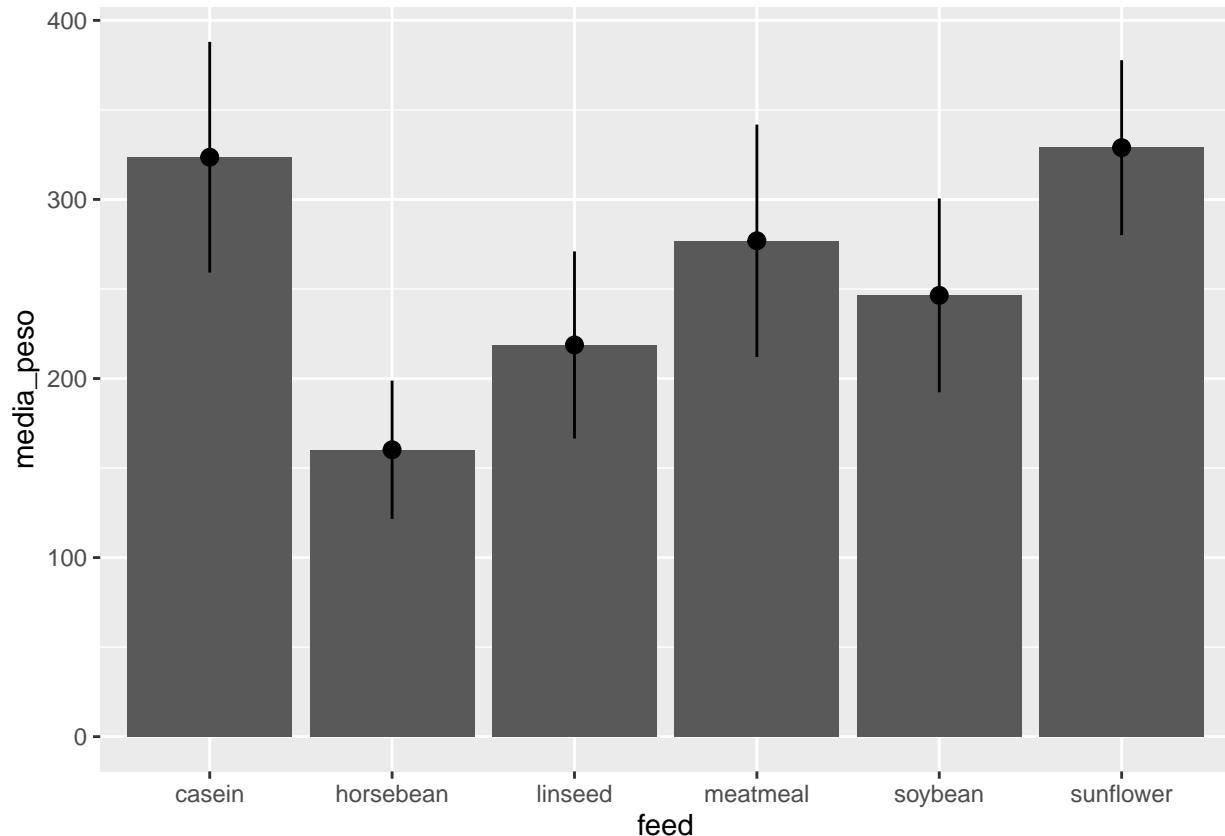
```

```

##   <fct>      <dbl>      <dbl>
## 1 casein     324.       64.4
## 2 horsebean  160.       38.6
## 3 linseed    219.       52.2
## 4 meatmeal   277.       64.9
## 5 soybean    246.       54.1
## 6 sunflower  329.       48.8

ggplot(data = medias_pollos, aes(x = feed, y = media_peso)) +
  geom_col() +
  geom_pointrange(aes(ymin = media_peso - desv_peso,
                        ymax = media_peso + desv_peso))

```



Las barras de error sirven como guías visuales para separar grupos que son diferentes uno de otros, pero no es una manera siempre clara de ver esas diferencias. Para identificar grupos con letras se requiere un proceso algo engorroso, pero se describe aquí en términos generales.

La función `multcompLetters4` del paquete `multcompView` permite crear los grupos con sus letras, solo se requieren poner los objetos del ANOVA y de la prueba de Tukey en los argumentos `object` y `comp`, respectivamente:

```

grupos <- multcompLetters4(object = modelo_pollos,
                            comp = prueba_tukey)
grupos
## $feed
## sunflower  casein  meatmeal  soybean  linseed horsebean
##      "a"      "a"      "ab"     "b"      "bc"     "c"

```

Al imprimir este nuevo objeto se muestra que tiene solo un elemento que es `feed`. Sin embargo, se trata de

un objeto más complejo de lo que inicialmente parece:

```
str(grupos)

## List of 1
## $ feed:List of 3
##   ..$ Letters      : Named chr [1:6] "a" "a" "ab" "b" ...
##   ...- attr(*, "names")= chr [1:6] "sunflower" "casein" "meatmeal" "soybean" ...
##   ..$ monospacedLetters: Named chr [1:6] "a" "a" "ab" "b" ...
##   ...- attr(*, "names")= chr [1:6] "sunflower" "casein" "meatmeal" "soybean" ...
##   ..$ LetterMatrix    : logi [1:6, 1:3] TRUE TRUE TRUE FALSE FALSE FALSE ...
##   ...- attr(*, "dimnames")=List of 2
##     ..$ : chr [1:6] "sunflower" "casein" "meatmeal" "soybean" ...
##     ..$ : chr [1:3] "a" "b" "c"
##   ..- attr(*, "class")= chr "multcompLetters"
```

El resultado de `str()` muestra que `grupos` es una lista cuyo único elemento es a su vez una lista que tiene tres elementos. Para hacer el cuento corto, es necesario convertir este objeto a un data frame para después juntarlo con `medias_pollos` de modo que las letras que representan los grupos puedan ser colocadas por encima de las barras de error siguiendo el tipo de alimento al que corresponden. La función `as.data.frame.list` permite hacer la conversión de la lista a un data frame, debido a que la lista que buscamos se encuentra dentro de `grupos` es necesario usar el operador `$`:

```
letras <- as.data.frame.list(grupos$feed)
letras

##           Letters monospacedLetters LetterMatrix.a LetterMatrix.b
## sunflower      a            a        TRUE      FALSE
## casein        a            a        TRUE      FALSE
## meatmeal      ab           ab        TRUE       TRUE
## soybean        b            b        FALSE      TRUE
## linseed        bc           bc        FALSE      TRUE
## horsebean      c            c        FALSE      FALSE
##                   LetterMatrix.c
## sunflower      FALSE
## casein        FALSE
## meatmeal      FALSE
## soybean        FALSE
## linseed        TRUE
## horsebean      TRUE
```

Este data frame aún cuenta con una desventaja, los nombres de los alimentos se encuentran como nombres de filas, pero no como una variable dentro de un data frame, para incluirlos como tal se requiere el siguiente código:

```
letras$feed <- row.names(letras)
```

Ahora, es posible ver en el mismo data frame como se corresponden los grupos de acuerdo con las letras que representan los grupos:

```
letras %>%
  select(Letters, feed)

##           Letters      feed
## sunflower      a  sunflower
## casein        a   casein
## meatmeal      ab  meatmeal
## soybean        b  soybean
## linseed        bc linseed
```

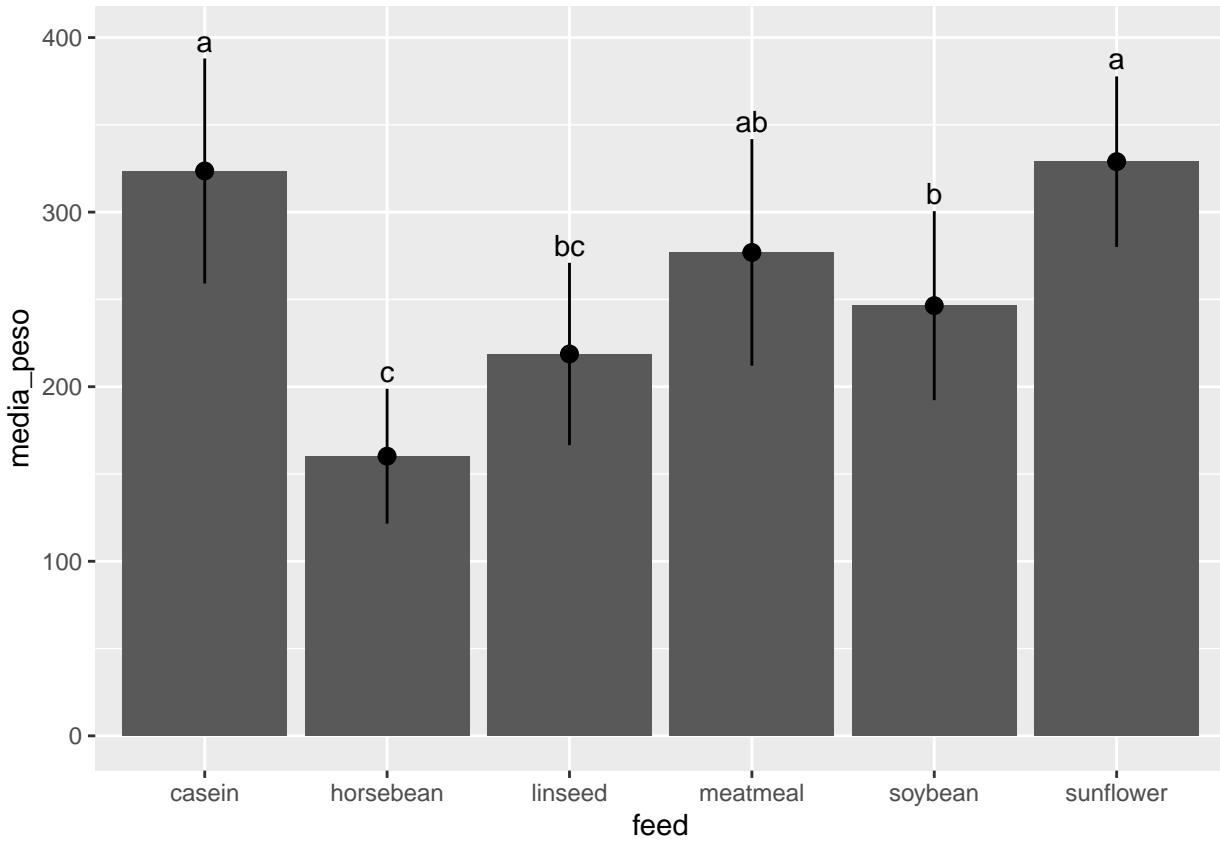
```
## horsebean      c horsebean
El último paso consiste en unir este data frame con medias_pollos usando la columna feed de ambos data frames como guía para que las filas de ambos objetos sean emparejadas adecuadamente:
```

```
medias_pollos <- medias_pollos %>%
  left_join(y = letras, by = "feed")
medias_pollos

## # A tibble: 6 x 8
##   feed      media_peso desv_peso Letters monospacedLet~1 Lette~2 Lette~3 Lette~4
##   <chr>     <dbl>     <dbl> <chr>    <chr>      <lgl>    <lgl>    <lgl>
## 1 casein     324.      64.4 a     "a "      TRUE     FALSE    FALSE
## 2 horsebean   160.      38.6 c     " c"      FALSE    FALSE    TRUE
## 3 linseed     219.      52.2 bc    " bc"     FALSE    TRUE    TRUE
## 4 meatmeal    277.      64.9 ab    "ab "     TRUE     TRUE    FALSE
## 5 soybean     246.      54.1 b     " b"      FALSE    TRUE    FALSE
## 6 sunflower   329.      48.8 a     "a "      TRUE     FALSE    FALSE
## # ... with abbreviated variable names 1: monospacedLetters, 2: LetterMatrix.a,
## #   3: LetterMatrix.b, 4: LetterMatrix.c
```

Ahora, `medias_pollos` tiene la organización adecuada para realizar la gráfica con letras que indiquen los grupos. Usando `geom_text()` se coloca la columna `Letters` como estética solamente para esa función y `nudge_y` indica el desplazamiento vertical a partir de la media:

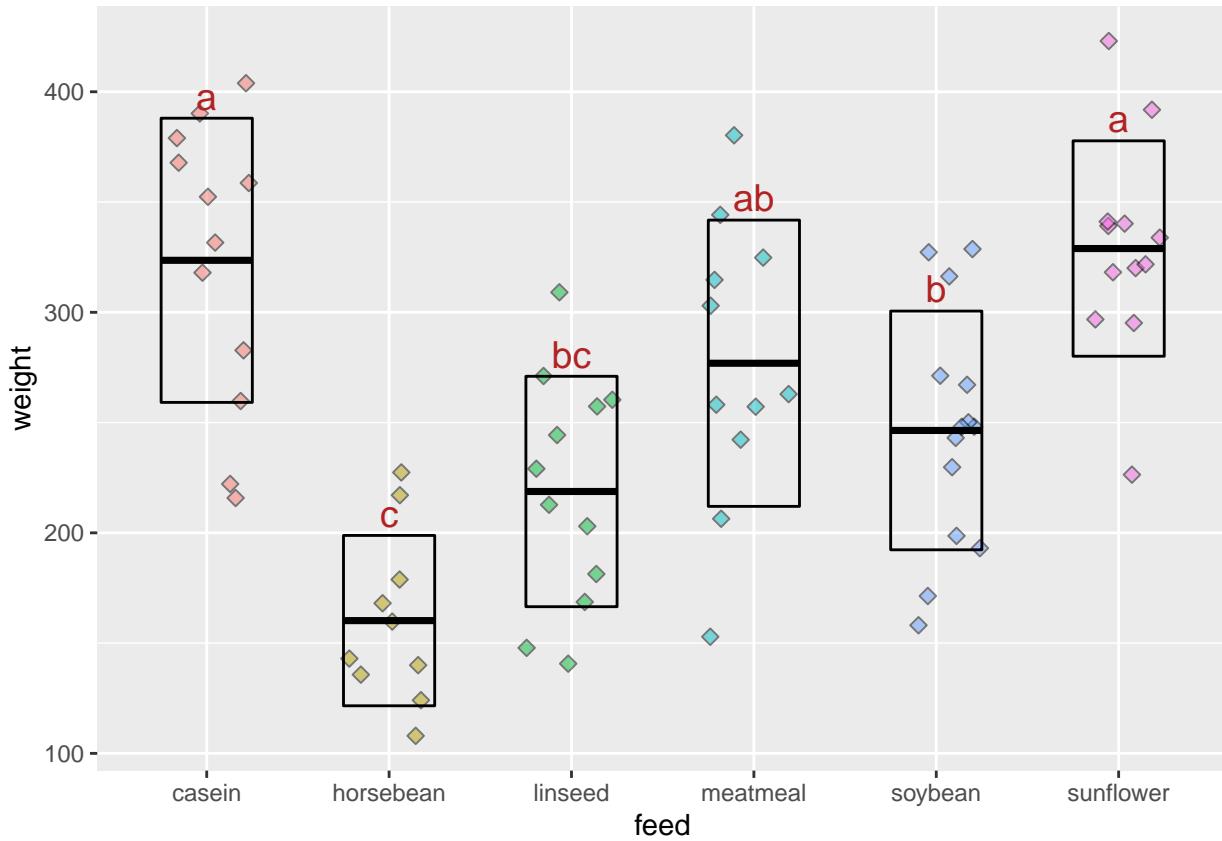
```
ggplot(data = medias_pollos, aes(x = feed, y = media_peso)) +
  geom_col() +
  geom_pointrange(aes(ymin = media_peso - desv_peso,
                       ymax = media_peso + desv_peso)) +
  geom_text(aes(label = Letters),
            nudge_y = medias_pollos$desv_peso + 10)
```



Ahora es más sencillo determinar los tratamientos que son diferentes entre si. Aquellos que tienen la misma letra pertenecen al mismo grupo y por lo tanto no hay diferencia significativa entre ellos. Por ejemplo, caseína y girasol. Pero entre caseína y haba (horsebean) si hay diferencia significativa. Entre caseína y linaza (linseed) también hay diferencia significativa, pero al mismo tiempo linaza y haba pertenecen al mismo grupo. De acuerdo con estas comparaciones, la gráfica muestra que la caseína y el son los alimentos que resultan en mayor peso, mientras que los menores pesos se deben a la alimentación con linaza y habas.

Una representación más informativa puede ser mostrando las observaciones individuales de cada uno de los grupos usando `geom_jitter()`:

```
ggplot() +
  geom_jitter(data = chickwts, aes(x = feed, y = weight,
                                    fill = feed),
              width = 1/4, pch = 23, size = 2, alpha = 1/2,
              show.legend = FALSE) +
  geom_crossbar(data = medias_pollos, aes(x = feed, y = media_peso,
                                            ymin = media_peso - desv_peso,
                                            ymax = media_peso + desv_peso),
                width = 1/2) +
  geom_text(data = medias_pollos, aes(x = feed, y = media_peso,
                                       label = Letters),
            nudge_y = medias_pollos$desv_peso + 10, color = "firebrick",
            size = 5)
```



7.4.2 Prueba de Kruskal Wallis

La prueba no paramétrica debe usarse cuando al menos uno de los dos supuestos requeridos por el ANOVA no se cumple. `OrchardSprays` contiene datos de un experimento que mide la reducción de abejas en celdas de cera con 8 tratamientos consistentes de 7 concentraciones de polisulfuro de calcio y un control. Los tratamientos están identificados con letras:

```
str(OrchardSprays)
```

```
## 'data.frame': 64 obs. of 4 variables:
## $ decrease : num 57 95 8 69 92 90 15 2 84 6 ...
## $ rowpos   : num 1 2 3 4 5 6 7 8 1 2 ...
## $ colpos   : num 1 1 1 1 1 1 1 2 2 ...
## $ treatment: Factor w/ 8 levels "A","B","C","D",...: 4 5 2 8 7 6 3 1 3 2 ...
```

Las columnas `rowpos` y `colpos` indican la posición de los tratamientos de acuerdo con el diseño experimental de cuadro latino, por lo que las únicas columnas que se usarán para el análisis son `decrease` y `treatment`:

```
bartlett.test(formula = decrease ~ treatment, data = OrchardSprays)
```

```
##
##  Bartlett test of homogeneity of variances
##
## data: decrease by treatment
## Bartlett's K-squared = 42.031, df = 7, p-value = 5.128e-07
```

Debido a que no hay homogeneidad de varianzas, no es posible realizar un ANOVA. La función para realizar el análisis de Kruskal-Wallis es `kruskal.test`, que usa los argumentos `formula` y `data` para introducir los datos.

```

kruskal.test(formula = decrease ~ treatment,
             data = OrchardSprays)

##
##  Kruskal-Wallis rank sum test
##
## data: decrease by treatment
## Kruskal-Wallis chi-squared = 48.874, df = 7, p-value = 2.402e-08

```

El estadístico del análisis es la χ^2 , después se muestran los grados de confianza y el valor de p, el cual es significativo en este caso. La hipótesis nula de este análisis establece que la mediana de los grupos es la misma, mientras que la alternativa indica que al menos uno de los grupos tiene una mediana diferente al resto:

$$H_0 : Md_1 = Md_2 = Md_3 \dots = Md_n$$

$$H_A : Md_1 \neq Md_2 \neq Md_3 \dots \neq Md_n$$

De modo que a partir del resultado obtenido solo podemos determinar que alguno de los grupos difiere respecto a al menos uno de los otros en su mediana. Para determinar cómo son estas diferencias es necesario hacer una prueba post hoc no paramétrica. En este caso, una opción disponible está en el paquete `dunn.test`, cuya única función lleva el mismo nombre: `dunn.test()`. Usando los argumentos `x` y `g` se introducen vectores con los datos y las categorías, respectivamente:

```

dunn.test(x = OrchardSprays$decrease,
          g = OrchardSprays$treatment)

##
##  Kruskal-Wallis rank sum test
##
## data: x and group
## Kruskal-Wallis chi-squared = 48.8735, df = 7, p-value = 0
##
##
##                                     Comparison of x by group
##                                     (No adjustment)
## Col Mean-
## Row Mean |      A      B      C      D      E      F
## -----+-----
##       B | -0.537239
##           |  0.2956
##       C | -1.974356 -1.437116
##           |  0.0242*   0.0753
##       D | -2.565320 -2.028080 -0.590963
##           |  0.0052*   0.0213*   0.2773
##       E | -3.861411 -3.324172 -1.887055 -1.296091
##           |  0.0001*   0.0004*   0.0296   0.0975
##       F | -4.197186 -3.659946 -2.222830 -1.631866 -0.335774
##           |  0.0000*   0.0001*   0.0131*   0.0514   0.3685
##       G | -4.304634 -3.767394 -2.330278 -1.739314 -0.443222 -0.107447
##           |  0.0000*   0.0001*   0.0099*   0.0410   0.3288   0.4572

```

```

##          |
##      H | -5.123925 -4.586685 -3.149569 -2.558605 -1.262513 -0.926738
##          | 0.0000*   0.0000*  0.0008*  0.0053*   0.1034   0.1770
## Col Mean-|
## Row Mean |      G
## -----
##      H | -0.819290
##          | 0.2063
##
## alpha = 0.05
## Reject Ho if p <= alpha/2

```

El resultado del análisis muestra también los resultados del análisis de Kruskal Wallis, así como una matriz con las comparaciones. Aquellas que son significativamente diferentes están marcadas con un asterisco. Si se desean ver estas diferencias usando un data frame es necesario primero guardar la prueba como un objeto:

```

prueba_dunn <- dunn.test(x = OrchardSprays$decrease,
                           g = OrchardSprays$treatment)

##  Kruskal-Wallis rank sum test
##
## data: x and group
## Kruskal-Wallis chi-squared = 48.8735, df = 7, p-value = 0
##
##
##                                     Comparison of x by group
##                                     (No adjustment)
## Col Mean-|
## Row Mean |      A      B      C      D      E      F
## -----
##      B | -0.537239
##          | 0.2956
##      |
##      C | -1.974356 -1.437116
##          | 0.0242*   0.0753
##      |
##      D | -2.565320 -2.028080 -0.590963
##          | 0.0052*   0.0213*   0.2773
##      |
##      E | -3.861411 -3.324172 -1.887055 -1.296091
##          | 0.0001*   0.0004*   0.0296   0.0975
##      |
##      F | -4.197186 -3.659946 -2.222830 -1.631866 -0.335774
##          | 0.0000*   0.0001*   0.0131*   0.0514   0.3685
##      |
##      G | -4.304634 -3.767394 -2.330278 -1.739314 -0.443222 -0.107447
##          | 0.0000*   0.0001*   0.0099*   0.0410   0.3288   0.4572
##      |
##      H | -5.123925 -4.586685 -3.149569 -2.558605 -1.262513 -0.926738
##          | 0.0000*   0.0000*  0.0008*  0.0053*   0.1034   0.1770
## Col Mean-|
## Row Mean |      G
## -----
##      H | -0.819290
##          | 0.2063

```

```

## 
## alpha = 0.05
## Reject Ho if p <= alpha/2

str(prueba_dunn)

## List of 5
## $ chi2      : num 48.9
## $ Z          : num [1:28] -0.537 -1.974 -1.437 -2.565 -2.028 ...
## $ P          : num [1:28] 0.29555 0.02417 0.07534 0.00515 0.02128 ...
## $ P.adjusted : num [1:28] 0.29555 0.02417 0.07534 0.00515 0.02128 ...
## $ comparisons: chr [1:28] "A - B" "A - C" "B - C" "A - D" ...

```

La estructura del objeto es una lista con cinco vectores, de modo que la función `as.data.frame.list()` puede ser de ayuda para convertirla a un data frame:

```

as.data.frame.list(prueba_dunn)

##      chi2      Z      P   P.adjusted comparisons
## 1 48.87353 -0.5372399 2.955509e-01 2.955509e-01     A - B
## 2 48.87353 -1.9743567 2.417060e-02 2.417060e-02     A - C
## 3 48.87353 -1.4371168 7.534241e-02 7.534241e-02     B - C
## 4 48.87353 -2.5653206 5.154025e-03 5.154025e-03     A - D
## 5 48.87353 -2.0280807 2.127601e-02 2.127601e-02     B - D
## 6 48.87353 -0.5909639 2.772723e-01 2.772723e-01     C - D
## 7 48.87353 -3.8614119 5.636682e-05 5.636682e-05     A - E
## 8 48.87353 -3.3241720 4.434074e-04 4.434074e-04     B - E
## 9 48.87353 -1.8870552 2.957645e-02 2.957645e-02     C - E
## 10 48.87353 -1.2960913 9.747201e-02 9.747201e-02     D - E
## 11 48.87353 -4.1971869 1.351255e-05 1.351255e-05     A - F
## 12 48.87353 -3.6599470 1.261337e-04 1.261337e-04     B - F
## 13 48.87353 -2.2228302 1.311363e-02 1.311363e-02     C - F
## 14 48.87353 -1.6318663 5.135383e-02 5.135383e-02     D - F
## 15 48.87353 -0.3357750 3.685203e-01 3.685203e-01     E - F
## 16 48.87353 -4.3046349 8.363069e-06 8.363069e-06     A - G
## 17 48.87353 -3.7673949 8.247996e-05 8.247996e-05     B - G
## 18 48.87353 -2.3302782 9.895727e-03 9.895727e-03     C - G
## 19 48.87353 -1.7393142 4.098975e-02 4.098975e-02     D - G
## 20 48.87353 -0.4432229 3.288022e-01 3.288022e-01     E - G
## 21 48.87353 -0.1074480 4.572168e-01 4.572168e-01     F - G
## 22 48.87353 -5.1239257 1.496194e-07 1.496194e-07     A - H
## 23 48.87353 -4.5866858 2.251688e-06 2.251688e-06     B - H
## 24 48.87353 -3.1495690 8.175574e-04 8.175574e-04     C - H
## 25 48.87353 -2.5586051 5.254652e-03 5.254652e-03     D - H
## 26 48.87353 -1.2625138 1.033820e-01 1.033820e-01     E - H
## 27 48.87353 -0.9267389 1.770311e-01 1.770311e-01     F - H
## 28 48.87353 -0.8192909 2.063102e-01 2.063102e-01     G - H

```

Ahora se pueden filtrar los valores menores a 0.05 para claramente visualizar las comparaciones que son significativamente diferentes:

```

as.data.frame.list(prueba_dunn) %>%
  filter(P.adjusted < 0.05)

##      chi2      Z      P   P.adjusted comparisons
## 1 48.87353 -1.974357 2.417060e-02 2.417060e-02     A - C
## 2 48.87353 -2.565321 5.154025e-03 5.154025e-03     A - D
## 3 48.87353 -2.028081 2.127601e-02 2.127601e-02     B - D

```

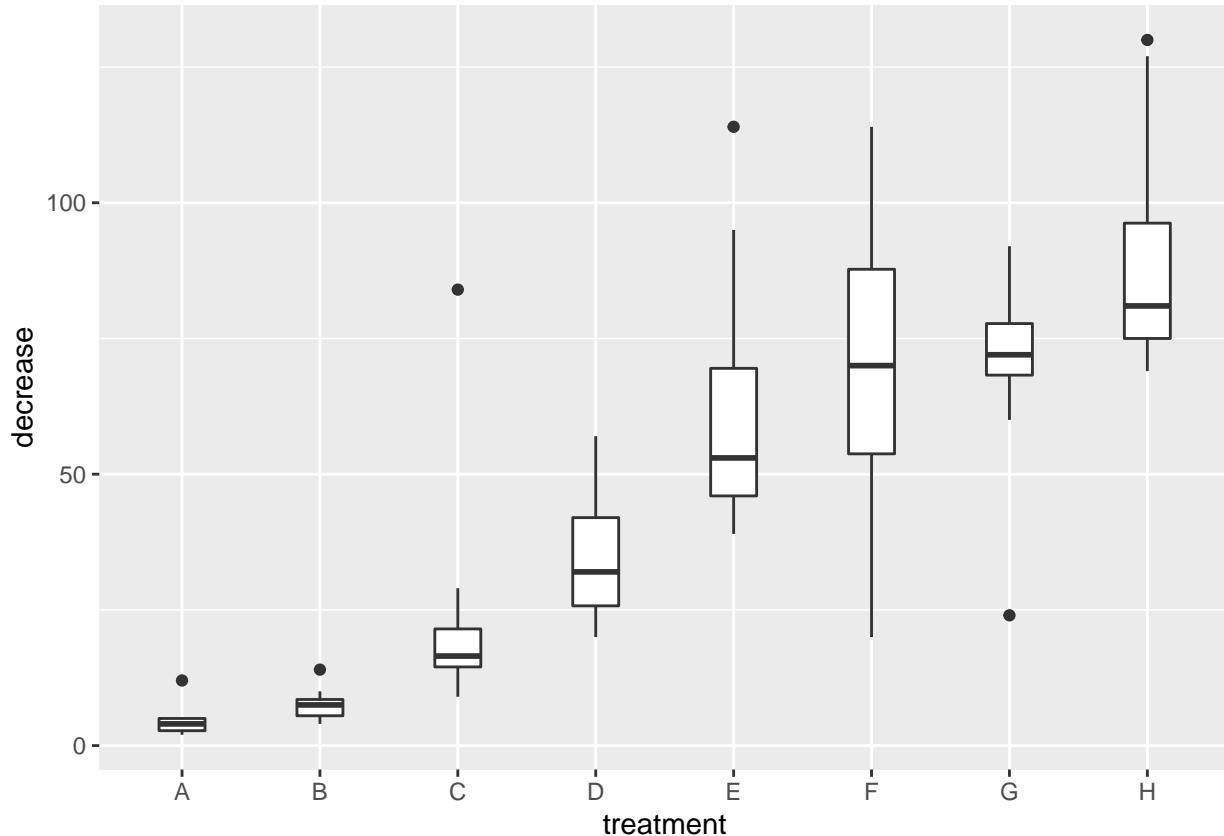
```

## 4 48.87353 -3.861412 5.636682e-05 5.636682e-05      A - E
## 5 48.87353 -3.324172 4.434074e-04 4.434074e-04      B - E
## 6 48.87353 -1.887055 2.957645e-02 2.957645e-02      C - E
## 7 48.87353 -4.197187 1.351255e-05 1.351255e-05      A - F
## 8 48.87353 -3.659947 1.261337e-04 1.261337e-04      B - F
## 9 48.87353 -2.222830 1.311363e-02 1.311363e-02      C - F
## 10 48.87353 -4.304635 8.363069e-06 8.363069e-06     A - G
## 11 48.87353 -3.767395 8.247996e-05 8.247996e-05     B - G
## 12 48.87353 -2.330278 9.895727e-03 9.895727e-03     C - G
## 13 48.87353 -1.739314 4.098975e-02 4.098975e-02     D - G
## 14 48.87353 -5.123926 1.496194e-07 1.496194e-07     A - H
## 15 48.87353 -4.586686 2.251688e-06 2.251688e-06     B - H
## 16 48.87353 -3.149569 8.175574e-04 8.175574e-04     C - H
## 17 48.87353 -2.558605 5.254652e-03 5.254652e-03     D - H

```

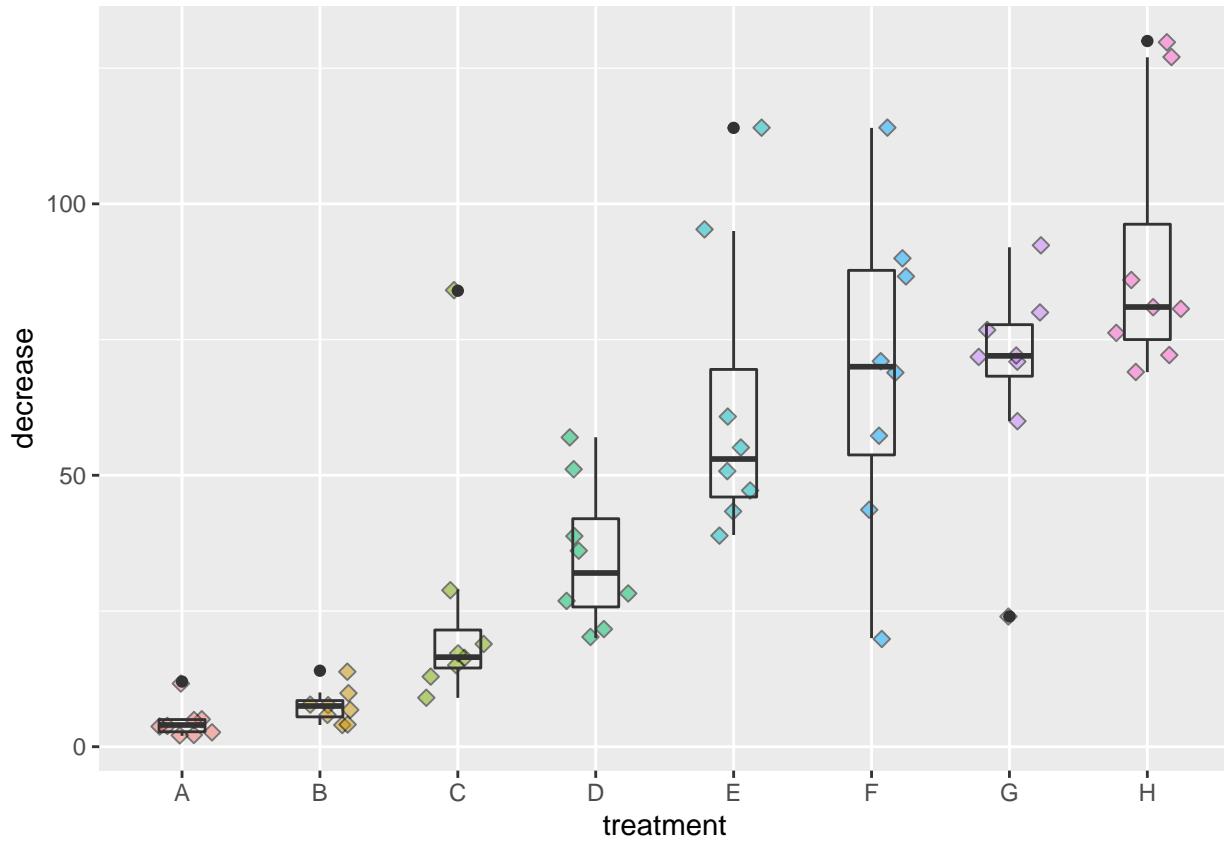
Debido a que la prueba de Kruskal-Wallis compara medias de los grupos, una representación gráfica adecuada es una de cajas y bigotes:

```
ggplot(data = OrchardSprays, aes(x = treatment, y = decrease)) +
  geom_boxplot(width = 1/3)
```



Para una gráfica más informativa se pueden mostrar las observaciones individuales de cada grupo:

```
ggplot(data = OrchardSprays, aes(x = treatment, y = decrease)) +
  geom_jitter(width = 1/4, aes(fill = treatment), pch = 23,
              size = 2, alpha = 1/2, show.legend = FALSE) +
  geom_boxplot(width = 1/3, fill = "transparent")
```



Actualmente no conozco ninguna manera de mostrar grupos como es el caso del ANOVA, ya que las funciones utilizadas para esa sección no funcionan con los análisis no paramétricos. Mi recomendación sería acompañar la gráfica con la tabla con los valores de p para cada comparación.

7.5 Prueba de xi cuadrada

La función `chisqu.test()` puede usarse de dos maneras: con tablas de contingencia y con datos que se ajustan a una proporción. En ambos casos, los datos provienen de conteos. Para una tabla de contingencia esos conteos están asociados a dos variables y la organización de la tabla permite mostrar los datos desglosados de acuerdo con los niveles de cada variable. Lo que busca probar esta prueba es independencia entre ambas variables, la hipótesis nula establece que las dos variables son independientes y la hipótesis alternativa que no lo son:

H_0 : Las variables son independientes

H_1 : Las variables no son independientes

Por ejemplo, en un muestreo de insectos que se realizó en tres sitios y cada individuo fue separado de acuerdo con su sexo, se busca determinar si el sexo y el sitio de muestreo son o no independientes:

```
insectos_sitio <- read_csv(file = "datos_manual/insectos_machos_hembras_sitio.csv",
                           col_type = "fiii")
insectos_sitio
## # A tibble: 2 x 4
##   sexo      A     B     C
##   <fct>  <int> <int> <int>
```

```
## 1 macho      46    92   128
## 2 hembra     21    56    51
```

La tabla de contingencia es solo una manera de organizar dos variables en formato ancho, en formato largo se vería así:

```
insectos_sitio %>%
  pivot_longer(cols = c(A, B, C),
               values_to = "conteo",
               names_to = "sitio")

## # A tibble: 6 x 3
##   sexo   sitio conteo
##   <fct> <chr>  <int>
## 1 macho  A        46
## 2 macho  B        92
## 3 macho  C       128
## 4 hembra A        21
## 5 hembra B        56
## 6 hembra C        51
```

La función `chisq.test()` solo funcionará con el data frame si las columnas son numéricas, con la presencia de `sexo` no funciona:

```
chisq.test(insectos_sitio)

## Error in chisq.test(insectos_sitio): all entries of 'x' must be nonnegative and finite
```

Usando `select()` es posible excluir a la columna `sexo` y se usa `%>%` para mandar el nuevo data frame a la función `chisq.test`:

```
insectos_sitio %>%
  select(-sexo) %>%
  chisq.test()

##
##  Pearson's Chi-squared test
##
## data: .
## X-squared = 3.2747, df = 2, p-value = 0.1945
```

De acuerdo con el valor de p , se cumple la hipótesis nula y los datos son independientes.

La otra manera de usar `chisq.test()` es usando un vector con proporciones observadas y otro con esperadas. En este caso las hipótesis nula establece que las proporciones observadas y esperadas son iguales, mientras que la alternativa indica que no lo son:

$$H_0 : \text{Prop}_{\text{observadas}} = \text{Prop}_{\text{esperadas}}$$

$$H_0 : \text{Prop}_{\text{observadas}} \neq \text{Prop}_{\text{esperadas}}$$

Los siguientes datos muestran las proporciones observadas y esperadas de 1000 chícharos resultantes de la segunda generación de cruzas dihíbridas entre chícharos amarillos lisos con verdes rugosos:

```
chicharos <- read_csv(file = "datos_manual/chicharos.csv",
                      col_types = "fnn")
chicharos
```

```

## # A tibble: 4 x 3
##   fenotipo      observados   esperados
##   <fct>          <dbl>        <dbl>
## 1 amarillo_liso     540        562.
## 2 verde_liso       193        188.
## 3 amarillo_rugoso   190        188.
## 4 verde_rugoso      65        62.5

```

Los datos en `esperados` corresponden a la proporción 9:3:3:1 de los fenotipos indicados. Para realizar la prueba, solo es necesario introducir en `x` los datos observados y en `p` los esperados. Sin embargo, los datos en `p` deben ser probabilidades, es decir, tienen que ser frecuencias relativas que se expresan como números decimales que en conjunto suman 1. Usando el argumento `rescale.p = TRUE` se puede realizar la conversión necesaria partiendo de las frecuencias absolutas:

```
chisq.test(x = observados, p = esperados, rescale.p = TRUE)
```

```
## Error in is.data.frame(x): object 'observados' not found
```

Otra manera de usar la función sería realizar la conversión manualmente o introducir los datos en `p` como números decimales:

```
chisq.test(x = observados, p = esperados /1000)
```

```
## Error in is.data.frame(x): object 'observados' not found
```

De cualquier modo, resultado es el mismo. Debido a que el valor de `p` es mayor a 0.05, se cumple la hipótesis nula y los datos observados son aproximadamente iguales a los datos esperados.

8 Prácticas biología

En esta sección se describe cómo manejar los datos que se presentan en algún momento relacionados con diferentes materias de la carrera. Digo, eventualmente porque ahorita solo hay una.

8.1 Cuantificación de BSA mediante curva patrón

En este ejercicio se usarán datos de absorbancia de acuerdo con la concentración de albúmina de suero bovina (BSA). Algunas de las lecturas provienen de soluciones cuyas concentraciones son conocidas, mientras que el resto son lecturas de dos soluciones con concentraciones desconocidas.

```

read_csv(file = "practicas_manual/curva_patron/curva_patron_bsa.csv",
         col_types = "fnnnfn")

## # A tibble: 11 x 6
##   tubo    volumen_bsa    volumen_agua    volumen_tot tipo      absorbancia
##   <fct>      <dbl>        <dbl>        <dbl> <fct>        <dbl>
## 1 1           0            100          100 conocida      0
## 2 2          12.5          87.5          100 conocida    0.106
## 3 3           25            75          100 conocida    0.159
## 4 4          37.5          62.5          100 conocida    0.265
## 5 5           50            50          100 conocida    0.35
## 6 6          62.5          37.5          100 conocida    0.452
## 7 7           75            25          100 conocida    0.509
## 8 8           NA            NA           50 problema_1  0.335
## 9 9           NA            NA           25 problema_1  0.304
## 10 10         NA            NA           80 problema_2  0.215
## 11 11         NA            NA           60 problema_2  0.208

```

El data frame muestra el número de tubo, el volumen de BSA (0.2 µg/µL) en µL usado para cada solución, el volumen de agua en µL, volumen total de la solución en µL, tipo de solución y la absorbancia a 595 nm

registrada para cada tubo. Las le BSA, se puede crear un modelo lineal para calcular las concentraciones desconocidas de ambas soluciones. Las dos observaciones de la solución `problema_1` tienen la misma concentración, al igual que aquellas de la solución `problema_2`. Una vez obtenidas las concentraciones de cada observación, se promediarán las de cada solución para estimar la concentración de cada una.

Primero se introduce la información a R. Debemos indicar que todas las variables son numéricas, a excepción de `tubo` y `tipo`, siguiendo el orden de las columnas y usando `col_types`. Debido a que R muestra el número de filas, no es necesario incluir la columna `tubo`, la cual puede removese con `col_select`:

```
bsa <- read_csv(file = "practicas_manual/curva_patron/curva_patron_bsa.csv",
                col_types = "fnnnfn",
                col_select = - tubo)

bsa

## # A tibble: 11 x 5
##   volumen_bsa    volumen_agua  volumen_tot tipo      absorbancia
##       <dbl>        <dbl>        <dbl> <fct>      <dbl>
## 1         0          100        100 conocida     0
## 2        12.5        87.5        100 conocida  0.106
## 3         25          75        100 conocida  0.159
## 4        37.5        62.5        100 conocida  0.265
## 5         50          50        100 conocida  0.35
## 6        62.5        37.5        100 conocida  0.452
## 7         75          25        100 conocida  0.509
## 8         NA          NA        50 problema_1 0.335
## 9         NA          NA        25 problema_1 0.304
## 10        NA          NA        80 problema_2 0.215
## 11        NA          NA        60 problema_2 0.208
```

Ahora, se debe calcular la concentración de las soluciones dividiendo la cantidad de BSA en μg entre el total de la solución. Este paso no necesariamente tiene que hacerse en R. Debido a que no se conoce aún la cantidad de BSA de las soluciones problema, su concentración permanecerá como `NA`. Se sobrescribirá el mismo archivo `bsa`, para calcular la concentración, pero si se comete algún error siempre se puede volver a comenzar con el data frame original usando `read_csv()`.

Se sabe que la solución inicial usada para todas las observaciones (incluidas las desconocidas) tiene una concentración de $0.2 \mu\text{g}/\mu\text{L}$, de modo que para calcular la concentración de cada solución primero es necesario calcular los microgramos de BSA presentes en cada una. Esto puede hacerse multiplicando 0.2 por el volumen de BSA, los resultados se colocarán en una nueva columna usando `mutate()`:

```
bsa <- bsa %>%
  mutate(microg_bsa = 0.2 * volumen_bsa)

bsa

## # A tibble: 11 x 6
##   volumen_bsa    volumen_agua  volumen_tot tipo      absorbancia microg_bsa
##       <dbl>        <dbl>        <dbl> <fct>      <dbl>        <dbl>
## 1         0          100        100 conocida     0          0
## 2        12.5        87.5        100 conocida  0.106      2.5
## 3         25          75        100 conocida  0.159      5
## 4        37.5        62.5        100 conocida  0.265     7.5
## 5         50          50        100 conocida  0.35       10
## 6        62.5        37.5        100 conocida  0.452     12.5
## 7         75          25        100 conocida  0.509     15
## 8         NA          NA        50 problema_1 0.335     NA
## 9         NA          NA        25 problema_1 0.304     NA
```

```

## 10      NA      NA      80 problema_2      0.215      NA
## 11      NA      NA      60 problema_2      0.208      NA

```

La concentración en $\mu\text{g}/\mu\text{L}$ se calcula dividiendo los μg de BSA entre el volumen total:

```

bsa <- bsa %>%
  mutate(conc = microg_bsa / volumen_tot)
bsa

## # A tibble: 11 x 7
##   volumen_bsa volumen_agua volumen_tot tipo      absorbancia microg_bsa    conc
##       <dbl>      <dbl>      <dbl> <fct>      <dbl>      <dbl>    <dbl>
## 1        0        100        100 conocida      0          0     0
## 2      12.5       87.5       100 conocida    0.106      2.5  0.025
## 3        25         75       100 conocida    0.159      5    0.05
## 4      37.5       62.5       100 conocida    0.265      7.5  0.075
## 5        50         50       100 conocida    0.35       10    0.1
## 6      62.5       37.5       100 conocida    0.452     12.5  0.125
## 7        75         25       100 conocida    0.509      15   0.15
## 8        NA         NA       50 problema_1  0.335      NA   NA
## 9        NA         NA       25 problema_1  0.304      NA   NA
## 10       NA         NA       80 problema_2  0.215      NA   NA
## 11       NA         NA       60 problema_2  0.208      NA   NA

```

Para realizar la curva patrón primero es necesario crear un subconjunto que incluya únicamente las observaciones de las soluciones conocidas usando `filter()`:

```

curva_patron <- bsa %>%
  filter(tipo == "conocida")

curva_patron

## # A tibble: 7 x 7
##   volumen_bsa volumen_agua volumen_tot tipo      absorbancia microg_bsa    conc
##       <dbl>      <dbl>      <dbl> <fct>      <dbl>      <dbl>    <dbl>
## 1        0        100        100 conocida      0          0     0
## 2      12.5       87.5       100 conocida    0.106      2.5  0.025
## 3        25         75       100 conocida    0.159      5    0.05
## 4      37.5       62.5       100 conocida    0.265      7.5  0.075
## 5        50         50       100 conocida    0.35       10    0.1
## 6      62.5       37.5       100 conocida    0.452     12.5  0.125
## 7        75         25       100 conocida    0.509      15   0.15

```

Con estos datos se realiza un modelo lineal, cuyos parámetros se usarán para calcular la concentración de las soluciones problema. En este modelo lineal la concentración de BSA es la variable independiente y la absorción la variable dependiente.

La función necesaria es `lm()` (line model), en la cual se introducen las variables de la siguiente manera: `variable_dependiente ~ variable_independiente`. El operador `~` debe leerse como “en función de”:

```
modelo_bsa <- lm(data = bsa, formula = absorbancia ~ conc)
```

Es necesario guardar el modelo como un objeto debido a que el resultado consiste de múltiples elementos:

```

str(modelo_bsa)

## List of 13
## $ coefficients : Named num [1:2] 0.00479 3.44286
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "conc"
## $ residuals     : Named num [1:7] -0.004786 0.015143 -0.017929 0.002 0.000929 ...

```

```

## ..- attr(*, "names")= chr [1:7] "1" "2" "3" "4" ...
## $ effects      : Named num [1:7] -0.69583 0.45545 -0.01707 0.00541 0.00688 ...
## ..- attr(*, "names")= chr [1:7] "(Intercept)" "conc" "" "" ...
## $ rank         : int 2
## $ fitted.values: Named num [1:7] 0.00479 0.09086 0.17693 0.263 0.34907 ...
## ..- attr(*, "names")= chr [1:7] "1" "2" "3" "4" ...
## $ assign        : int [1:2] 0 1
## $ qr           :List of 5
##   ..$ qr    : num [1:7, 1:2] -2.646 0.378 0.378 0.378 0.378 ...
##   ...- attr(*, "dimnames")=List of 2
##     .. .$. : chr [1:7] "1" "2" "3" "4" ...
##     .. .$. : chr [1:2] "(Intercept)" "conc"
##   ...- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.38 1.22
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
## $ df.residual : int 5
## $ na.action   : 'omit' Named int [1:4] 8 9 10 11
## ..- attr(*, "names")= chr [1:4] "8" "9" "10" "11"
## $ xlevels     : Named list()
## $ call         : language lm(formula = absorbancia ~ conc, data = bsa)
## $ terms        :Classes 'terms', 'formula' language absorbancia ~ conc
## ..- attr(*, "variables")= language list(absorbancia, conc)
## ..- attr(*, "factors")= int [1:2, 1] 0 1
## ...- attr(*, "dimnames")=List of 2
##   .. .$. : chr [1:2] "absorbancia" "conc"
##   .. .$. : chr "conc"
##   ...- attr(*, "term.labels")= chr "conc"
##   ...- attr(*, "order")= int 1
##   ...- attr(*, "intercept")= int 1
##   ...- attr(*, "response")= int 1
##   ...- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   ...- attr(*, "predvars")= language list(absorbancia, conc)
##   ...- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   ...- attr(*, "names")= chr [1:2] "absorbancia" "conc"
## $ model        :'data.frame': 7 obs. of 2 variables:
##   ..$ absorbancia: num [1:7] 0 0.106 0.159 0.265 0.35 0.452 0.509
##   ..$ conc       : num [1:7] 0 0.025 0.05 0.075 0.1 0.125 0.15
##   ..- attr(*, "terms")=Classes 'terms', 'formula' language absorbancia ~ conc
##   ...- attr(*, "variables")= language list(absorbancia, conc)
##   ...- attr(*, "factors")= int [1:2, 1] 0 1
##   ...- attr(*, "dimnames")=List of 2
##     .. .$. : chr [1:2] "absorbancia" "conc"
##     .. .$. : chr "conc"
##   ...- attr(*, "term.labels")= chr "conc"
##   ...- attr(*, "order")= int 1
##   ...- attr(*, "intercept")= int 1
##   ...- attr(*, "response")= int 1
##   ...- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   ...- attr(*, "predvars")= language list(absorbancia, conc)
##   ...- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   ...- attr(*, "names")= chr [1:2] "absorbancia" "conc"

```

```

## ..- attr(*, "na.action")= 'omit' Named int [1:4] 8 9 10 11
## ... ..- attr(*, "names")= chr [1:4] "8" "9" "10" "11"
## - attr(*, "class")= chr "lm"

```

Para acceder a cada uno de estos es necesario usar el nombre del modelo seguido del operador `$`. En el vector `coefficients` se muestra primero la ordenada al origen (Intercept) y la pendiente, que siempre llevará el nombre de la variable independiente:

```
modelo_bsa$coefficients
```

```

## (Intercept)      conc
## 0.004785714 3.442857143

```

Estos valores son necesarios para sustituirlos en la ecuación de la recta: $y = mx + b$, en donde m y b corresponden a la pendiente y a la ordenada al origen respectivamente. Antes de continuar, el modelo también tiene información útil que sirve para validar el modelo y las predicciones realizadas a partir de él:

```
summary(modelo_bsa)
```

```

##
## Call:
## lm(formula = absorbancia ~ conc, data = bsa)
##
## Residuals:
##       1        2        3        4        5        6        7
## -0.0047857  0.0151429 -0.0179286  0.0020000  0.0009286  0.0168571 -0.0122143
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.004786   0.009693   0.494    0.642
## conc       3.442857   0.107536  32.016 5.58e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.01423 on 5 degrees of freedom
## (4 observations deleted due to missingness)
## Multiple R-squared:  0.9951, Adjusted R-squared:  0.9942
## F-statistic: 1025 on 1 and 5 DF,  p-value: 5.584e-07

```

Aquí se pueden observar no solamente los coeficientes previamente mencionados, sino también el valor de p , así como el valor de R^2 y R^2 ajustada. En este caso el valor de p es significativo e indica que la concentración de BSA tiene una relación estadísticamente significativa con la absorbancia. El valor de R^2 explica la proporción de la variación observada en la absorbancia que es explicada por el modelo, de modo que se busca un número cercano a 1.

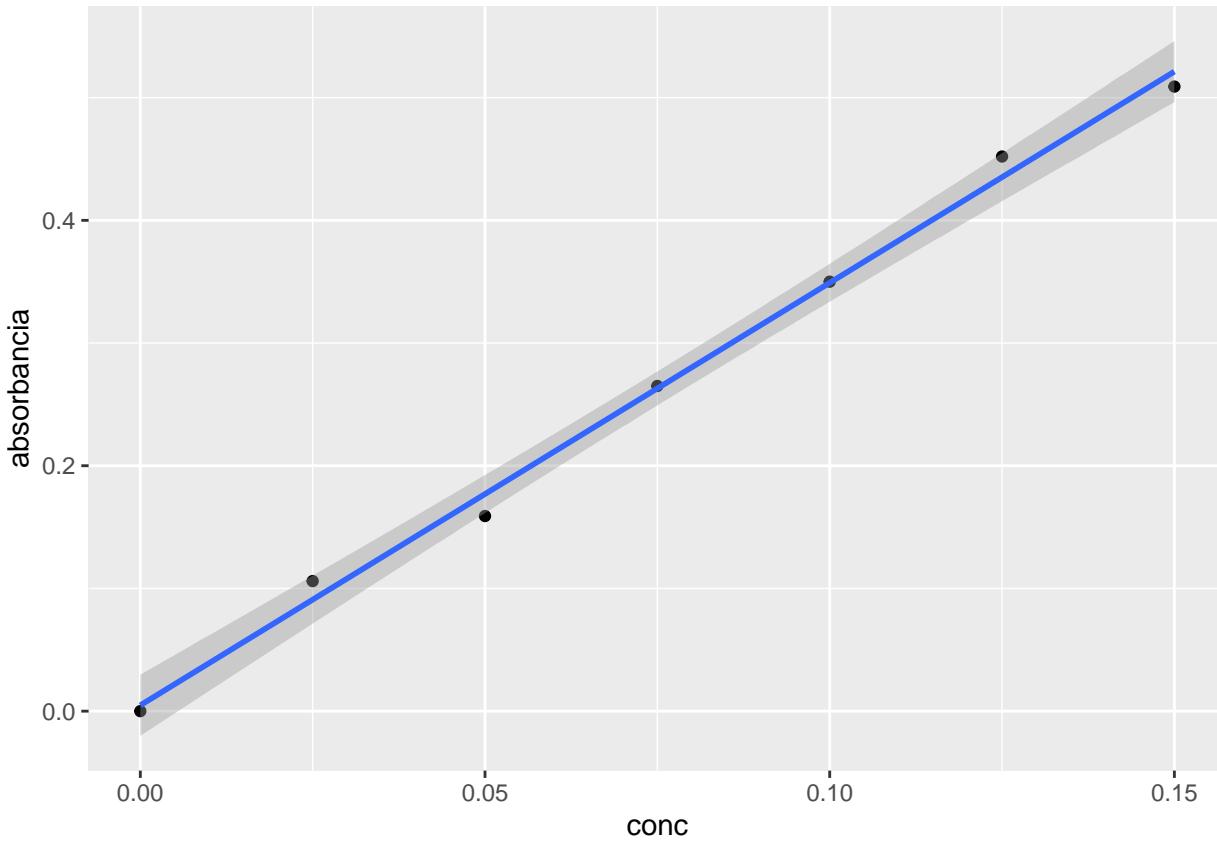
Para realizar la gráfica de la curva patrón es se usa `geom_point()` para graficar las observaciones y `geom_smooth()` para mostrar la recta del modelo:

```

ggplot(data = curva_patron, aes(x = conc, y = absorbancia)) +
  geom_point() +
  geom_smooth(method = "lm")

## `geom_smooth()` using formula 'y ~ x'

```



No es necesario usar el objeto `modelo_bsa` para crear la recta debido a que al especificar las mismas variables en `aes()` y `lm` en `method`, se está creando el mismo modelo.

En la ecuación de la recta:

$$y = mx + b$$

y representa la absorbancia y x la concentración de BSA. Los valores de m y b se obtuvieron a partir del modelo. Debido a que se desea conocer la concentración, se debe despejar x :

$$x = \frac{y - b}{m}$$

O bien:

$$\text{conc BSA} = \frac{\text{absorbancia} - b}{m}$$

Los valores de b y m pueden guardarse como objetos para evitar copiarlos manualmente. Se encuentran en el vector `coefficients` dentro de `modelo_bsa`, por lo que pueden ser accedidos con `[]`:

```
b <- modelo_bsa$coefficients[1]
m <- modelo_bsa$coefficients[2]
```

```
b; m
## (Intercept)
## 0.004785714
```

```
##      conc
## 3.442857
```

El punto y coma permite imprimir dos objetos al mismo tiempo. Ahora que ambos valores, están guardados como objetos diferentes, es posible usarlos dentro de `mutate()` para calcular la concentración. Primero es necesario separar los datos de las soluciones problema:

```
problema <- bsa %>%
  filter(tipo %in% c("problema_1", "problema_2"))

problema

## # A tibble: 4 x 7
##   volumen_bsa  volumen_agua  volumen_tot tipo      absorbancia microg_bsa  conc
##       <dbl>        <dbl>        <dbl> <fct>      <dbl>        <dbl> <dbl>
## 1       NA         NA          50 problema_1    0.335       NA     NA
## 2       NA         NA          25 problema_1    0.304       NA     NA
## 3       NA         NA          80 problema_2    0.215       NA     NA
## 4       NA         NA          60 problema_2    0.208       NA     NA
```

Usando `mutate()` se sustituyen los NA en `conc` cuyos valores se calculan a partir de la ecuación despejada:

```
problema <- problema %>%
  mutate(conc = (absorbancia - b) / m)

problema

## # A tibble: 4 x 7
##   volumen_bsa  volumen_agua  volumen_tot tipo      absorbancia microg_bsa  conc
##       <dbl>        <dbl>        <dbl> <fct>      <dbl>        <dbl> <dbl>
## 1       NA         NA          50 problema_1    0.335       NA  0.0959
## 2       NA         NA          25 problema_1    0.304       NA  0.0869
## 3       NA         NA          80 problema_2    0.215       NA  0.0611
## 4       NA         NA          60 problema_2    0.208       NA  0.0590
```

En caso de que el objetivo sea conocer únicamente la concentración de las soluciones problemas no se necesita hacer nadamás que terminar la gráfica. Sin embargo también es posible calcular el volumen de BSA inicial, los microgramos de BSA en cada solución y el volumen de agua. Partiendo de los valores recién calculados de la concentración podemos obtener la cantidad de microgramos de BSA usando la ecuación:

$$\text{concentracion} = \frac{\mu\text{g BSA}}{\mu\text{L}}$$

Se despejan los microgramos de BSA:

$$\mu\text{g BSA} = \text{concentracion} \bullet \mu\text{L}$$

Los microlitros de la ecuación corresponden al total de la solución, entonces los microgramos de BSA se calculan de la siguiente manera:

```
problema <- problema %>%
  mutate(microg_bsa = conc * volumen_tot)

problema

## # A tibble: 4 x 7
##   volumen_bsa  volumen_agua  volumen_tot tipo      absorbancia microg_bsa  conc
##       <dbl>        <dbl>        <dbl> <fct>      <dbl>        <dbl> <dbl>
## 1       NA         NA          50 problema_1    0.335       4.80 0.0959
```

## 2	NA	NA	25	problema_1	0.304	2.17 0.0869
## 3	NA	NA	80	problema_2	0.215	4.88 0.0611
## 4	NA	NA	60	problema_2	0.208	3.54 0.0590

Se sabe que la concentración de la solución original de BSA es de 0.2 $\mu\text{g}/\mu\text{L}$, por lo cual se puede calcular el volumen de BSA con la siguiente ecuación despejada:

$$\mu\text{L} = \frac{\mu\text{g BSA}}{\text{concentración}}$$

Al mismo tiempo se puede calcular el volumen de agua que se utilizó:

```
problema <- problema %>%
  mutate(volumen_bsa = microg_bsa / 0.2,
        volumen_agua = volumen_tot - volumen_bsa)
problema

## # A tibble: 4 x 7
##   volumen_bsa volumen_agua volumen_tot tipo      absorbancia microg_bsa conc
##       <dbl>       <dbl>       <dbl> <fct>       <dbl>       <dbl>   <dbl>
## 1     24.0       26.0       50.0 problema_1     0.335     4.80  0.0959
## 2     10.9       14.1       25.0 problema_1     0.304     2.17  0.0869
## 3     24.4       55.6       80.0 problema_2     0.215     4.88  0.0611
## 4     17.7       42.3       60.0 problema_2     0.208     3.54  0.0590
```

Finalmente, la concentración de cada solución puede aproximarse al promediar los valores de absorbancia obtenidos para cada solución. En este caso es necesario agrupar las observaciones de acuerdo con la variable tipo, por lo que se usan `group_by()` y `summarize`:

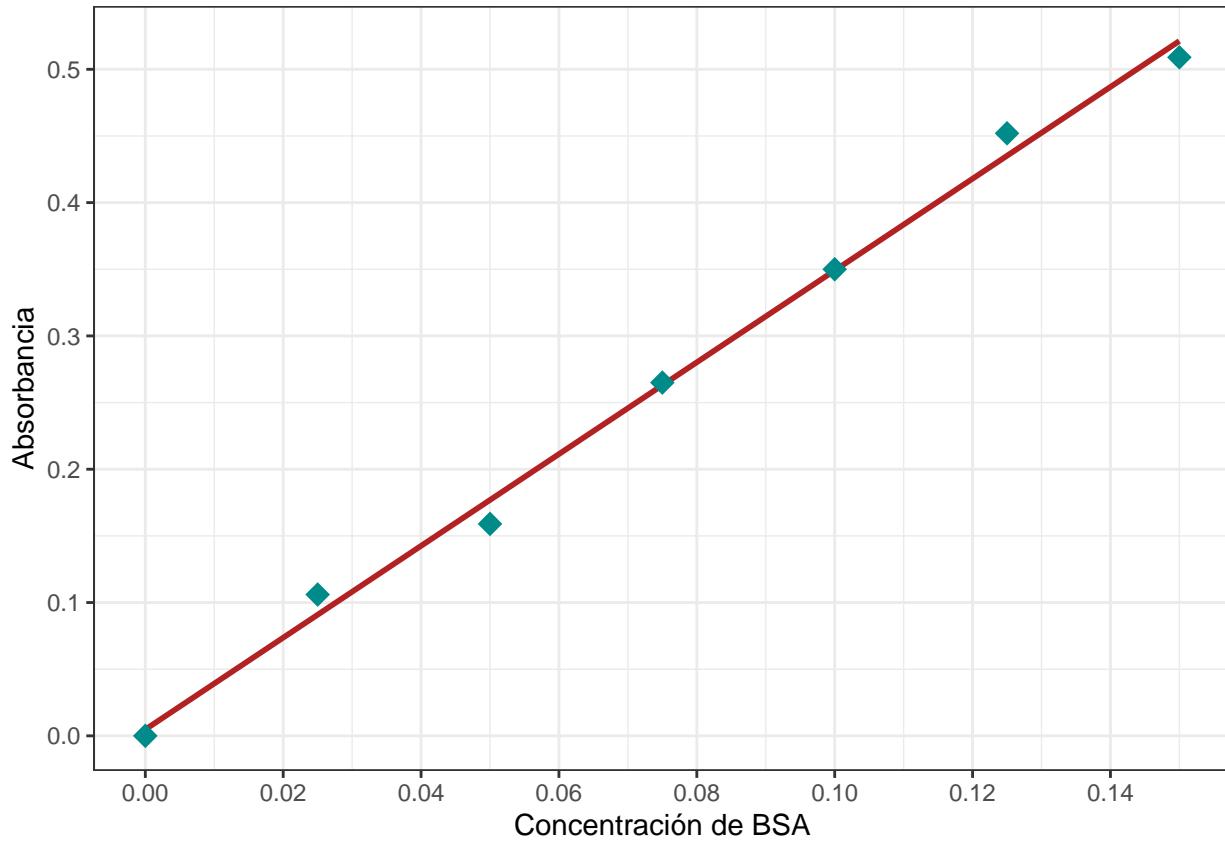
```
problema %>%
  group_by(tipo) %>%
  summarize(conc_prom = mean(conc))

## # A tibble: 2 x 2
##   tipo      conc_prom
##   <fct>       <dbl>
## 1 problema_1    0.0914
## 2 problema_2    0.0600
```

De este modo se tiene que las soluciones `problema_1` y `problema_2` tienen concentraciones aproximadas de .09 y .06 $\mu\text{g}/\mu\text{L}$, aproximadamente. La gráfica de la regresión lineal puede mostrarse así:

```
ggplot(data = curva_patron, aes(x = conc, y = absorbancia)) +
  geom_smooth(method = "lm", color = "firebrick", se = FALSE) +
  geom_point(size = 4, pch = 18, color = "cyan4") +
  labs(x = "Concentración de BSA",
       y = "Absorbancia") +
  scale_x_continuous(breaks = seq(from = 0, to = 1, by = .02)) +
  scale_y_continuous(breaks = seq(from = 0, to = 1, by = .1)) +
  theme_bw()

## `geom_smooth()` using formula 'y ~ x'
```



Otra manera de presentar la información podría ser incluyendo los datos de las soluciones problema para mostrar su posición dentro del modelo:

```
ggplot(data = curva_patron, aes(x = conc, y = absorbancia)) +
  geom_smooth(method = "lm", color = "firebrick", se = FALSE) +
  geom_point(size = 4, pch = 18, color = "deeppink2") +
  geom_point(data = problema, aes(x = conc, y = absorbancia,
                                    color = tipo),
             size = 4, pch = 15) +
  labs(x = "Concentración de BSA",
       y = "Absorbancia",
       color = NULL
      ) +
  scale_x_continuous(breaks = seq(from = 0, to = 1, by = .02)) +
  scale_y_continuous(breaks = seq(from = 0, to = 1, by = .1)) +
  scale_color_paletteer_d(palette = "awtools::a_palette",
                         labels = c("Problema 1", "Problema 2")) +
  theme(legend.position = "bottom") +
  theme_bw()

## `geom_smooth()` using formula 'y ~ x'
```

