BIO 4022. Manipulación de datos e investigación reproducible en R

Derek Corcoran

2018-08-22

Contents

Requerimientos

Para comenzar el trabajo se necesita la última versión de R y RStudio (R Core Team, 2018).También se requiere de los paquetes *pacman*, *rmarkdown*, *tidyverse* y *tinytex*. Si no se ha usado R o RStudio anteriormente, el siguiente video muestra cómo instalar ambos programas y los paquetes necesarios para este curso en el siguiente link.

El código para la instalación de esos paquetes es el siguiente:

```
install.packages("pacman", "rmarkdown", "tidyverse", "tinytex")
```

En caso de necesitar ayuda para la instalación, contactarse con el instructor del curso.

0.1 Antes de comenzar

Si nunca se ha trabajado con `R` antes de este curso, una buena herramienta es provista por el paquete Swirl (Kross et al., 2017). Para comenzar la práctica, realizar los primeros 7 modulos del programa *R Programming: The basics of programming in R* que incluye:

- Basic Building Blocks
- Workspace and Files
- Sequences of Numbers
- Vectors
- Missing Values
- Subsetting Vectors
- Matrices and Data Frames

El siguiente link muestra un video explicativo de cómo usar el paquete swirl Video

0.2 Descripción del curso

Este curso está enfocado en entregar principios básicos de investigación reproducible en R, con énfasis en la recopilación y/o lectura de datos de forma reproducible y automatizada. Para esto se trabajará con bases de datos complejas, las cuales deberán ser transformadas y organizadas para optimizar su análisis. Se generarán documentos reproducibles integrando en un documento: código, bibliografía, exploración y análisis de datos. Se culminará el curso con la generación de un manuscrito, una presentación y/o un documento interactivo reproducible.

0.3 Objetivos del curso

1. Conocer y entender el concepto de investigación reproducible como una forma y filosofía de trabajo que permite que las investigaciones sean más ordenadas y replicables, desde la toma de datos hasta la escritura de resultados.

2. Conocer y aplicar el concepto de pipeline, el cual permite generar una modularidad desde la toma de datos hasta la escritura de resultados, donde la corrección independiente de un paso tiene un efecto cascada sobre el resultado final.

3. Aprender buenas prácticas de recolección y estandarización de bases de datos, con la finalidad de optimizar el análisis de datos y la revisión de éstas por pares.

4. Realizar análisis críticos de la naturaleza de los datos al realizar análisis exploratorios, que permitirán determinar la mejor forma de comprobar hipótesis asociadas a estas bases de datos.

0.4 Contenidos

- Capítulo 1 *Tidy Data*: En este capítulo se aprenderá a cómo optimizar una de base de datos, sobre la limpieza y transformación de bases de datos, qué es una base de datos *tidy* y cómo manipular estas bases de datos con el paquete *dplyr* (**?**).

- Capítulo 2 *Investigación reproducible*: En este capítulo se trabajará en la confección de un documento que combine códigos de `R` y texto para generar documentos reproducibles utilizando el paquete *rmarkdown* (Allaire et al., 2018). Además, se verá cómo al usar RStudio se pueden guardar los proyectos en un repositorio de github.

- Capítulo 3 *El tidyverso* y el concepto de pipeline:En este capítulo se aprenderá sobre la limpieza de datos complejos.

- Capítulo 4 *Visualización de datos* visualizar datos vs. visualizar modelos. Insertar gráficos con leyenda en un documento Rmd

- Capítulo 6 *Loops*. Generación de funciones propias en R y loops

6. Escritura de manuscritos en R, transformación de documentos Rmd en un manuscrito

7. Presentaciones en R y generar documentos interactivos. Transformación de datos en una presentación o en una Shiny app. Realizar una presentación o aplicación en R.

0.5 Metodología

Todas las clases estarán divididas en dos partes: I. Clases expositivas de principios y herramientas, donde se presentarán los principios de investigación reproducible y tidy data, junto con las herramientas actuales más utilizadas, y II. Clases prácticas donde cada estudiante trabajará con datos propios para desarrollar un documento reproducible. Los estudiantes que no cuenten con datos propios podrán acceder a sets de datos para su trabajo o podrán simularlos, dependiendo del caso.

Además, se deberán generar informes y presentaciones siguiendo los principios de investigación reproducible, en base al trabajo con sus datos. Se realizará un informe final, en el cual se espera un trabajo que compile los conociminetos adquiridos durante el curso.

0.6 Evaluación

- Evaluación 1: Informe exploratorio de base de datos 25%
- Evaluación 2: Presentación 25%
- Evaluación 3: Informe final 50%

0.7 Libros de consulta

Los principios de este curso están explicados en los siguientes libros gratuitos.

- Gandrud, Christopher. Reproducible Research with R and R Studio. CRC Press, 2013. Available for free in the following link
- Stodden, Victoria, Friedrich Leisch, and Roger D. Peng, eds. Implementing reproducible research. CRC Press, 2014. Available for free in the following link

0.8 Bibliografía

Chapter 1

Tidy Data y manipulación de datos

1.1 Paquetes necesarios para este capítulo

Para este capitulo necesitas tener instalado el paquete *tidyverse*

En este capítulo se explicará qué es una base de datos *tidy* (Wickham et al., 2014) y se aprenderá a usar funciones del paquete *dplyr* (**?**) para manipular datos.

Dado que este libro es un apoyo para el curso BIO4022, esta clase del curso puede también ser seguida en este link. El video de la clase se encuentra disponible en este link.

1.2 Tidy data

Una base de datos tidy es una base de datos en la cuál (modificado de (Leek, 2015)):

- Cada vararible que se medida debe estar en una columna.
- Cada observación distinta de esa variable debe estar en una fila diferente.

En general, la forma en que representaríamos una base de datos *tidy* en `R` es usando un *data frame*.

1.3 dplyr

El paquete *dplyr* es definido por sus autores como una gramática para la manipulación de datos. De este modo sus funciones son conocidas como verbos. Un resumen útil de muchas de estas funciones se encuentra en este link.

Este paquete tiene un gran número de verbos y sería difícil ver todos en una clase, en este capítulo nos enfocaremos en sus funciones más utilizadas, las cuales son:

- *group_by* (agrupa datos)
- *summarize* (resume datos agrupados)
- *mutate* (genera variables nuevas)
- *%>%* (pipeline)
- *filter* (encuentra filas con ciertas condiciones)
- *select* junto a *starts_with*, *ends_with* o *contains*

Table 1.1: una tabla con 10 filas de la base de datos iris.

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.8	4.0	1.2	0.2	setosa
4.7	3.2	1.6	0.2	setosa
5.1	3.8	1.9	0.4	setosa
5.2	2.7	3.9	1.4	versicolor
6.4	2.9	4.3	1.3	versicolor
5.5	2.5	4.0	1.3	versicolor
6.5	3.0	5.8	2.2	virginica
6.0	2.2	5.0	1.5	virginica
6.1	2.6	5.6	1.4	virginica
5.9	3.0	5.1	1.8	virginica

Table 1.2: Resumen del promedio y desviación estándar del largo de pétalo de las flores del generi Iris.

Mean.Petal.Length	SD.Petal.Length
3.758	1.765298

1.3.1 summarize

La función `summarize` toma los datos de un data frame y los resume. Para usar esta función, el primer argumento que tomaríamos sería un data frame, se continúa del nombre que queremos darle a una variable resumen, seguida del signo = y luego la fórmula a aplicar a una o mas columnas. COmo un ejemplo se utilizará la base de datos `iris` (Anderson, 1935) que viene en `R` y de las cual podemos ver parte de sus datos en la tabla 1.1

Si quisieramos resumir esa tabla y generar un par de variables que fueran la media y la desviación estándar del largo del pétalo, lo haríamos con el siguiente código:

```
library(tidyverse)
Summary.Petal <- summarize(iris, Mean.Petal.Length = mean(Petal.Length),
    SD.Petal.Length = sd(Petal.Length))
```

El resultado se puedde ver en la tabla 1.2, en el cuál se obtienen los promedios y desviaciones estándar de los largos de los pétalos. Es importante notar que al usar summarize, todas las otras variables desapareceran de la tabla.

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

Table 1.3: Resumen del promedio y desviación estándar del largo de pétalo de las flores del generi Iris.

Species	Mean.Petal.Length	SD.Petal.Length
setosa	1.462	0.1736640
versicolor	4.260	0.4699110
virginica	5.552	0.5518947

Table 1.4: Millas por galón promedio en vehiculos automáticos (am = 0) y manuales (am = 1), con los distintos tipos de cilindros

cyl	am	Eficiencia
4	0	22.90000
4	1	28.07500
6	0	19.12500
6	1	20.56667
8	0	15.05000
8	1	15.40000

1.3.2 group_by

La función `group_by` por si sola no genera cambios visibles en las bases de datos. Sin embargo, al ser utilizada en conjunto con `summarize` permite resumir una variable agrupada (usualmente) basada en una o más variables categóricas.

Se puede ver que para el ejemplo con el caso de las plantas del género *Iris*, el resumen que se obtiene en el caso de la tabla 1.2 no es tan útil considerando que tenemos tres especies presentes. Si se quiere ver el promedio del largo del pétalo por especie, se debe ocupar la función `group_by` de la siguiente forma:

```
BySpecies <- group_by(iris, Species)
Summary.Byspecies <- summarize(BySpecies, Mean.Petal.Length = mean(Petal.Length),
    SD.Petal.Length = sd(Petal.Length))
```

Esto dá como resultado la tabla 1.3, con la cuál se puede ver que *Iris setosa* tiene pétalos mucho más cortos que las otras dos especies del mismo género.

1.3.2.1 group_by en más de una variable

Se puede usar la función `group_by` en más de una variable, y esto generaría un resumen anidado. Como ejemplo se usará la base de datos `mtcars` presente en R (Henderson and Velleman, 1981). Esta base de datos presenta una variable llamada *mpg* (miles per gallon) y una medida de eficiencia de combustible. Se resumirá la información en base a la variable *am* (que se refiere al tipo de transmisión, donde 0 es automático y 1 es manual) y al número de cilindros del motor. Para eso se utilizará el siguiente código:

```
Grouped <- group_by(mtcars, cyl, am)
Eficiencia <- summarize(Grouped, Eficiencia = mean(mpg))
```

Como puede verse en la tabla 1.4, en todos los casos los autos con cambios manuales tienen mejor eficiencia de combustible. Se podría probar el cambiar el orden de las variables con las cuales agrupar y observar los distintos resultados que se pueden obtener.

Table 1.5: Tabla con diez de las observaciones de la nueva base de datos con la variable nueva creada con mutate

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	Petal.Sepal.Ratio
5.8	4.0	1.2	0.2	setosa	0.21
4.7	3.2	1.6	0.2	setosa	0.34
5.1	3.8	1.9	0.4	setosa	0.37
5.2	2.7	3.9	1.4	versicolor	0.75
6.4	2.9	4.3	1.3	versicolor	0.67
5.5	2.5	4.0	1.3	versicolor	0.73
6.5	3.0	5.8	2.2	virginica	0.89
6.0	2.2	5.0	1.5	virginica	0.83
6.1	2.6	5.6	1.4	virginica	0.92
5.9	3.0	5.1	1.8	virginica	0.86

1.3.3 mutate

Esta función tiene como objetivo crear variables nuevas basadas en otras variables. Es muy facil de usar, como argumento se usa el nombre de la variable nueva que se quiere crear y se realiza una operación con variables que ya estan ahí. Por ejemplo, si se continúa el trabajo con la base de datos *Iris*, al crear una nueva variable que sea la razón entre el largo del pétalo y el del sépalo, resulta lo siguiente:

```
DF <- mutate(iris, Petal.Sepal.Ratio = Petal.Length/Sepal.Length)
```

El resultado de esta operación es la tabla 1.5. Siempre la variable que se acaba de crear aparecerá al final del data frame.

1.3.4 Pipeline (%>%)

El pipeline es un simbolo operatorio `%>%` que sirve para realizar varias operaciones de forma secuencial sin recurrir a parentesis anidados o a sobrescribir muúltiples bases de datos.

Para ver como funciona esto como un vector, supongamos que se tiene una variable a la cual se quiere primero obtener su logaritmo, luego su raíz cuadrada y finalmente su promedio con dos cifras significativas. Para realizar esto se debe seguir lo siguiente:

```
x <- c(1, 4, 6, 8)
y <- round(mean(sqrt(log(x))), 2)
```

Si se utiliza pipeline, el código sería mucho más ordenado. En ese caso, se partiría por el objeto a procesar y luego cada una de las funciones con sus argumentos si es necesario:

```
x <- c(1, 4, 6, 8)
y <- x %>% log() %>% sqrt() %>% mean() %>% round(2)
```

```
## [1] 0.99
```

El código con pipeline es mucho más fácil de interpretar a primera vista ya que se lee de izquierda a derecha y no de adentro hacia afuera. EL uso de pipeli se hace aun más importante cuando se usa con un *Data frame*, como se ve en el siguiente ejemplo:

Table 1.6: Razón pétalo sépalo promedio para las tres especies de Iris

Species	MEAN	SD
setosa	0.2927557	0.0347958
versicolor	0.7177285	0.0536255
virginica	0.8437495	0.0438064

Table 1.7: Símbolos lógicos de R y su significado

simbolo	significado	simbolo_cont	significado_cont
>	Mayor que	!=	distinto a
<	Menor que	%in%	dentro del grupo
==	Igual a	is.na	es NA
>=	mayor o igual a	!is.na	no es NA
<=	menor o igual a	| &	o, y

1.3.4.1 El pipeline en data frames

POr ejemplo se quiere resumir la variable recien creada de la razón entre el sépalo y el petalo. Para hacer esto, si se partiera desde la base de datos original, tomaría varias líneas de código y la creación de múltiples bases de datos intermedias

```
DF <- mutate(iris, Petal.Sepal.Ratio = Petal.Length/Sepal.Length)
BySpecies <- group_by(DF, Species)
Summary.Byspecies <- summarize(BySpecies, MEAN = mean(Petal.Sepal.Ratio),
    SD = sd(Petal.Sepal.Ratio))
```

Otra opción es usar paréntesis anidados, lo que se traduce en el siguiente código:

```
Summary.Byspecies <- summarize(group_by(mutate(iris, Petal.Sepal.Ratio = Petal.Length/Sepal.Length),
    Species), MEAN = mean(Petal.Sepal.Ratio), SD = sd(Petal.Sepal.Ratio))
```

Esto se simplifica mucho más al usar el pipeline, lo cual permite partir en un *Data Frame* y luego usar el pipeline. Esto permite obtener el mismo resultado que en las operaciones anteriores con el siguiente código:

```
Summary.Byspecies <- iris %>% mutate(Petal.Sepal.Ratio = Petal.Length/Sepal.Length) %>%
    group_by(Species) %>% summarize(MEAN = mean(Petal.Sepal.Ratio),
    SD = sd(Petal.Sepal.Ratio))
```

Estos tres códigos son correctos (tabla 1.6), pero definitivamente el uso del pipeline da el código más conciso y fácil de interpretar sin pasos intermedios.

1.3.5 filter

Esta función permite seleccionar filas que cumplen con ciertas condiciones, como tener un valor mayor a un umbral o pertenecer a cierta clase Los símbolos más típicos a usar en este caso son los que se ven en la tabla 1.7.

Por ejemplo si se quiere estudiar las características florales de las plantas del género *Iris*, pero no tomar en cuenta a la especie *Iris versicolor* se deberá usar el siguiente código:

Table 1.8: Resumen de la media de todas las características florales de las especies Iris setosa e Iris virginica

Species	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
setosa	5.006	3.428	1.462	0.246
virginica	6.588	2.974	5.552	2.026

Table 1.9: Número de plantas de cada especie con un largo de pétalo mayor a 4 y un largo de sépalo mayor a 5 centímetros

Species	N
versicolor	39
virginica	49

```
data("iris")
DF <- iris %>% filter(Species != "versicolor") %>% group_by(Species) %>%
    summarise_all(mean)
```

De esta forma se obtiene como resultado la tabla 1.8. En este caso se introduce la función `summarize_all` de `summarize`, la cual aplica la función que se le da como argumento a todas las variables de la base de datos.

Por otro lado si se quiere estudiar cuántas plantas de cada especie tienen un largo de pétalo mayor a 4 y un largo de sépalo mayor a 5 se deberá usar el siguiente código:

```
DF <- iris %>% filter(Petal.Length >= 4 & Sepal.Length >= 5) %>%
    group_by(Species) %>% summarise(N = n())
```

En la tabla tabla 1.9 se ve que con este filtro desaparecen de la base de datos todas las plantas de *Iris setosa* y que todas menos una planta de *Iris virginica* tienen ambas características.

1.3.6 select

Esta función permite seleccionar las variables a utilizar dado que en muchos casos nos encontraremos con bases de datos con demasiadas variables y por lo tanto, se querrá reducirlas para solo trabajar en una tabla con las variables necesarias.

Con select hay varias formas de trabajar, por un lado se puede escribir las variables que se utilizarán, o restar las que no. En ese sentido estos cuatro códigos dan exactamente el mismo resultado. Esto se puede ver en la tabla 1.10

```
iris %>% group_by(Species) %>% select(Petal.Length, Petal.Width) %>%
    summarize_all(mean)
```

```
iris %>% group_by(Species) %>% select(-Sepal.Length, -Sepal.Width) %>%
    summarize_all(mean)
```

```
iris %>% group_by(Species) %>% select(contains("Petal")) %>%
    summarize_all(mean)
```

Table 1.10: Promedio de largo de pétalo y ancho de pétalo para las especies del genero Iris

Species	Petal.Length	Petal.Width
setosa	1.462	0.246
versicolor	4.260	1.326
virginica	5.552	2.026

```
iris %>% group_by(Species) %>% select(-contains("Sepal")) %>%
    summarize_all(mean)
```

1.3.7 Ejercicios

1.3.7.1 Ejercicio 1

Usando la base de datos `storms` del paquete *dplyr*, calcular la velocidad promedio y diámetro promedio (hu_diameter) de las tormentas que han sido declaradas huracanes para cada año.

1.3.7.2 Ejercicio 2

La base de datos `mpg` del paquete ggplot2 tiene datos de eficiencia vehicular en millas por galón en ciudad (*cty*) en varios vehículos. Obtener los datos de vehículos del año 2004 en adelante que sean compactos y transformar la eficiencia Km/litro (1 milla = 1.609 km; 1 galón = 3.78541 litros)

Las soluciones a estos ejercicios se encuentran en el capítulo 8

Chapter 2

Investigación reproducible

2.1 Paquetes necesarios para este capítulo

Para este capítulo se necesita tener instalado los paquetes *rmarkdown*, *knitr* y *stargazer*

En este capítulo se explicará qué es investigación reproducible, cómo aplicarla usando github más los paquetes *rmarkdown* (Allaire et al., 2018) y *knitr* (Xie, 2015). Además, se aprenderá a usar tablas usando *knitr* (Xie, 2015) y *stargazer* (Hlavac, 2018)

Recuerda que este libro es un apoyo para el curso BIO4022, puedes seguir la clase de este curso en este link, y en cuanto el video de la clase encontrarás un link aca.

2.2 Investigación reproducible

La investigación reproducible no es lo mismo que la investigación replicable. La replicabilidad implica que experimentos o estudios llevados a cabo en condiciones similares nos llevarán a conclusiones similares. La investigación reproducible implica que desde los mismos datos y/o el mismo código se generarán los mismos resultados.

En la figura 2.1 vemos el continuo de reproducibilidad (Peng, 2011). En este continuo tenemos el ejemplo de no reproducibilidad como una publicación sin código. Se pasa de menos a más reproducible por la publicación y el código que generó los resultados y gráficos; seguido por la publicación, el código y los datos que generan los resultados y gráficos; y por último código, datos y texto entrelazados de forma tal que al correr el código obtenemos exactamente la mismma publicación que leímos.

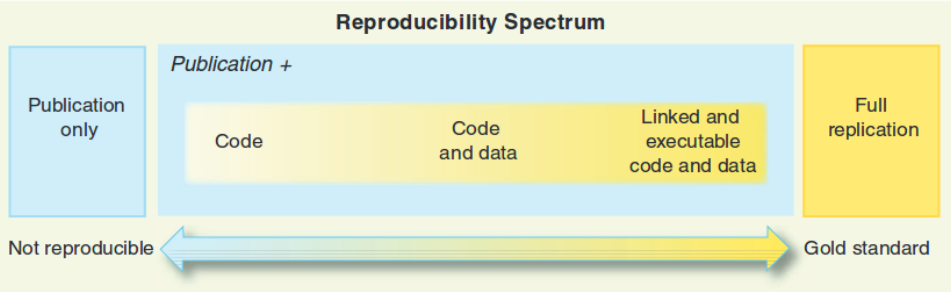

Figure 2.1: Continuo de reproducibilidad (extraido de Peng 2011)

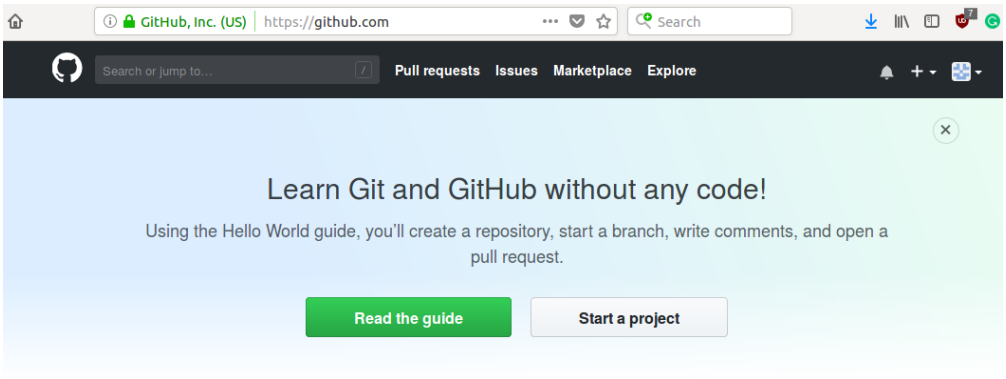

Figure 2.2: Para empezar un projecto en github, debes presionar Start a project en tu página de inicio

Esto tiene muchas ventajas, incluyendo el que es más fácil aplicar exactamente los mismos métodos a otra base de datos. Basta poner la nueva base de datos en el formato que tenía el autor de la primera publicación y podremos comparar los resultados.

Además en un momento en que la ciencia está basada cada vez más en bases de datos, se puede poner en el código la recolección y/o muestreo de datos.

2.3 Guardando nuestro proyecto en github

2.3.1 Que es github?

Github es una suerte de dropbox o google drive pensado para la investigación reproducible, en donde cada proyecto es un *repositorio*. La mayoría de los investigadores que trabajan en investigación reproducible dejan todo su trabajo documentado en sus repositorios, lo cual permite interactuar con otros autores.

2.3.2 creando un proyecto de github en RStudio

Para crear un proyecto en github presionamos **start a project** en la página inicial de nuestra cuenta, como vemos en la figura 2.2

Luego se debe crear un nombre único, y sin cambiar nada más presiona **create repository** en el botón verde como vemos en la figura 2.3.

Esto te llevará a una página donde aparecerá una url de tu nuevo repositorio como en la figura 2.4

Para incorporar tu proyecto en tu repositorio, lo primero que debes hacer es generar un proyecto en RStudio. Para esto debes ir en el menú superior de *Rstudio* a *File* > *New Project* > *Git* como se ve en las figuras 2.5 y 2.5.

Luego seleccionar la ubicación del proyecto nuevo y pegar el url que aparece en la figura 2.4 en el espacio que dice **Repository URL:**, como muestra en la figura 2.7.

Cuando tu proyecto de R ya este siguiendo los cambios en github, te aparecerá una pestaña git dentro de la ventana superior derecha de tu sesión de RStudio, tal como vemos en la figura 2.8

2.3.3 Los tres principales pasos de un repositorio

Github es todo un mundo, existen muchas funciones y hay expertos en el uso de github. En este curso, nos enfocaremos en los 3 pasos principales de un repositorio: *add*, *commit* y *push*. Para entender bien qué

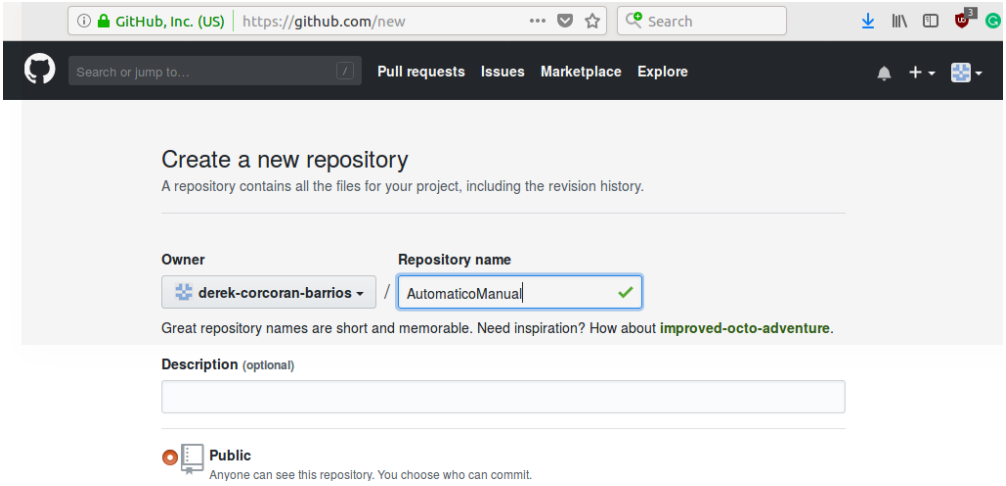

Figure 2.3: Crea el nombre de tu repositorio y apreta el boton create repository

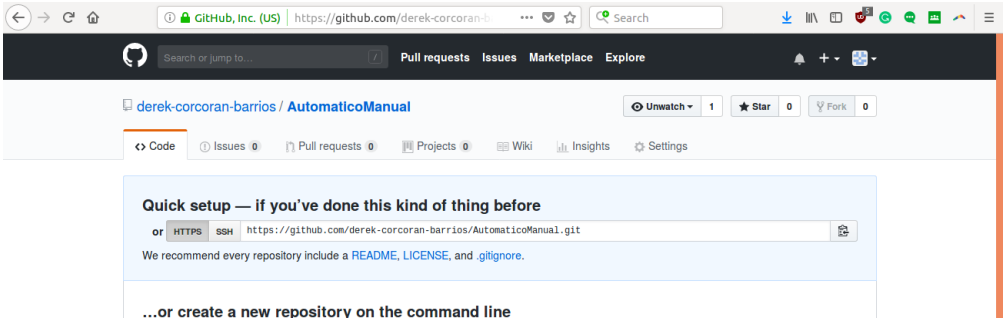

Figure 2.4: El contenido del cuadro en el cual dice ssh es la url de tu repisitorio

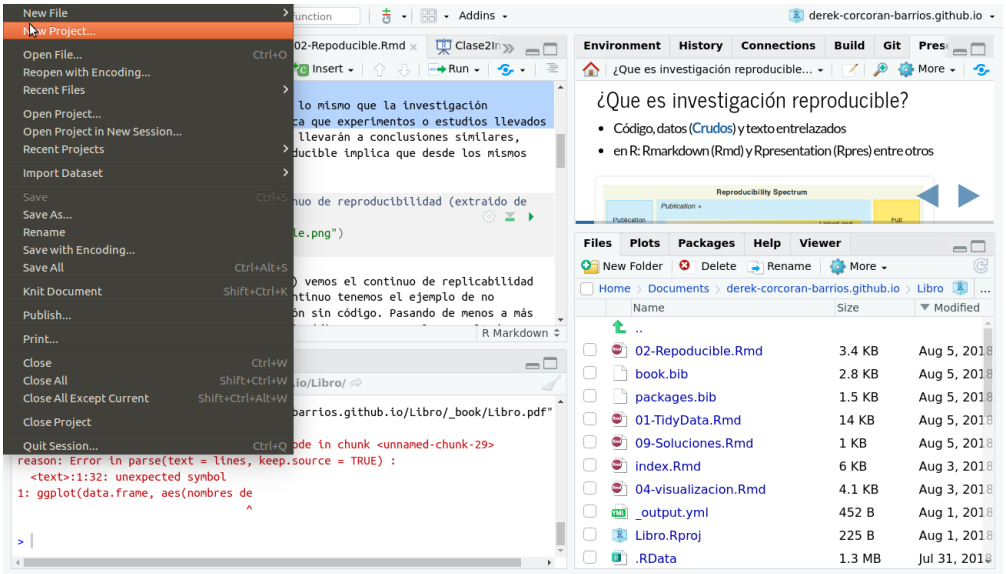

Figure 2.5: Menú para crear un proyecto nuevo

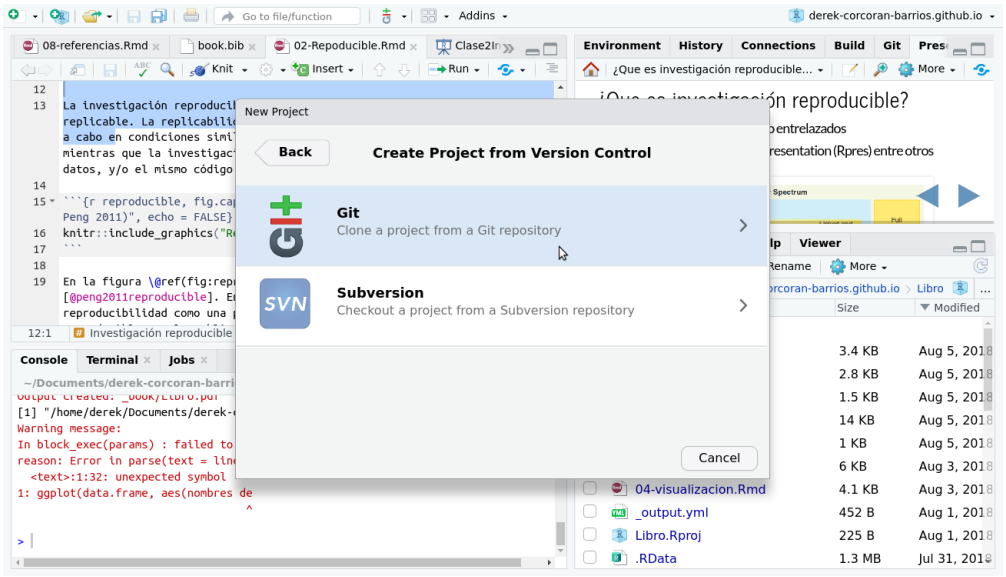

Figure 2.6: Seleccionar git dentro de las opciones

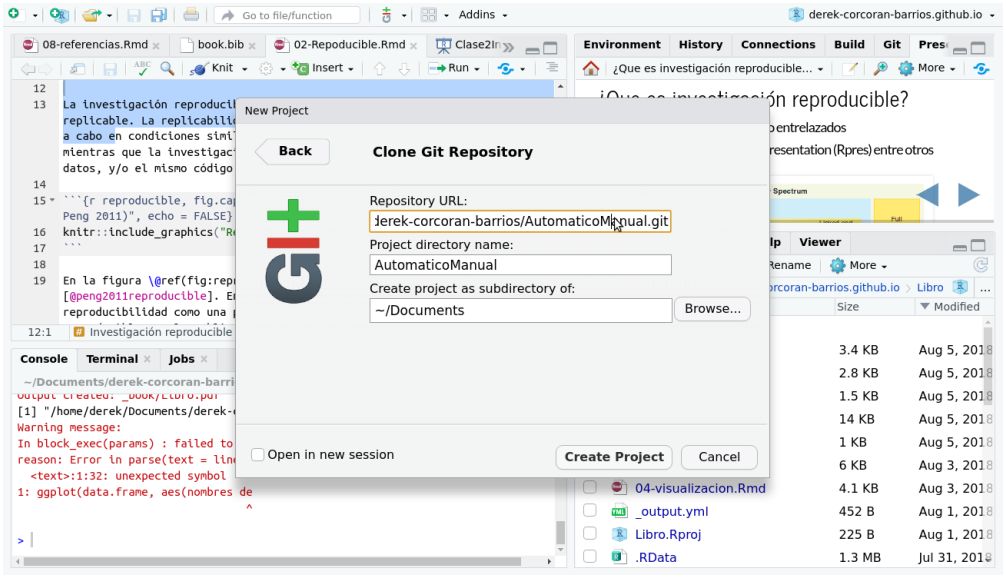

Figure 2.7: Pegar el url del repositorio en el cuadro de dialogo Repository URL:

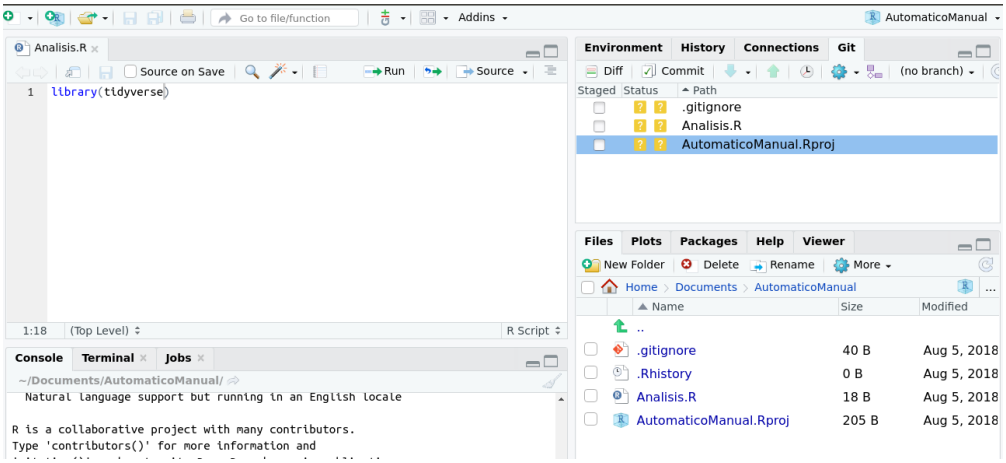

Figure 2.8: Al incluir tu repositorio en tu sesión de Rstudio, aparecera la pestaña git en la ventana superior derecha

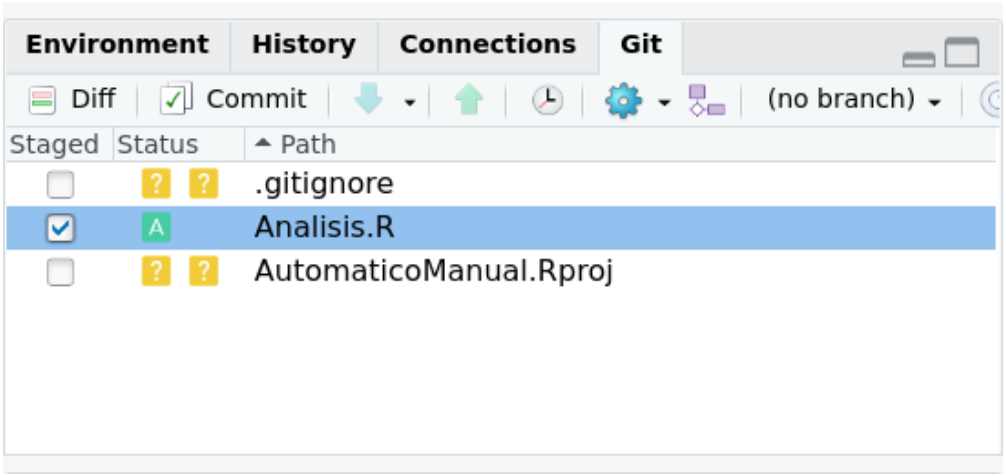

Figure 2.9: Al incluir tu repositorio en tu sesión de Rstudio, aparecera la pestaña git en la ventana superior derecha

significa cada uno de estos pasos, tenemos que entender que existen dos repositorios en todo momento: uno local (en tu computador) y otro remoto (en github.com). Los dos primeros pasos *add* y *commit*, solo generan cambios en tu repositorio local. Mientras que *push*, salva los cambios al repositorio remoto.

2.3.3.1 git add

Esta función es la que agrega archivos a tu repositorio local. Solo estos archivos serán guardados en github. Github tienen un límite de tamaño de repositorio de 1 GB y de archivos de 100 MB, ya que si bien te dan repositorios ilimitados, el espacio de cada uno no lo es, en particular en cuanto a bases de datos. Para adicionar un archivo a tu repositorio tan solo debes selecionar los archivos en la pestaña git. Al hacer eso una letra A verde aparecerá en vez de los dos signos de interrogación amarillos, como vemos en la figura 2.9. En este caso solo adicionamos al repositorio el archivo *Analisis.r* pero no el resto.

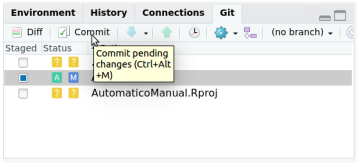

Figure 2.10: Para guardar los cambios en tu repositorio apretar commit en la pestaña git de la ventana superior derecha

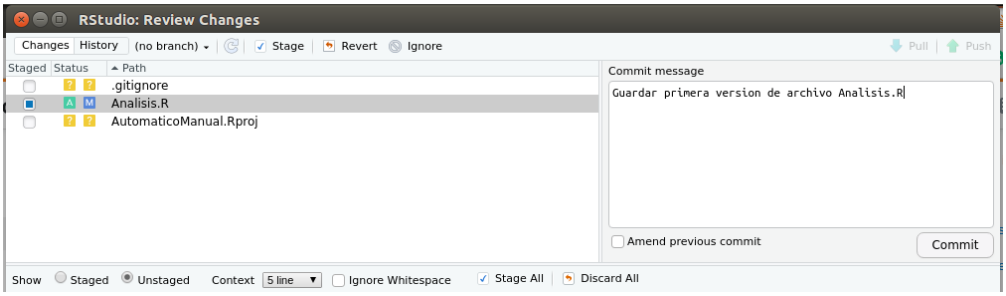

Figure 2.11: Escribir un mensaje que recuerde los cambios que hiciste en la ventana emergente

2.3.3.2 git commit

Cuando ocupas el comando *commit* estas guardando los cambios de los archivos que adicionaste en tu repositorio local. Para hacer esto en Rstudio, en la misma pestaña de git, debes presionar el botón *commit* como vemos en la figura 2.10.

Al presionar *commit*, se abrirá una ventana emergente, donde deberás escribir un mensaje que describa lo que guardarás. Una vez echo eso, presiona *commit* nuevamente en la ventana emergente como aparece en la figura 2.11.

2.3.3.3 git push

Finalmente, *push* te permitirá guardar los cambios en tu repositorio remoto, lo cual asegura tus datos en la nube y además lo hace disponible a otros investigadores. Luego de apretar *commit* en la ventana emergente (figura 2.11), podemos presionar *push* en la flecha verde de la ventana emergente como se ve el a figura 2.12. Luego se nos pedirá nuestro nombre de usuario y contraseña, y ya podemos revisar que nuestro repositorio esta online entrando a nuestra sesión de github.

2.4 Reproducibilidad en R

Existen varios paquetes que permiten que hagamos investigación reproducible en `R`, pero sin duda los más relevantes son *rmarkdown* y *knitr*. Ambos paquetes funcionan en conjunto cuando generamos un archivo *Rmd* (Rmarkdown), en el cual ocupamos al mismo tiempo texto, código de R y otros elementos para generar un documento word, pdf, página web, presentación y/o aplicación web (fig 2.13).

2.4.1 Creando un Rmarkdown

Para crear un archivo Rmarkdown, simplemente ve a el menu *File* > *New file* > *Rmarkdown* y con eso habrás creado un nuevo archivo *Rmd*. Veremos algunos de los elementos más típicos de un archivo Rmarkdown.

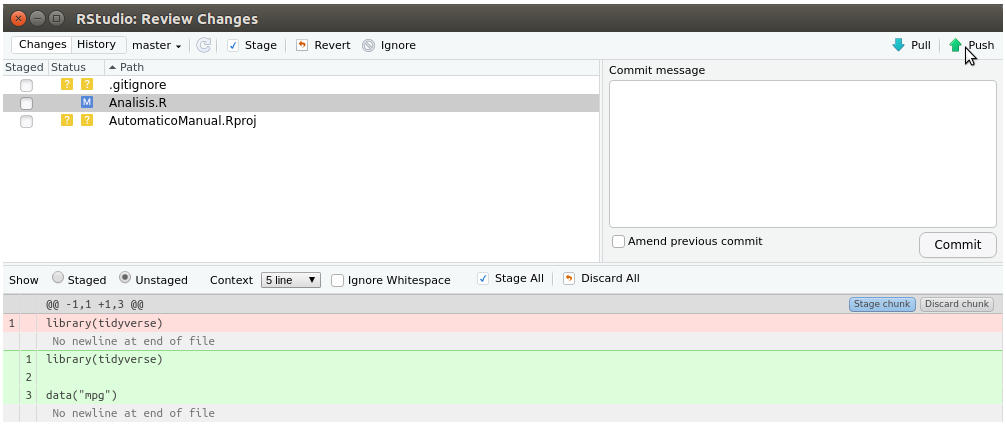

Figure 2.12: Para guardar en el repositorio remoto apretar push en la ventana emergente

Figure 2.13: El objetivo de Rmarkdown es el unir código de r con texto y datos para generar un documento reproducible

2.4.1.1 Markdown

El markdown es la parte del archivo en que simplemente escribimos texto, aunque tiene algunos detalles para el formato como generar texto en negrita, cursiva, títulos y subtitulos.

Para hacer que un texto este en **negrita**, se debe poner entre dos asteriscos `**negrita**`, para que un texto aparezca en *cursiva* debe estar entre asteriscos `*cursiva*`. Otros ejemplos son los títulos de distintos niveles, los cuales se denotan con distintos números de `#`, así los siguientes 4 títulos o subtítulos:

subtitulo 1

subtítulo 2

subtítulo 3

2.4.1.1.1 subtítulo 4

se vería de la siguiente manera en el código

```
## subtitulo 1

### subtítulo 2

#### subtítulo 3

##### subtítulo 4
```

2.4.1.2 Chunks

Los *chunks* son una de las partes más importantes del un Rmarkdown. En estos es donde se agrega el código de R (u otros lenguajes de programación). Lo cual permíte que el producto de nuestro código no sea sólo un escrito con resultados pegados, sino que efectivamente generados en el mismo documento que nuestro escrito. La forma más fácil de agregar un chunk es apretando el botón de *insert chunk* en Rstudio, este boton se encuentra en la ventana superior izquierda de nuestra sesión de RStudio, tal como se muestra en la figura 2.14

Al apretar este botón aparecera un espacio, ahí se puede agregar un código como el que aparece a continuación, y ver a continuación los resultados.

````
```{r}
library(tidyverse)
iris %>% group_by(Species) %>% summarize(Petal.Length = mean(Petal.length))
```
````

```
## # A tibble: 3 x 2
##   Species    Petal.Length
##   <fct>             <dbl>
## 1 setosa             1.46
## 2 versicolor         4.26
## 3 virginica          5.55
```

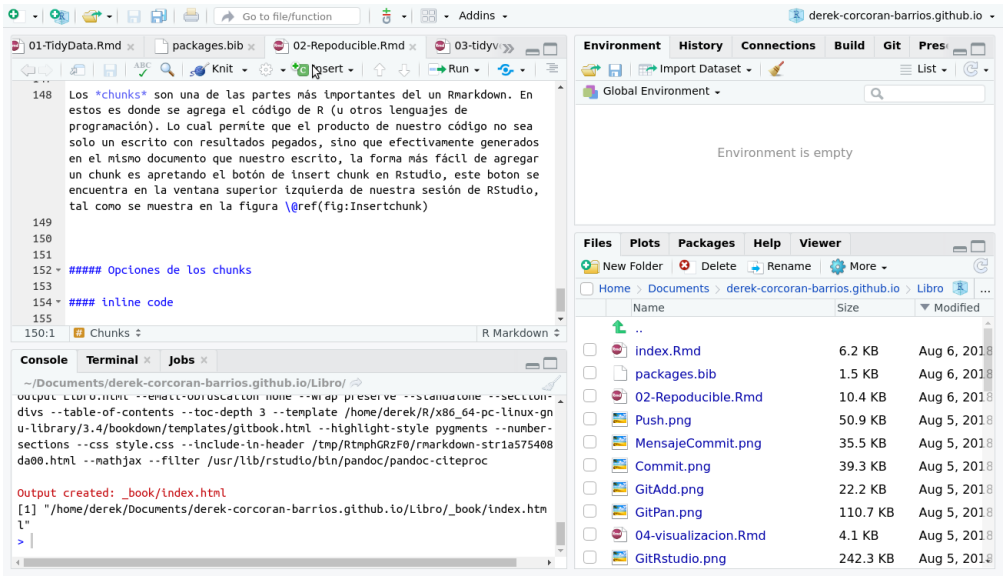

Figure 2.14: Al apretar el botón insert chunk, aparecera un espacio en el cuál insertar código

2.4.1.2.1 Opciones de los chunks

Existen muchas opciones para los chunks, una documentación completa podemos encontrarle en el siguiente link, pero acá mostraremos los más comunes:

- *echo* = T o F muestro o no el código, respectivamente
- *message* = T o F muestra mensajes de paquetes, respectivamente
- *warning* = T o F muestra advertencias, respectivamente
- *eval* = T o F evaluar o no el código, respectivamente
- *cache* = T o F guarda o no el resultado, respectivamente

2.4.1.3 inline code

Los *inline codes* son útiles para agregar algún valor en el texto, como por ejemplo el valor de p o la media. Para usarlo, se debe poner un backtick (comilla simple hacia atrás), r, el código en cuestion y otro backtick como se ve a continuación `` `r R_código` ``. No podemos poner cualquier cosa en un *inline code*, ya que sólo puede generar vectores, lo cuál muchas veces requiere de mucha creatividad para lograr lo que queremos. Por ejemplo si se quisiera poner el promedio del largo del sépalo de la base da dato `iris` en un *inline code* pondríamos `` `r mean(iris$Sepal.Length)` ``, lo cual resultaría en 5.8433333. Como en un texto se vería extraño un número con 7 cifras significativas, querríamos usar ademas la función `round`, para que tenga 2 cifras significativas, para eso ponemos el siguiente inline code `` `r round(mean(iris$Sepal.Length),2)` `` que da como resultado 5.84. Esto se puede complejizar más aún si se quiere trabajar con una tabla resumen. Por ejemplo, si quisieramos listar el promedio del tamaño de sépalo usaríamos `summarize` de *dplyr*, pero esto nos daría como resultado un data.frame, el cual no aparece si intentamos hacer un inline code. Partamos por ver como se vería el código donde obtuvieramos la media del tamaño del sépalo.

```
iris %>% group_by(Species) %>% summarize(Mean = mean(Sepal.Length))
```

El resultado de ese código lo veríamos 2.1

Para sacar de este data frame el vector de la media podríamos subsetearlo con el signo `$`. Entonces si queremos sacar como vector la columna *Mean* del data frame que creamos, haríamos lo siguiente `` `r (iris %>% ``

Table 2.1: Resumen del promedio del largo de sépalo de las flores del genero Iris.

Species	Mean
setosa	5.006
versicolor	5.936
virginica	6.588

Table 2.2: Promedio por especie de todas las variables de la base de datos iris.

Species	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
setosa	5.006	3.428	1.462	0.246
versicolor	5.936	2.770	4.260	1.326
virginica	6.588	2.974	5.552	2.026

`group_by(Species) %>% summarize(Mean = mean(Sepal.Length)))$Mean`. Esto daría como resultado 5.006, 5.936, 6.588.

2.4.2 Ejercicios

2.4.2.1 Ejercicio 1

Usando la base de datos *iris*, crea un inline code que diga cuál es la media del largo del pétalo de la especie *Iris virginica*

La solución a este ejercicio se encuentra en el capítulo 8

2.4.2.2 Tablas en Rmarkdown

La función más típica para generar tablas en un archivo *rmd* es `kable` del paquete *knitr*, que en su forma más simple se incluye un dataframe como único argumento. Además de esto, podemos agregar algunos parámetros como *caption*, que nos permite poner un título a la tabla o *row.names*, que si se pone como se ve en el código (FALSE) no mostrará en la tabla los nombres de las filas, tal como se ve en la tabla 2.2.

```
DF <- iris %>% group_by(Species) %>% summarize_all(mean)
kable(DF, caption = "Promedio por especie de todas las variables de la base de datos iris.",
    row.names = FALSE)
```

Chapter 3

El Tidyverso y tidyr

3.1 Paquetes necesarios para este capítulo

Para este capítulo necesitas tener instalado el paquete *tidyverse* y el paquete *dismo* para uno de los ejercicios.

En este capítulo se explicará qué es el paquete *tidyverse* (Wickham, 2017) y cuales son sus componentes. Además veremos las funciones del paqute *tidyr* (Wickham and Henry, 2018) con sus dos funciones `gather` y `spread`.

Dado que este libro es un apoyo para el curso BIO4022, esta clase puede también ser seguida en este link. El video de la clase se encontrará disponible en este link.

3.2 El tidyverso

El tidiverso se refiere al paquete tidiverse, el cual es una colección de paquetes coehrentes, que tienen una gramática, filosofía y estructura similar. Todos se basan en la idea de tidy data propuesta por Hadley Wickham (Wickham et al., 2014).

Los paquetes que forman parte del tidyverso son:

- readr (ya la estamos usando)
- dplyr (Clase anterior)
- tidyr (Hoy)
- ggplot2 (Próxima clase)
- purrr (En clase sobre loops)
- forcats (Para variables categóricas)
- stringr (Para carácteres, Palabras)

3.2.1 readr

El paquete *readr* (Wickham et al., 2017) tiene como función el importar (leer) y exportar archivos. Dado que en general nosotros usaremos archivos del tipo *csv*, para este tipo de archivos, *readr* tiene la función `read_csv`. Para exportar un archivo ocupamos la función `write_csv`. Ambas funciones son 10 veces más rápidas que las versiones de r base. Para más información sobre este revisar su página oficial.

3.2.2 dplyr

Este paquete sirve para modificar variables y sus detalles los vimos en el capítulo 1. Los cinco verbos principales que tiene son `mutate` para generar nuevas variables y que vienen de variables ya existentes, `select` para seleccionar variables basadas en su nombre, `filter` para seleccionar filas de acuerdo a si cumplen o no con condiciones en una o mas variables, `summarize` para resumir las variables, y `arrange` para reordenar las filas de acuerdo a alguna variable. Para más información sobre este paquete revisar su página oficial.

3.2.3 tidyr

Con sólo dos funciones: `gather` y `spread`. El paquete *tidyr* (Wickham and Henry, 2018) tiene como finalidad el tomar bases de datos no tidy y transformalas en tidy (datos limpios y ordenados). Para esto, `gather` transforma tablas anchas en largas y `spread` transforma tablas anchas en larga. En este capítulo explicaremos en más detalle estos dos verbos. Para más información sobre este paquete revisar su página oficial.

3.2.4 ggplot2

Una vez que una base de datos está en formato tidy, podemos usar *ggplot2* (Wickham, 2016) para visualizar estos datos. Los datos pueden ser categóricos, continuos e incluso espaciales en conjunto con el paquete *sf*. Este paquete es el más antiguo del *tidyverse* y por ello posee una gramática un poco diferente. Hablaremos más de este paquete en el capítulo 4. Por ahora si se quiere aprender más sobre *ggplot2* se puede revisar la página oficial

3.2.5 purrr

Purrr (Henry and Wickham, 2018) permite formular loops de una forma más sencilla e intuitiva que los **for** loops. Utilizando sus funciones `map`, `map2`, `walk` y `reduce` podemos realizar loops dentro de la gramática del tidyverse. Trabajaremos en este paquete en el capítulo 6. Como siempre puedes encontras más información en su página oficial

3.2.6 forcats

Trabajar con factores es una de las labores más complejas en R, es por eso que se creó el paquete *forcats* (Wickham, 2018a). Si bien no hay un capítulo en este libro en el cuál se trabajará exclusivamente con este paquete, se utilizará al menos una función en el capítulo 4

3.2.7 stringr

El modíficar variables de texto para hacer que las variables tengan sentido humano es algo muy importante, para este tipo de modificaciones se utiliza el paquete *stringr* (Wickham, 2018b). En este capítulo, para algunos ejercicios, introduciremos algunas funcionalidades de este paquete. Para más información revisar su página oficial.

3.3 tidyr

Este paquete como ya fue explicado en la sección anterior, solo posee dos funciones: `gather` y `spread`. Estas funciones sirven para pasar de tablas anchas a largas y viceversa, pero ¿qué significa que la misma información sea presentada en una tabla larga o en una tabla ancha?

Table 3.1: Tabla ancha.

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
7.0	3.2	4.7	1.4	versicolor
6.3	3.3	6.0	2.5	virginica

Table 3.2: Tabla larga.

Species	Atributos_florales	Medidas
setosa	Sepal.Length	5.1
versicolor	Sepal.Length	7.0
virginica	Sepal.Length	6.3
setosa	Sepal.Width	3.5
versicolor	Sepal.Width	3.2
virginica	Sepal.Width	3.3
setosa	Petal.Length	1.4
versicolor	Petal.Length	4.7
virginica	Petal.Length	6.0
setosa	Petal.Width	0.2
versicolor	Petal.Width	1.4
virginica	Petal.Width	2.5

Tomemos por ejemplo dos tablas. En la tabla 3.1 vemos una tabla ancha y en la tabla 3.2 una tabla larga.

3.3.0.1 DATO

Usualmente las tablas anchas son mejores para ser mostradas ya que se distinguen más facilmente las variables trabajadas, mientras que las tablas largas son mejores para programar y hacer análisis.

3.3.1 gather

Esta función nos permite pasar de una tabla ancha a una larga. En muchos casos esto es necesario para generar una base de datos *tidy*, y en otras ocaciones es importante para generación de gráficos que necesitamos tal como veremos en el capítulo 4. En esta función partimos con un data frame y luego tenemos 3 argumentos: en el primero `key`, ponemos el nombre de la variable que va a llevar como observaciones los nombres de las columnas; luego en el argumento *value*, ponemos el nombre de la columna que llevará los valores de cada columna al transformarse en una columna larga; Por último hay un argumento (sin nombre), en el cual ponemos las columas que queremos que sean "*alargadas*", o con un signo negativo, las que no queremos que sean parte de esta transformación. Todo esto quedará más claro en el siguiente ejemplo.

3.3.1.1 Ejemplo de los censos

Supongamos que un estudiante de biología va a realizar un censo en un parque nacional por tres días y genera la siguiente tabla (el código a continuación es el que permite generar el data frame obervado en la tabla 3.3)

Table 3.3: Abundancia detectada por especie en tres días de muestreo

dia	Lobo	Liebre	Zorro
Lunes	2	20	4
Martes	1	25	4
Miercoles	3	30	4

Table 3.4: Abundancia detectada por especie en tres días de muestreo

Columnas	Valores
dia	Lunes
dia	Martes
dia	Miercoles
Lobo	2
Lobo	1
Lobo	3
Liebre	20
Liebre	25
Liebre	30
Zorro	4
Zorro	4
Zorro	4

```
df_cuentas <- data.frame(dia = c("Lunes", "Martes", "Miercoles"),
    Lobo = c(2, 1, 3), Liebre = c(20, 25, 30), Zorro = c(4, 4,
        4))
```

Claramente esta base de datos no es tidy, ya que deberíamos tener una columna para la variable día, otra para especie y por último una para la abundancia de cada especie en cadad día. Antes de mostrar como realizaríamos esto con `gather`, veamos sus efectos para entenderlo mejor. La forma más básica de usar esta función sería el solo darle un nombre a la columna *key* (que tendrá el nombre de las columnas) y otro a *value*, que tendría el valor de las celdas. Veamos que ocurre si hacermos eso en el siguiente código y tabla 3.4.

```
library(tidyverse)
DF_largo <- df_cuentas %>% gather(key = Columnas, value = Valores)
```

Como vemos en la tabla 3.4, en la columna llamada *Columnas*, tenemos sólo los nombres de las columnas de la tabla 3.3, y en la columna *Valores*, tenemos los valores encontrados en la tabla 3.3. Sin embargo, para tener las tres columnas que desearíamos tener (día, especie y abundancia), necesitamos que la variable día no participe de este "*alargamiento*", para esto lo que haríamos sería los siguiente:

```
DF_largo <- df_cuentas %>% gather(key = Columnas, value = Valores,
    -dia)
```

Al agregar `-día` como tercer argumento quitamos esa variable del día en el "*alargamiento*", en ese caso obtenemos la tabla 3.5. Ahora sólo falta arreglar los nombres.

Para cambiar los nombres de las columnas que nos faltan, sólo cambiamos los valores de los argumentos `key` y `value` como se ve a continuación y en la tabla 3.6.

Table 3.5: Abundancia detectada por especie en tres días de muestreo

dia	Columnas	Valores
Lunes	Lobo	2
Martes	Lobo	1
Miercoles	Lobo	3
Lunes	Liebre	20
Martes	Liebre	25
Miercoles	Liebre	30
Lunes	Zorro	4
Martes	Zorro	4
Miercoles	Zorro	4

Table 3.6: Abundancia detectada por especie en tres días de muestreo

dia	Especie	Abundancia
Lunes	Lobo	2
Martes	Lobo	1
Miercoles	Lobo	3
Lunes	Liebre	20
Martes	Liebre	25
Miercoles	Liebre	30
Lunes	Zorro	4
Martes	Zorro	4
Miercoles	Zorro	4

```
DF_largo <- df_cuentas %>% gather(key = Especie, value = Abundancia,
    -dia)
```

3.3.2 spread

`spread` es la función inversa a `gather`, esto es, toma una tabla de datos en formato ancho y la trnasforma en una base de datos de formato largo. Esta función tiene dos argumentos básicos. *key* que es el nombre de la variable que pasará a ser nombres de columna y *value*, que es el nombre de la columna con los valores que llenarán estas columnas.

3.3.2.1 Continuación ejemplo de censos

Volvamos al ejemplo de los censos donde quedamos, en nuestro último ejercicio creamos el data frame *DF_largo* que vemos en la tabla 3.6. Veremos algunos ejemplos de como podemos cambiar este data frame en una tabla ancha:

```
DF_ancho <- DF_largo %>% spread(key = dia, value = Abundancia)
```

Con el código anterior generamos la 3.7, la cuál es distinta a la original en la tabla 3.3), en esta los días quedaron como nombres de columnas, y las especies pasaron a ser una variable.

Table 3.7: Abundancia detectada por especie en tres días de muestreo

Especie	Lunes	Martes	Miercoles
Liebre	20	25	30
Lobo	2	1	3
Zorro	4	4	4

Table 3.8: Todas las opciones a probar para generar una tabla ancha

Key	Value
Especie	dia
Abundancia	dia
dia	Especie
Abundancia	Especie
dia	Abundancia
Especie	Abundancia

En la tabla 3.8 se ven todas las opciones de como generar una tabla ancha en base a el data frame *DF_largo*, pruebe opciones hasta entender la función, algunas de estas opciones darán errores.

3.4 Ejercicios

3.4.1 Ejercicio 1

Utilizando el siguiente código usando el paquete dismo bajaras la base de datos del *GBIF* (Global Biodiversity Information Facility) de presencias conocidas del huemul (*Hippocamelus bisulcus*):

```
library(dismo)
Huemul <- gbif("Hippocamelus", "bisulcus", down = TRUE)
colnames(Huemul)
```

Tomando la base de datos generada:

a. Quedarse con solo las observaciones que tienen coordenadas geograficas

b. Determinar cuantas observaciones son de observacion humana y cuantas de especimen de museo

3.4.2 Ejercicio 2

Entrar a INE ambiental y bajar la base de datos de Dimensión Aire.

a. Generar una base de datos **tidy** con las siguientes 5 columnas

- El nombre de la localidad donde se encuntra la estación
- El año en que se tomo la medida
- El mes en que se tomo la medida
- La temperatura media de ese mes

- La media del mp25 de ese mes
- Humedad relativa media mensual

b. De la base de datos anterior obterner un segundo data frame en la cual calculen para cada variable y estación la media y desviación estandar para cada mes

Chapter 4

Visualización de datos

4.1 Paquetes necesarios para este capítulo

Para este capítulo necesitas tener instalado el paquete *tidyverse*.

En este capítulo se explicará qué es el paquete *ggplot2* (Wickham, 2016) y cómo utilizarlo para visualizar datos.

Dado que este libro es un apoyo para el curso BIO4022, esta clase puede también ser seguida en este link. El video de la clase se encontrará disponible en este link

4.2 El esqueleto

El esqueleto de una visualización usando *ggplot2* es la siguiente

```
ggplot(data.frame, aes(nombres_de_columna)) + geom_algo(argumentos,
    aes(columnas)) + theme_algo()
```

Como ejemplo para discutir usaremos el siguiente código que genera la figura 4.1:

```
library(tidyverse)
data("diamonds")
ggplot(diamonds, aes(x = carat, y=price)) + geom_point(aes(color = cut)) + theme_classic()
```

En este caso general, lo primero que ponemos después de ggplot es el data.frame desde el cual graficaremos algo. En el ejemplo de la figura 4.1 usamos la base de datos *diamonds* del paquete *ggplot2* (Wickham, 2016), luego dentro de `aes` ponemos las columnas que graficaremos como x y/o y. En nuestro ejemplo dentro de aes ponemos como eje x los quilates de los diamantes (caret) y como y el precio de los mismos (price). Ojo que existe la necesidad de poner `aes` en ggplot2 (algo que no había sido necesario cuando usamos *dplyr* o *tidyr*).

4.3 Por que usamos aes() y +

Al ser el primer paquete creado en el tidyverse, *ggplot2* tiene un par de convenciones distintas. Por un lado, cada vez que usamos el nombre de una columna que está en un data frame debemos usarlo dentro de la función `aes`. Además, cuando se creó el paquete *ggplot2* no existia el pipeline (`%>%`), por lo que se utilizaba el signo `+` con la misma función.

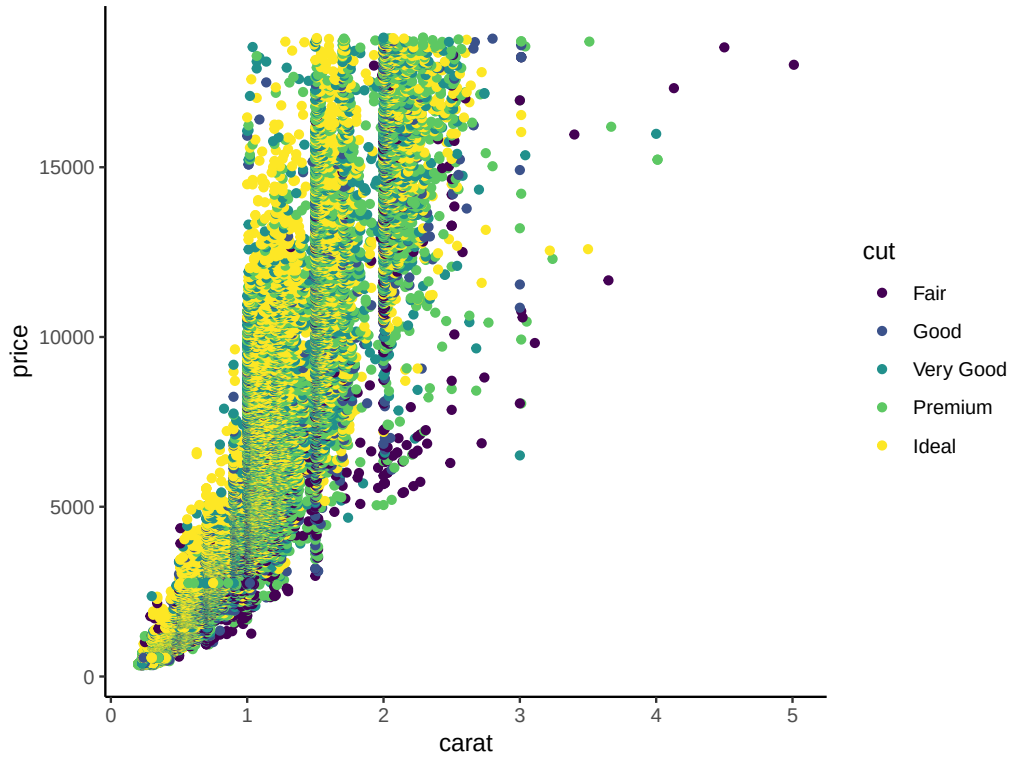

Figure 4.1: Gráfico en el cual gráficamos los quilates de diamantes versus su precio, con el corte del diamante representado por el color

4.4 geom_algo

Luego de especificar una base de datos, se debe continuar con un `geom_algo`, esto nos indicará que tipo de gráfico usaremos. Los gráficos pueden ser combinados como veremos en ejemplos futuros.

4.4.1 Una variable categórica una continua

Primero veremos algunos de los *geom* que podemos utilizar con una variable categórica y una continua

4.4.1.1 geom_boxplot

En la figura 4.2, generado a partir del código a continuación con la base de datos iris presente en `R` (Anderson, 1935).

```
data("iris")
ggplot(iris, aes(x = Species, y = Sepal.Length)) + geom_boxplot()
```

Los boxplots muestran una línea gruesa central (la mediana), una caja, que delimita el primer y tercer cuartil y los bigotes, los cuales se extienden hasta los valores extremos. En el caso que estos valores estén por sobre 1.5 veces la distancia entre el primer y tercer cuartil, estos serán representados por puntos (siendo considerados outlyers). En la figura 4.2, sólo *Iris virginica* presenta un outlayer en cuanto a las medidas del largo del sépalo.

Los boxplots, como todos los gráficos pueden ser personalizados usando otros argumentos, los que mostraremos en esta sección los iremos introduciendo de a poco. Si quisieramos por ejemplo que el color de

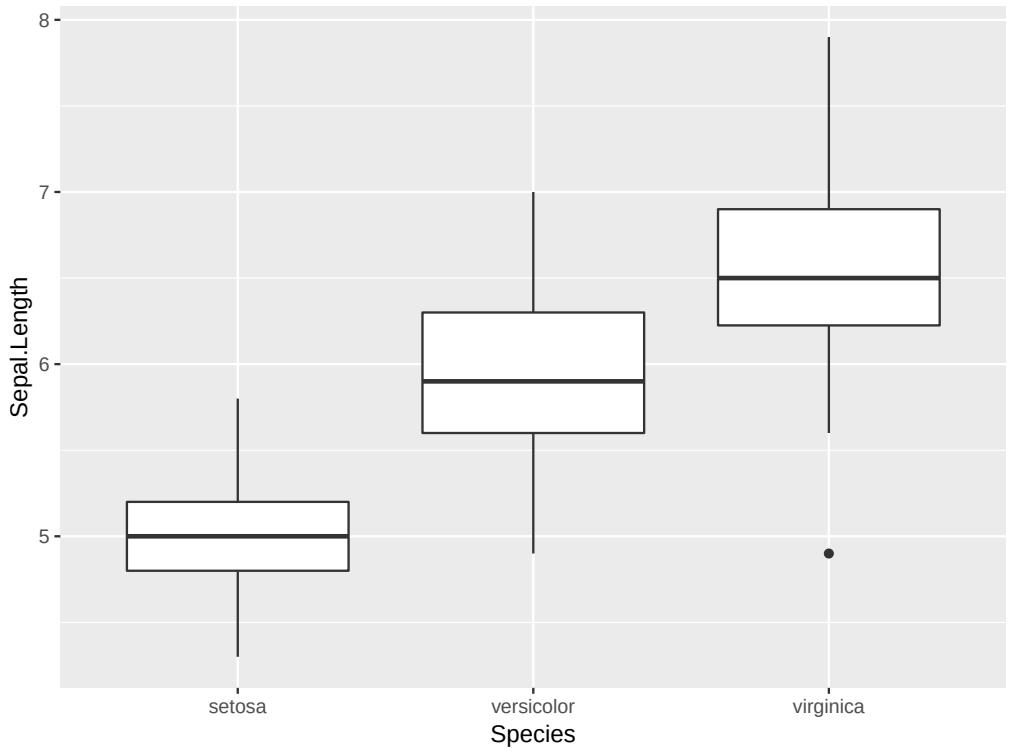

Figure 4.2: Boxplot que representa los largos del sépalo de tres especies del género Iris

las cajas del *boxplot* fueran dea cuerdo a la especie, cambiamos el llenado (**fill**) de la caja, como vemos en el siguiente ejemplo y figura 4.3

```
ggplot(iris, aes(x = Species, y = Sepal.Length)) + geom_boxplot(aes(fill = Species))
```

Dos cosas a notar en este ejemplo, por un lado la leyenda se genera de forma automática, y por otro lado, vemos que es necesario poner *Species* dentro de `aes`, esto es debido a que Species es una columna y como se explicó al principio de este capítulo, todas las columnas deben ser incuidas dentro de la función `aes` para poder ser referenciadas.

4.4.1.2 geom_jitter

Utilizando la misma base de datos, podemos crear un gráfico del tipo *jitter*. En este caso hay un punto por cada observación, lo cual puede ayudar a entender mejor los datos que tenemos.

```
ggplot(iris, aes(x = Species, y = Sepal.Length)) + geom_jitter(aes(color = Species))
```

En la figura 4.4 vemos los mismos datos que en la figura 4.2, el agregar el `color = Species` dentro del `aes` nos permite que el color de cada punto este determinado por la especie a la que pertenece.

4.4.1.3 Otros geom categóricos

Otros geom categóricos que podemos explorar con esta base de datos son:

- geom_violin

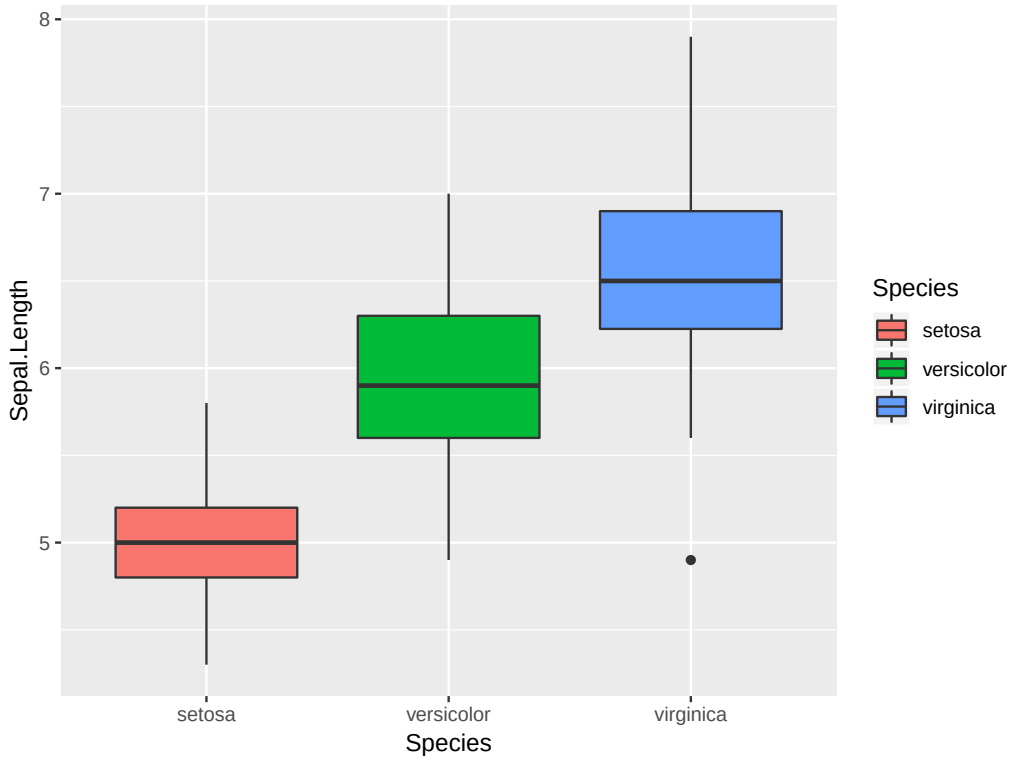

Figure 4.3: Boxplot que representa los largos del sépalo de tres especies del género Iris, en este caso el color de la caja representa la especie

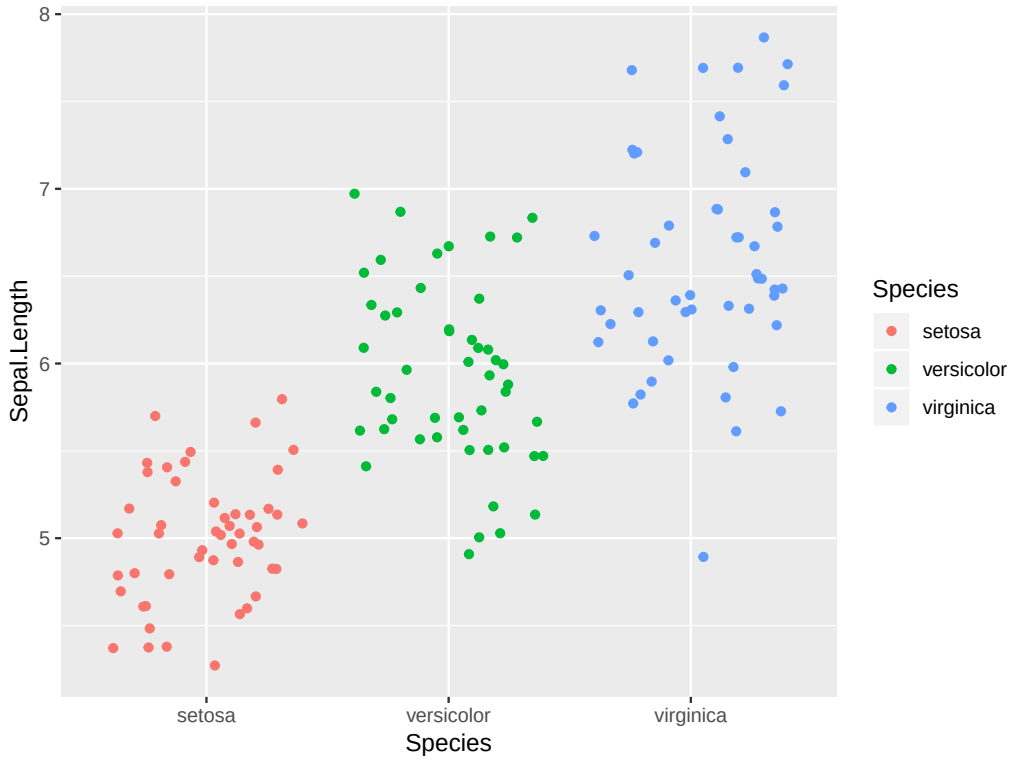

Figure 4.4: jitter plot que representa los largos del sépalo de tres especies del género Iris, en este caso el color de los puntos representan la especie

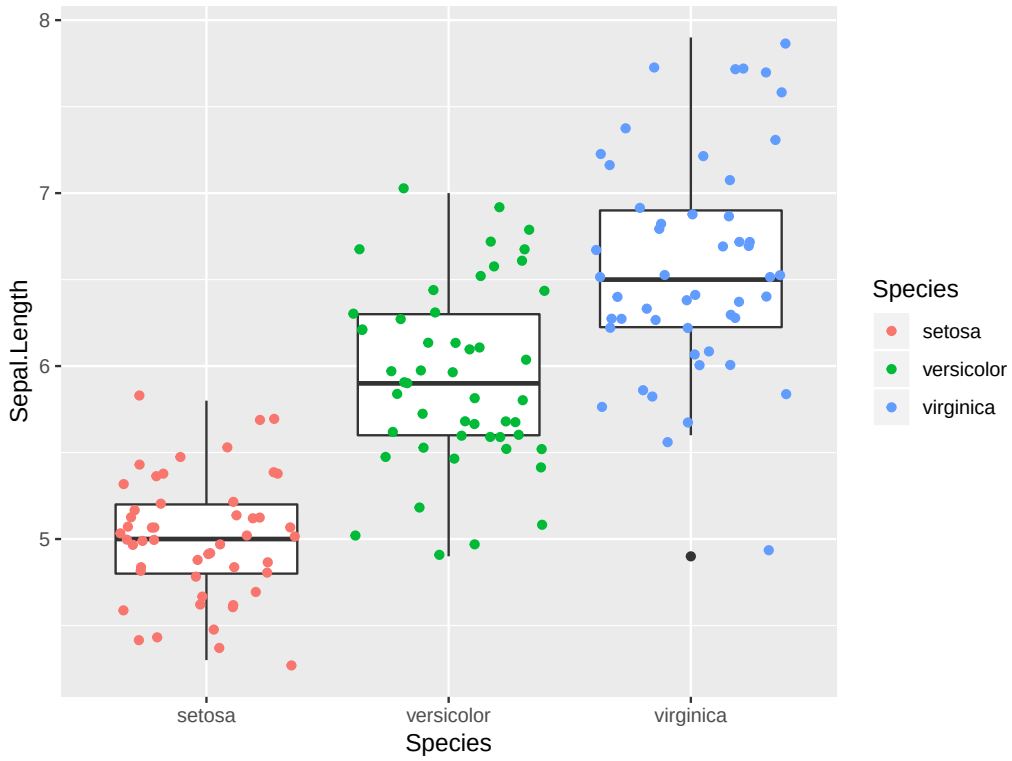

Figure 4.5: Boxplot y jitter plot combinados que representa los largos del sépalo de tres especies del género Iris.

- geom_bar
- geom_col

4.5 Combinando geoms

Uno puede escribir varios geoms para formar un gráfico combinado. Por ejemplo, podríamos generar un gráfico con un boxplot y un jitter plot, como vemos en la figura 4.5

```
ggplot(iris, aes(x = Species, y = Sepal.Length)) + geom_boxplot() +
    geom_jitter(aes(color = Species))
```

4.5.1 El orden importa

Si bien se pueden combinar los geom, el orden de estos importa, ya que *ggplot2* genera las figuras por capas. Esto es ilustrado en la figura 4.6, en la cual al crear primero el jitter y luego el boxplot, las cajas del boxplot tapan los puntos, a diferencia de la figura 4.5

```
ggplot(iris, aes(x = Species, y = Sepal.Length)) + geom_jitter(aes(color = Species)) +
    geom_boxplot()
```

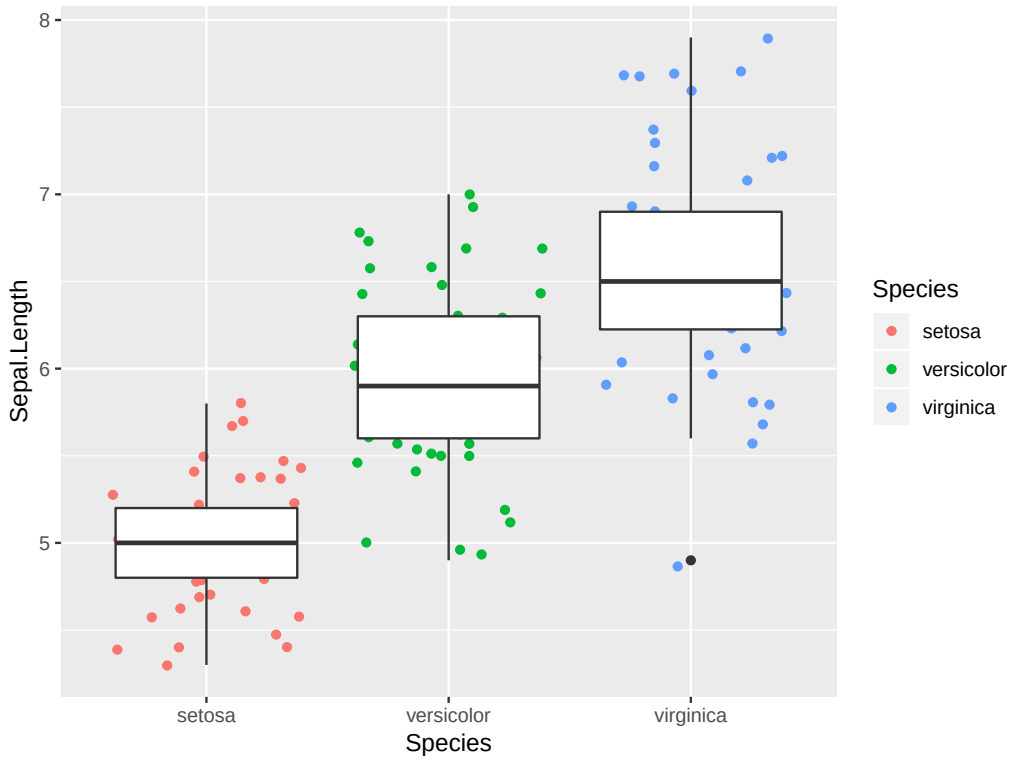

Figure 4.6: Boxplot y jitter plot combinados que representa los largos del sépalo de tres especies del género Iris, en este caso al llamar al jitter antes del boxplot, las cajas tapan los puntos.

4.5.2 Dos variables continuas

Algunos de los geoms que podemos usar para dos variables continuas son:

- geom_point
- geom_smooth
- geom_line
- geom_hex
- geom_rug

Ahora veremos algunos de ellos:

4.5.2.1 geom_point

Este geom es el que nos permite hacer un gráfico de dispersión en R. Para esto tenemos que poner variables continuas en x e y en ggplot y agregar la función `geom_point`, como vemos en el siguiente código y en la figura 4.7.

```
data("ChickWeight")
ggplot(ChickWeight, aes(x = Time, y = weight)) + geom_point()
```

Si quisieramos que el color de cada punto estuviera separado por dieta, podríamos agregarle `aes(color = Diet)` a geom_point. Sin embargo, deberíamos transformar Diet en factor, ya sea antes de usar ggplot o dentro de ggplot tal como vemos en el siguiente código y en la figura 4.8.

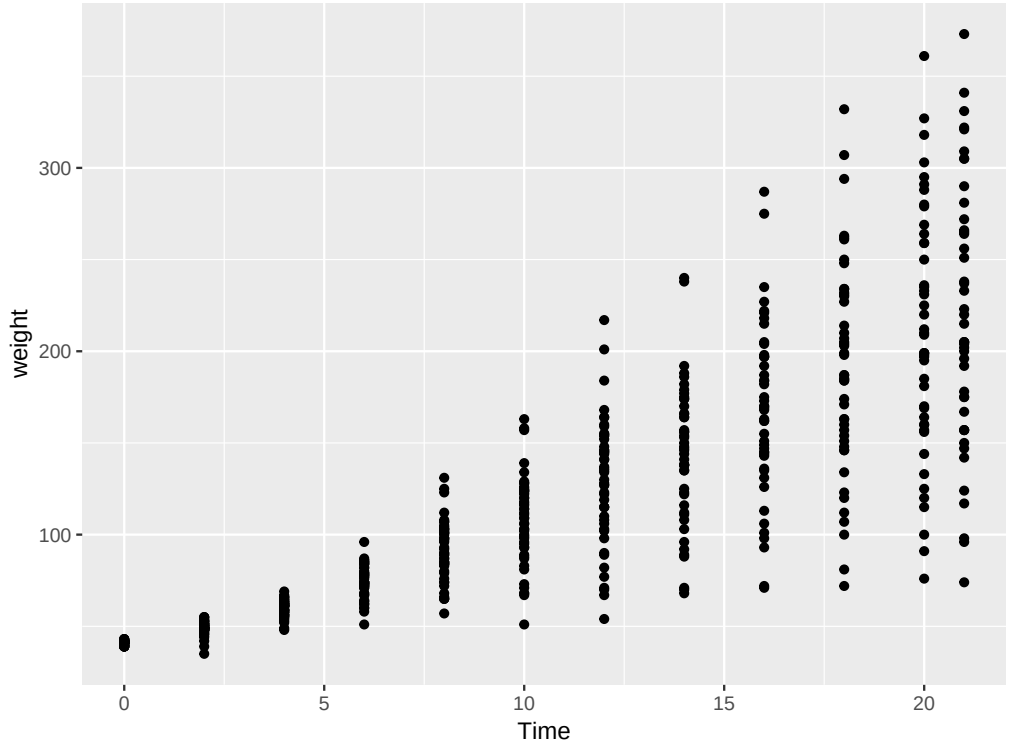

Figure 4.7: Gráfico en el cual vemos el peso de pollos en el tiempo

```
data("ChickWeight")
ggplot(ChickWeight, aes(x = Time, y = weight)) + geom_point(aes(color = factor(Diet)))
```

4.5.2.2 geom_smooth y stat_smooth

4.5.2.2.1 geom_smooth

Estas funciones nos permiten generar líneas de tendencias con intervalos de confianza. Así si quisieramos ver líneas de tendencias para nuestro scatterplot, dependiendo de la dieta, usaríamos el siguiente código, el cual nos da la figura 4.9.

```
ggplot(ChickWeight, aes(x = Time, y = weight)) + geom_point(aes(color = factor(Diet))) +
    geom_smooth(aes(fill = factor(Diet)))
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

Por defecto, la función `geom_smooth` generará una tendencia basada en *loess*, lo cual es una correlación local. En general, es mejor hacer una línea de tendencia basado en modelos que uno puede explicar mejor como un modelo lineal. Para esto, cambiamos el argumento method a lm como en el siguiente código y la figura 4.10.

```
ggplot(ChickWeight, aes(x = Time, y = weight)) + geom_point(aes(color = factor(Diet))) +
    geom_smooth(aes(fill = factor(Diet)), method = "lm")
```

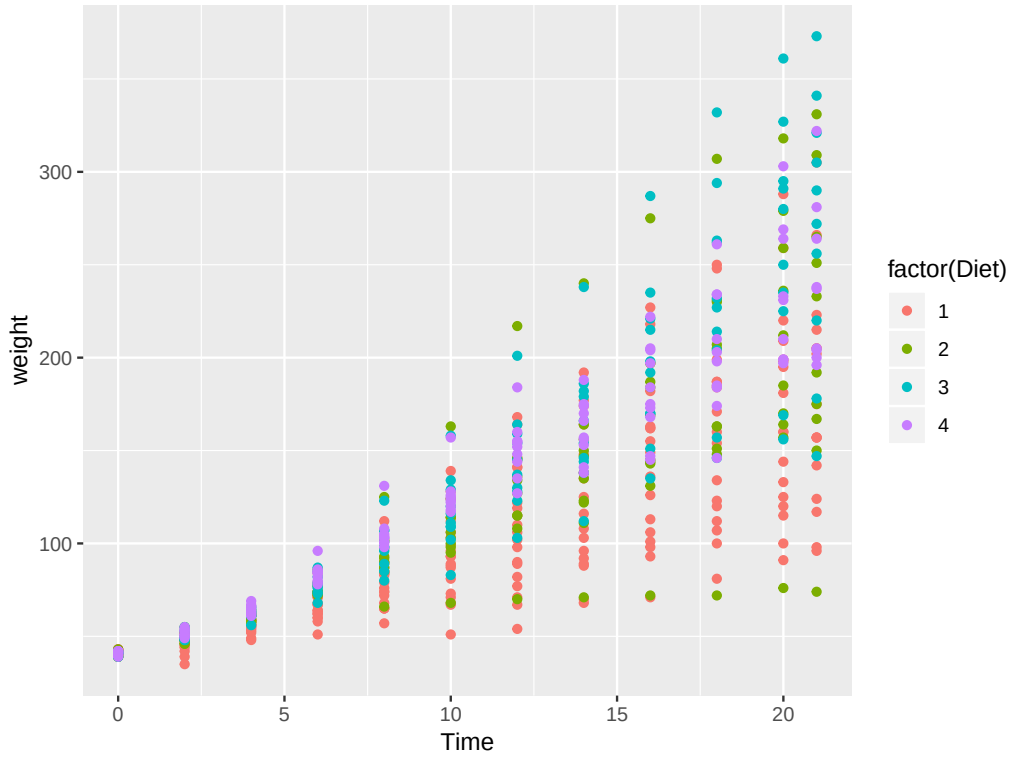

Figure 4.8: Gráfico en el cual vemos el peso de pollos en el tiempo, con colores distintos según el tipo de dieta

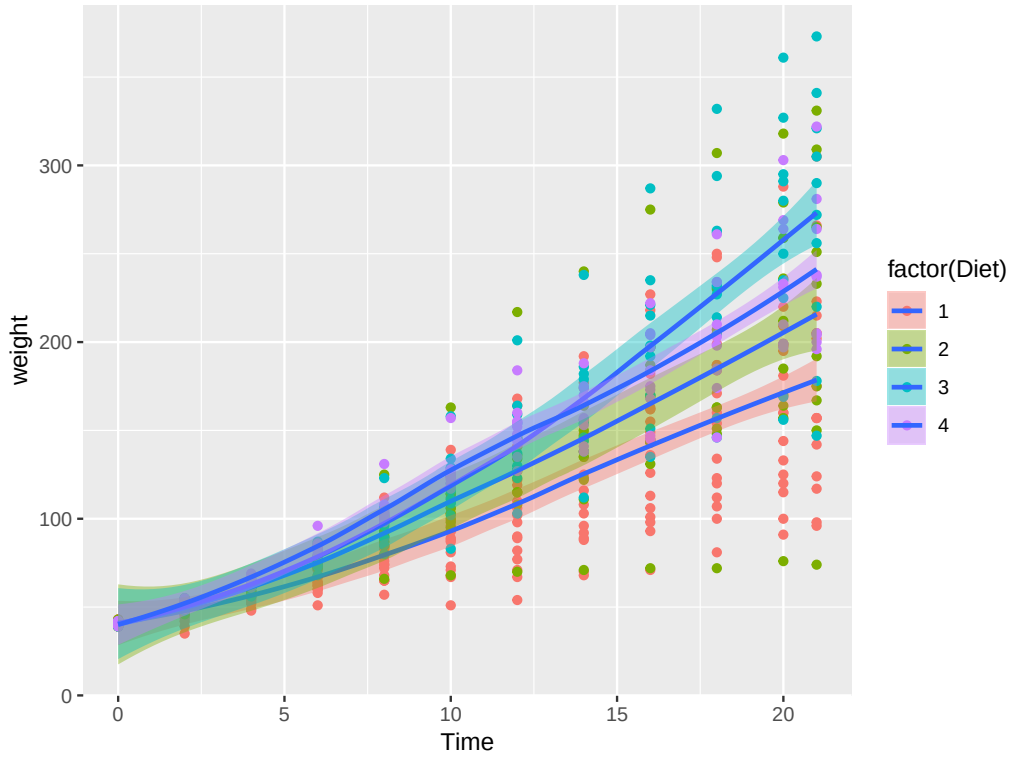

Figure 4.9: Gráfico en el cual vemos el peso de pollos en el tiempo, con colores distintos según el tipo de dieta, con líneas de tendencia e intervalos de confianza basados en el método loess

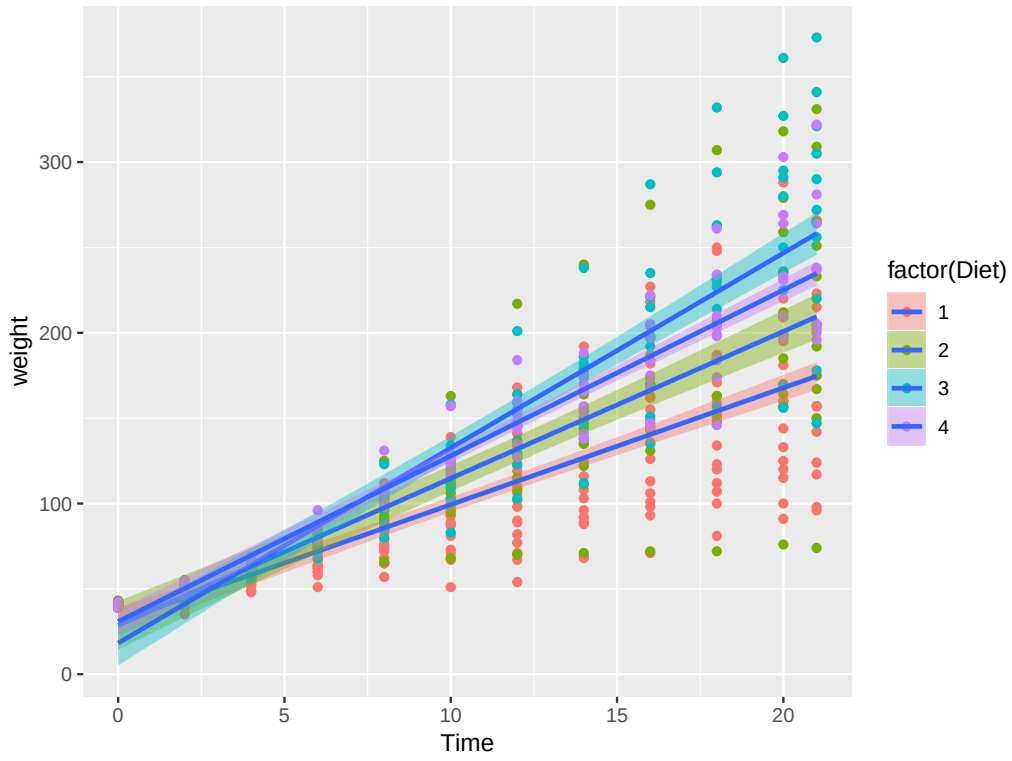

Figure 4.10: Gráfico en el cual vemos el peso de pollos en el tiempo, con colores distintos según el tipo de dieta, con líneas de tendencia e intervalos de confianza basados en modelos lineales

4.5.2.2.2 stat_smooth

La función `stat_smooth` es más flexible que `geom_smooth`. La gran diferencia es que nos permite incluir una fórmula para expresar la relación entre x e y. Por ejemplo, si pensaramos que en el caso de la base de datos `ChickWeight` la relación entre el peso y el tiempo se expresa mejor con una ecuación cuadrática (ver ecuación (4.1)) tendríamos el siguiente código que genera la figura 4.11.

```
ggplot(ChickWeight, aes(x = Time, y = weight)) + geom_point(aes(color = factor(Diet))) +
    stat_smooth(aes(fill = factor(Diet)), method = "lm", formula = y ~
        x + I(x^2))
```

$$y = \beta_2 x^2 + \beta_1 x + c \tag{4.1}$$

4.5.3 Combinando varios gráficos con facet_wrap

Algunas veces, en particular si tenemos muchas variables categóricas, no es recomendable generar una línea o punto de color distinto para cada variable. Por ejemplo, si seguimos con el crecimiento de los pollos de la base de datos `ChickWeight`, vemos que la variable *Chick* representa cada pollo. Dado que hay varios pollos por dieta se vuelve confuso y poco informativo como se ve en la figura 4.12 generada con el siguiente código.

```
ggplot(ChickWeight, aes(x = Time, y = weight)) + geom_point(aes(color = Diet)) +
    geom_line(aes(color = Diet, group = Chick))
```

Para aclarar este enredo, es mejor el generar un gráfico para cada dieta, y es ahí donde entra la función `facet_wrap`. Esta función nos permite generar el gráfico deseado al agregar como argumento dentro de la

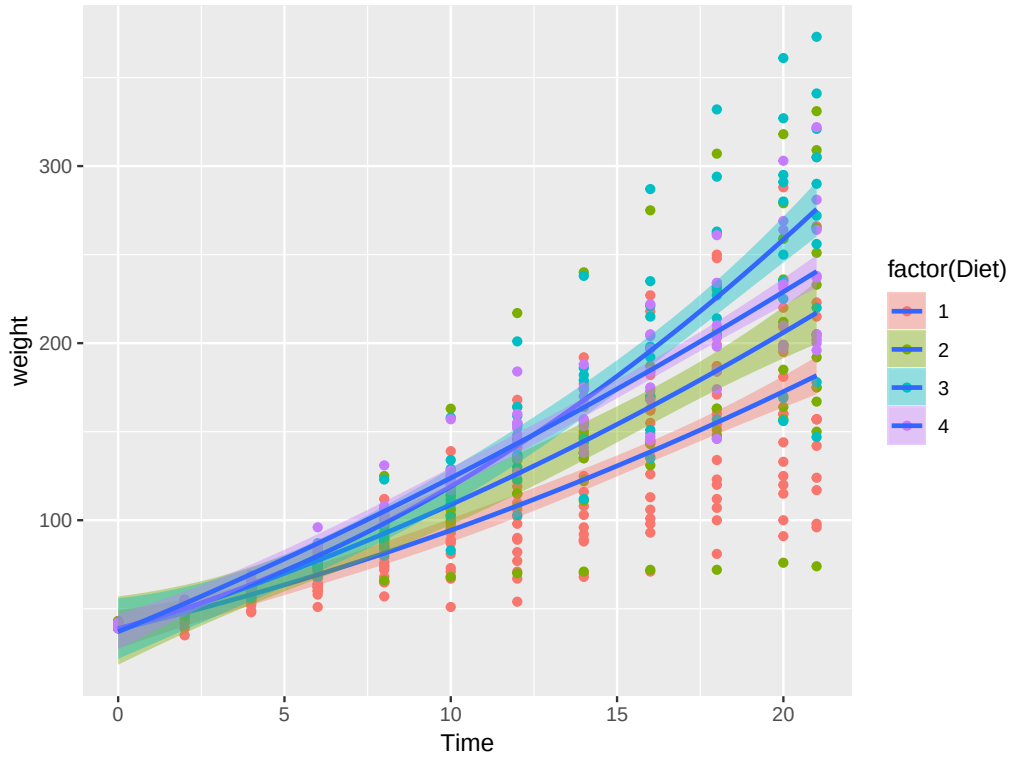

Figure 4.11: Gráfico en el cual vemos el peso de pollos en el tiempo, con colores distintos según el tipo de dieta, con líneas de tendencia e intervalos de confianza basados en modelos lineales con una relación cuadrática

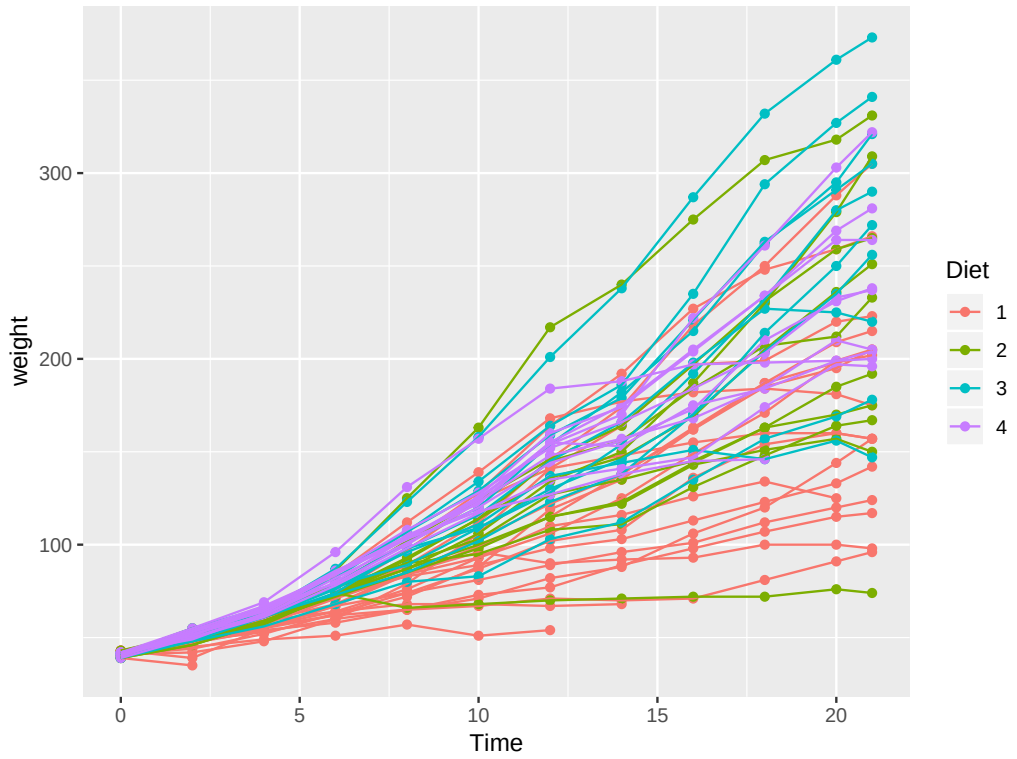

Figure 4.12: Gráfico en el cual vemos el peso de pollos en el tiempo, con colores distintos según el tipo de dieta y con líneas para cada pollo individual.

función el simbolo ~ seguido del nombre de la variable a utilizar para separar los gráficos, tal como en la figura 4.13 y su código correspondiente.

```
ggplot(ChickWeight, aes(x = Time, y = weight)) + geom_point(aes(color = Diet)) +
    geom_line(aes(color = Diet, group = Chick)) + facet_wrap(~Diet)
```

Esta función siempre tendrá los mismos ejes y escala para todos los gráficos. Además, intentará siempre dejar la disposición de los gráficos de la forma más cuadrada posible, pero esto puede ser modificado agregando el argumento *ncol* y un número de columnas, así como vemos en la figura 4.14 y su código correspondiente.

```
ggplot(ChickWeight, aes(x = Time, y = weight)) + geom_point(aes(color = Diet)) +
    geom_line(aes(color = Diet, group = Chick)) + facet_wrap(~Diet,
    ncol = 3)
```

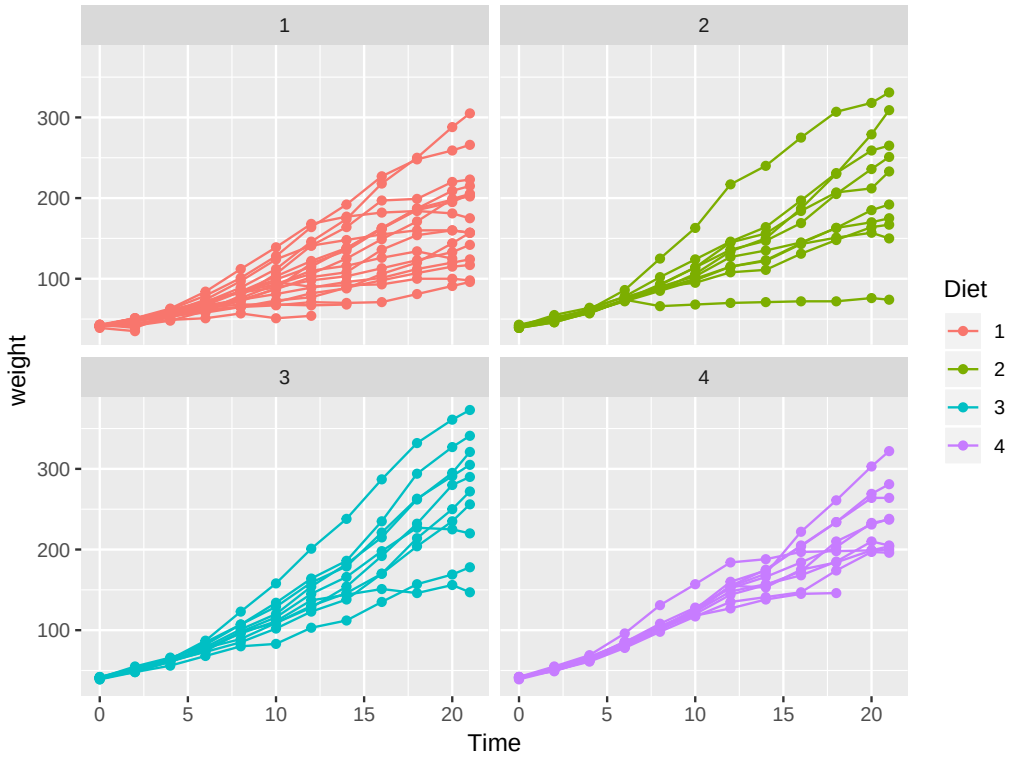

Figure 4.13: Gráfico en el cual vemos el peso de pollos en el tiempo, con colores y gráficos distintos según el tipo de dieta y con líneas para cada pollo individual.

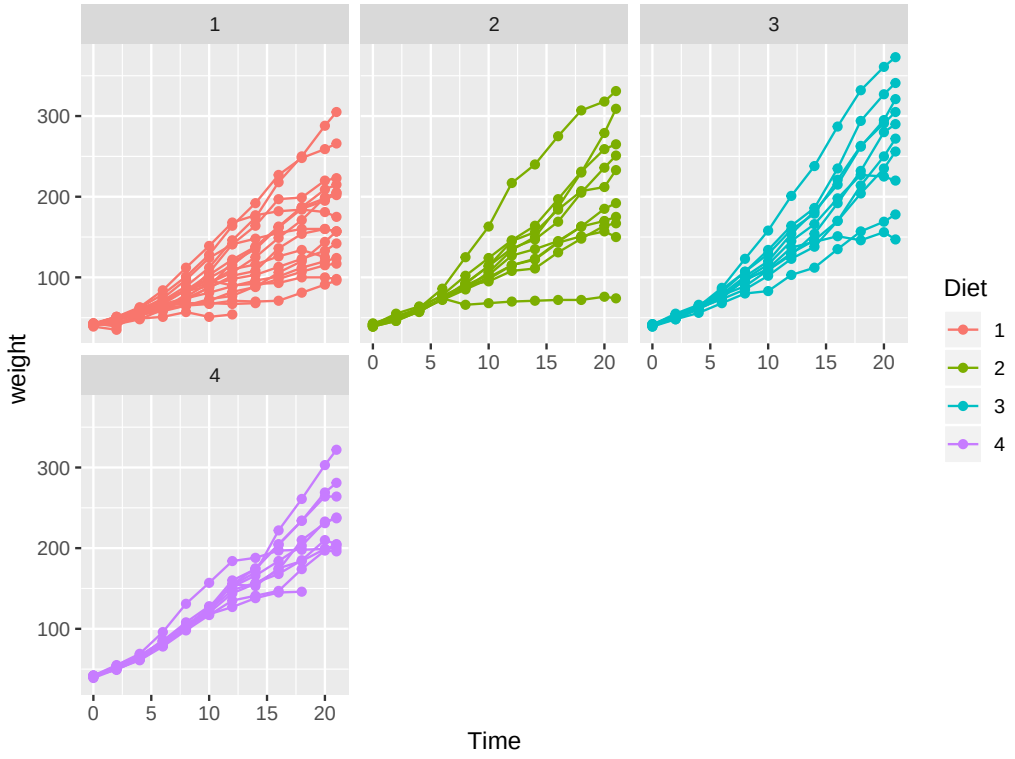

Figure 4.14: Gráfico en el cual vemos el peso de pollos en el tiempo, con colores y gráficos distintos según el tipo de dieta y con líneas para cada pollo individual.

Chapter 5

Modelos en R

We have finished a nice book.

Chapter 6

Loops (purrr) y bibliografía (rticles)

Chapter 7

Presentaciones en R

Chapter 8

Soluciones a problemas

Todos los problemas en programación tienen más de una forma de llegar a ellos, es por esto que las soluciones acá mostradas deben tomarse solo como una referencia, y revisar si el resultado final de tu código (aunque sea distinto de este), sea igual al que presentamos.

8.1 Capítulo 1

8.1.1 Ejercicio 1

Algunas posibles soluciones:

```
storms %>% filter(status == "hurricane") %>% select(year, wind,
    hu_diameter) %>% group_by(year) %>% summarize_all(mean)
```

```
storms %>% filter(status == "hurricane" & !is.na(hu_diameter)) %>%
    select(year, wind, hu_diameter) %>% group_by(year) %>% summarize_all(mean)
```

```
storms %>% filter(status == "hurricane") %>% select(year, wind,
    hu_diameter) %>% group_by(year) %>% summarize_all(funs(mean),
    na.rm = TRUE)
```

8.1.2 Ejercicio 2

Una de las soluciones posibles:

```
Solution <- mpg %>% filter(year > 2004 & class == "compact") %>%
    mutate(kpl = (cty * 1.609)/3.78541)
```

8.2 Capítulo 2

8.2.1 Ejercicio 1

Una posible solución a este problema sería:

```
`r mean((iris %>% filter(Species == "virginica"))$Petal.Length)`
```

8.3 Capítulo 3

8.3.1 Ejercicio 1

8.3.1.1 a

```
Sola <- Huemul %>% dplyr::select(lon, lat, basisOfRecord) %>%
    filter(!is.na(lat) & !is.na(lon))
```

lon	lat	basisOfRecord
-72.93940	-49.37483	HUMAN_OBSERVATION
-72.97712	-51.01511	HUMAN_OBSERVATION
-71.87026	-46.08686	HUMAN_OBSERVATION
-72.43751	-47.20485	HUMAN_OBSERVATION
-73.01456	-51.03635	HUMAN_OBSERVATION
-73.03190	-51.17531	HUMAN_OBSERVATION
-72.72944	-46.25602	HUMAN_OBSERVATION
-71.31538	-41.30110	PRESERVED_SPECIMEN
-71.31538	-41.30110	PRESERVED_SPECIMEN
-71.71667	-44.86667	PRESERVED_SPECIMEN
-71.71667	-44.86667	PRESERVED_SPECIMEN
-71.30989	-40.81978	PRESERVED_SPECIMEN
-71.31538	-41.30110	PRESERVED_SPECIMEN
-73.02467	-50.46476	PRESERVED_SPECIMEN
-71.33186	-41.26523	PRESERVED_SPECIMEN
-73.01764	-50.46747	PRESERVED_SPECIMEN
-71.70000	-45.26667	PRESERVED_SPECIMEN
-71.70000	-45.26667	PRESERVED_SPECIMEN
-71.70000	-45.26667	PRESERVED_SPECIMEN
-72.08000	-47.25000	PRESERVED_SPECIMEN
-72.00000	-41.50000	PRESERVED_SPECIMEN
-71.36714	-41.13574	PRESERVED_SPECIMEN
-71.71094	-42.75692	HUMAN_OBSERVATION
-71.64718	-40.22605	PRESERVED_SPECIMEN
-67.88534	-43.99376	PRESERVED_SPECIMEN

8.3.1.2 b

```
Solb <- Huemul %>% group_by(basisOfRecord) %>% summarize(N = n())
```

basisOfRecord	N
HUMAN_OBSERVATION	102
PRESERVED_SPECIMEN	65

8.3.2 Ejercicio 2

8.3.2.1 a

Primero bajamos la base de datos, lo cual se puede hacer de forma manual o como en el código siguiente utilizando la función `download.file`

```
download.file("http://www.ine.cl/docs/default-source/medioambiente-(micrositio)/variables-b%C3%A1sicas-a
    destfile = "test.xlsx")
```

Una vez bajada esta base de datos utilizaremos los paquetes *readxl* para leer los archivos excel, *tidyverse* para manipular los datos y *stringr* para trabajar con texto.

```
library(readxl)
library(tidyverse)
library(stringr)
```

Partimos leyendo la pestaña que contiene las estaciones meteorológicas con su código:

```
EM <- read_excel("test.xlsx", sheet = "T001")
```

Luego para poder más adelante unir esta base de datos con otras, cambiamos el nombre de la columna *Codigo_Est_Meteoro* a *Est_Meteoro* como aparece en las otras bases de datos.

```
EM <- EM %>% rename(Est_Meteoro = Codigo_Est_Meteoro)
```

Luego empezamos a trabajar con la base de datos de temperatura media, para esto leemos la pestaña *E10000003*

```
TempMedia <- read_excel("test.xlsx", sheet = "E10000003")
```

Existen varias variables que no utilizaremos, por ejemplo el código de la variable, y la unidad de medida. Además vemos que la variable día, siempre tiene valor 0, por lo cuál podemos eliminarla también.

```
TempMedia <- TempMedia %>% select(-Codigo_variable, -Unidad_medida,
    -Día)
```

Además podemos cambiar los nombres de la columna *ValorF* que no tiene ningún significado a *TempMedia* y *Año* a *Year*, esta última variable es cambiada solo por que la letra Ñ puede no ser leída por todos los computadores.

```
TempMedia <- TempMedia %>% rename(TempMedia = ValorF, Year = Año)
```

Si nos fijamos, hay algunos años, en los cuales todos los meses aparecen como 13, esto nos indica que en estos años no se registró en que mes se realizó la medición, por lo cual se eliminarán esas obsevaciones.

```
TempMedia <- TempMedia %>% filter(Mes != 13)
```

Posterior a esto, unumos la base de datos *TempMedia* con la base de datos *EM* y seleccionamos tan solo las columnas que nos interesan y finalmente transformamos el mes en una variable numérica:

```
## Joining, by = "Est_Meteoro"
```

```
TempMedia <- left_join(TempMedia, EM) %>% select(Mes, Year, TempMedia,
    Ciudad_localidad) %>% mutate(Mes = as.numeric(Mes))
```

Si hicieramos todo esto en un comando tendriamos el siguiente código

Table 8.1: Las primeras 20 observaciones de temperatura y humedad unidas

Mes	Year	TempMedia	Ciudad_localidad	HumMedia
1	1981	22.0	Arica	NA
2	1981	22.2	Arica	NA
3	1981	22.1	Arica	NA
4	1981	20.3	Arica	NA
5	1981	18.2	Arica	NA
6	1981	17.0	Arica	NA
7	1981	15.0	Arica	NA
8	1981	16.0	Arica	NA
9	1981	16.6	Arica	NA
10	1981	15.9	Arica	NA
11	1981	19.1	Arica	NA
12	1981	21.1	Arica	NA
1	1981	19.7	Iquique	NA
2	1981	21.1	Iquique	NA
3	1981	20.9	Iquique	NA
4	1981	19.9	Iquique	NA
5	1981	17.6	Iquique	NA
6	1981	15.9	Iquique	NA
7	1981	14.6	Iquique	NA
8	1981	15.6	Iquique	NA

```
TempMedia <- read_excel("test.xlsx", sheet = "E10000003") %>%
    select(-Codigo_variable, -Unidad_medida, -Día) %>% rename(TempMedia = ValorF,
    Year = Año) %>% filter(Mes != 13) %>% left_join(EM) %>%
    select(Mes, Year, TempMedia, Ciudad_localidad) %>% mutate(Mes = as.numeric(Mes))
```

De la misma manera modificamos el código de arriba para la humedad con la salvedad que la columna de día no tiene tilde en esta pestaña a la fecha de 19 de Agosto del 2018:

```
## Joining, by = "Est_Meteoro"
```

```
HumMedia <- read_excel("test.xlsx", sheet = "E10000006") %>%
    dplyr::select(-Codigo_variable, -Unidad_medida, -Dia) %>%
    rename(HumMedia = ValorF, Year = Año) %>% filter(Mes !=
    13) %>% left_join(EM) %>% dplyr::select(Mes, Year, HumMedia,
    Ciudad_localidad) %>% mutate(Mes = as.numeric(Mes))
```

En el siguiente código unimos las dos bases de datos, vemos las primeras 20 observaciones de la base de datos resultante en la tabla 8.1

```
TempHum <- full_join(TempMedia, HumMedia)
```

```
## Joining, by = c("Mes", "Year", "Ciudad_localidad")
```

Con esto vemos que la humedad media no es medida en los mismos años ni en todos los lugares que se mide la temperatura media, pero como ambas variables nos interesan por igual, la mantenemos de todas maneras con sus valores *NA*

Table 8.2: Las primeras 20 observaciones de temperatura y humedad agrupadas por mes y localidad

Mes	Ciudad_localidad	TempMedia_mean	HumMedia_mean	TempMedia_sd	HumMedia_sd
1	Antártica Chilena	1.388889	87.35000	0.6319031	3.483772
1	Antofagasta	20.125000	69.70000	0.8378118	1.589549
1	Arica	22.375000	62.72500	0.9391105	2.394960
1	Balmaceda	12.358823	56.37500	1.2200640	2.487804
1	Calama	17.000000	25.00000	NA	NA
1	Cerrillos	20.996000	NaN	0.7855359	NaN
1	Chillán	19.747059	63.05000	0.7054916	3.750111
1	Concepción	16.683333	73.75000	0.6222080	6.727308
1	Copiapó	19.604348	NaN	0.7449700	NaN
1	Coyhaique	13.980556	58.27500	1.2537531	1.543535
1	Curicó	20.632353	59.65000	0.7293503	6.310573
1	Graneros	21.480000	NaN	0.4661330	NaN
1	Iquique	21.791667	61.72500	1.0332680	3.981101
1	Isla Juan Fernández	18.513889	71.22500	0.5111068	3.044531
1	La Serena	17.311429	75.97500	0.7275145	3.187868
1	Osorno	15.807407	74.10000	0.9388725	2.265686
1	Pudahuel	20.652778	47.53333	0.7268272	5.852635
1	Puerto Montt	14.451429	78.00000	0.7184016	2.499333
1	Punta Arenas	10.852778	62.55000	0.7443064	3.349129
1	Quinta Normal	21.261111	53.35000	0.5530579	6.507688

8.3.2.2 b

El segundo ejercicio es mucho mas simple, donde solo tenemos que agrupar por ciudad y mes, y usar `summarize_all` para las funciones `mean` y `sd` como vemos en la tabla 8.2

```
TempHumMonthly <- TempHum %>% select(-Year) %>% group_by(Mes,
    Ciudad_localidad) %>% summarize_all(funs(mean, sd), na.rm = TRUE)
```

Bibliography

Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., and Chang, W. (2018). *rmarkdown: Dynamic Documents for R*. R package version 1.10.

Anderson, E. (1935). The irises of the gaspe peninsula. *Bulletin of the American Iris society*, 59:2–5.

Henderson, H. V. and Velleman, P. F. (1981). Building multiple regression models interactively. *Biometrics*, pages 391–411.

Henry, L. and Wickham, H. (2018). *purrr: Functional Programming Tools*. R package version 0.2.5.

Hlavac, M. (2018). *stargazer: Well-Formatted Regression and Summary Statistics Tables*. Central European Labour Studies Institute (CELSI), Bratislava, Slovakia. R package version 5.2.2.

Kross, S., Carchedi, N., Bauer, B., and Grdina, G. (2017). *swirl: Learn R, in R*. R package version 2.4.3.

Leek, J. (2015). The elements of data analytic style. *J. Leek.—Amazon Digital Services, Inc.*

Peng, R. D. (2011). Reproducible research in computational science. *Science*, 334(6060):1226–1227.

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

Wickham, H. (2017). *tidyverse: Easily Install and Load the 'Tidyverse'*. R package version 1.2.1.

Wickham, H. (2018a). *forcats: Tools for Working with Categorical Variables (Factors)*. R package version 0.3.0.

Wickham, H. (2018b). *stringr: Simple, Consistent Wrappers for Common String Operations*. R package version 1.3.1.

Wickham, H. et al. (2014). Tidy data. *Journal of Statistical Software*, 59(10):1–23.

Wickham, H. and Henry, L. (2018). *tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions*. R package version 0.8.1.

Wickham, H., Hester, J., and Francois, R. (2017). *readr: Read Rectangular Text Data*. R package version 1.1.1.

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.