

DocWebsite

JorgeCornejo

2024-10-08

Reconocimiento: Gran parte de los documentos introductorios son una traducción libre desde al material preparado y compartido por el grupo SASAP del NCEAS en su sitio SASAP-Training de en GitHub.

Introducción

La reproducibilidad es fundamental en la ciencia, ya que la ciencia está basada en observaciones empíricas acopladas a modelos explicativos. Si bien la reproducibilidad incluye el ciclo de vida completo de la ciencia y además incluye aspectos como la consistencia metodológica y tratamiento del sesgo, en este curso nos enfocaremos a la **reproducibilidad computacional**, esto es la habilidad para documentar datos, análisis y modelos en forma suficiente para que otros investigadores sean capaces de entender, reproducir y re-ejecutar los pasos computacionales que nos permitieron llegar a los resultados y sus conclusiones.

La crisis de reproducibilidad

@ioannidis_why_2005 destacó la crisis en reproducibilidad de la ciencia cuando escribió que “La mayoría de los descubrimientos son falsos para la mayoría de los diseños de investigación, en la mayoría de las áreas de investigación” (“*Most Research Findings Are False for Most Research Designs and for Most Fields*”). Ioannidis indica la formas en que el proceso de investigación ha inflado los efectos de los tamaños muestrales y los tests de hipótesis que codifican los sesgos existentes. Investigaciones posteriores han confirmado que la reproducibilidad es baja para muchos campos de la ciencia, incluyendo genética [@ioannidis_repeatability_2009], ecología [@fraser_questionable_2018], y psicología [@open_science_collaboration_estimating_2015], entre muchas otras. Por ejemplo, el efecto del tamaño de muestra en la psicología se ha mostrado que decrece significativamente cuando se repiten los experimentos (fig. @ref(fig:effectsize)).

Reconocimientos

Este curso está basado en el desarrollado por el *Arctic Data Center* en *NCEAS* y ha sido modificado y traducido con el apoyo del *Instituto de Fomento Pesquero* por Jorge Cornejo-Donoso.

Antes de comenzar con el material para la investigación reproducible, es necesario que aprendamos sobre algunas herramientas fundamentales, como son RStudio, Git/Github y con posterioridad comenzaremos a trabajar con *RMarkDown*.

Investigación reproducible: ¿De qué estamos hablando?

Objetivos

En este capítulo veremos:

- En qué consiste la reproducibilidad computacional y por qué es útil e importante.
- La importancia del control de versiones para la reproducibilidad computacional.
- Cómo verificar que el ambiente de RStudio está configurado adecuadamente para el análisis.
- Cómo configurar git.

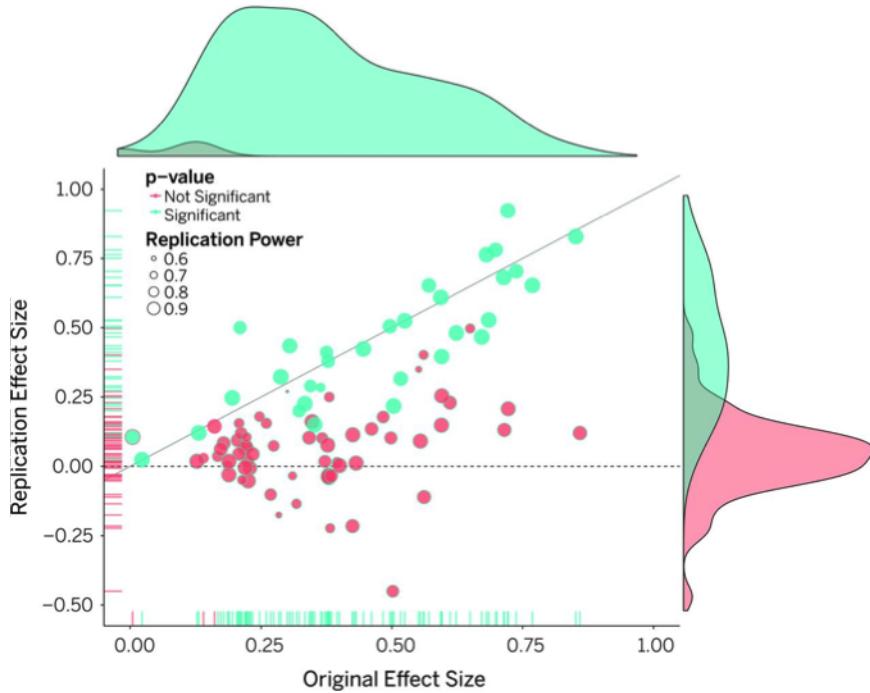


Figure 1: Efecto del tamaño de muestra descrese en experimentos repetidos (Open Sieze Collaboration 2015).

Investigación reproducible

La reproducibilidad es fundamental en la ciencia, que a su vez se basa en observaciones empíricas con las que se construyen o define modelos explicativos.

La reproducibilidad por su parte, envuelve todo el ciclo de vida de la ciencia, incluyendo aspectos como la consistencia metodológica y el tratamiento del sesgo. En este curso nos vamos a enfocar en la **reproducibilidad computacional**, entendiendo esto como la habilidad de documentar datos, análisis y modelos con la suficiente información para que otros investigadores sean capaces de entender e idealmente reproducir, paso a paso, el proceso que no a los resultados y conclusiones de nuestra investigación.

¿Qué es necesario para la reproducibilidad computacional?

El primer paso para abordar este problema es ser capaz de evaluar los datos, análisis y modelos con los que se sacan las conclusiones. Considerando cuales son las prácticas más comunes, esto resulta bastante difícil ya que la generalidad de los datos no se encuentran disponibles, la sección sobre la metodología de los informes y papers no describe en detalle las aproximaciones computacionales utilizadas o los análisis y además los modelos son generalmente realizados en programas gráficos o, cuando se usan ambientes tipo script, el código no se hace disponible.

Sin embargo todos estos inconvenientes pueden ser fácilmente solucionados. Los investigadores pueden lograr la reproducibilidad computacional con aproximaciones basada en las aproximaciones de *ciencia libre* (open science), siguiendo pasos simples para archivar y publicar (libremente) los datos y códigos fuentes utilizados, describiendo el flujo de trabajo y permitiendo de esta forma trazabilidad de los resultados (ej.: @hampton_tao_2015, @munaflo_manifesto_2017).

Conceptualizando los flujos de trabajo (workflows)

El flujo del trabajo científico encapsula todos los pasos desde la adquisición de los datos, limpieza, transformación, integración, análisis y visualización.

Una representación del flujo de trabajo puede variar desde simples diagramas de flujo (fig. @ref(fig:workflow))

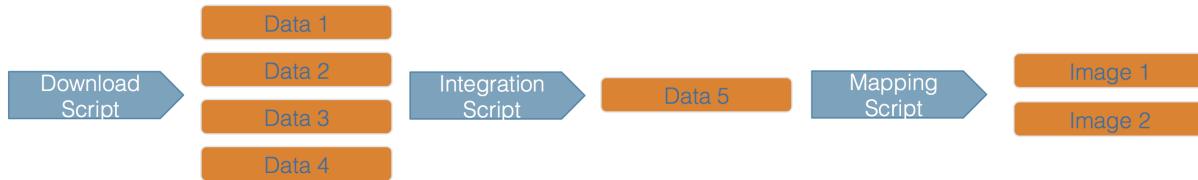


Figure 2: Captura de un flujo de trabajo científico y procedencia de múltiples pasos necesarios para reproducir un resultado científico desde los datos crudos.

a códigos totalmente ejecutables. Códigos de R y python son una forma literal de plasmár el flujo de trabajo lo que cuando son publicados por investigadores, como versiones específicas del código y los datos asociados, permite la repetibilidad de sus cómputos en forma simple y de esta forma entender la procedencia de las conclusiones.

Herramientas: RStudio - Git/GitHub Configuración y Motivación

Por que usar git?

El problema con los nombres de archivos

Cada archivo en un proceso científico sufre de cambios. Los manuscritos son editados. Las figuras son revisadas. Los códigos se corrigen cuando se encuentran problemas. Los archivos de datos se combinan, se corrigen errores, dividen y combinan nuevamente. En el curso de un análisis simple, uno puede esperar miles de cambios en los archivos. Y aún así, todo lo que usamos (al menos la mayoría) para identificar este sinnúmero de cambios son simples **nombres de archivos** (fig. @ref(fig:finalDoc)). Teniendo esto en consideración, es lógico pensar que debe existir una forma mejor de hacer eso... Y si, la hay, se conoce como **Control de Versiones**.

Un sistema de control de versiones ayuda a seguir los cambios que se realizan a nuestros archivos, sin el desastre que resulta de utilizar sólo los nombres de archivos. En los sistemas de control de versiones como **git**, se registra no sólo el nombre del archivo, si no que además su contenido, de esta forma, cuando el contenido cambia se puede identificar que partes cambiaron y donde. El sistema registra además que versión de un archivo viene de una versión previa, teniendo un historial de todos los cambios realizados, así se hace fácil dibujar un gráfico que represente los cambios que ha sufrido un archivo, con todas sus versiones, algo como lo que se muestra en la figura @ref(fig:figVersiones):

Los sistemas de control de versiones asignan un identificador a cada versión de cada archivo y mantiene un registro de como estos están relacionados entre si. Además, estos sistemas permiten ramificaciones en estas versiones y fusiones de regreso a la tronco principal de trabajo. Se puede ademas tener *múltiples copias* en múltiples computadores como respaldos y para trabajo colaborativo. Es posible además incluir etiquetas (tags) a versiones en particular y así poder retornar a una versión en particular de los archivos cuando fueron etiquetados. Por ejemplo, la versión exacta de los datos, código y texto de un manuscrito que fue enviado y que tiene la etiqueta R2 en la figura @ref(fig:figVersiones).

Revisando el ambiente de RStudio

Versión de R

Vamos a usar R version 4.4.1 (2024-06-14), que puede ser descargada e instalada desde CRAN.

Para ver que versión tiene instalada, ejecute el siguiente comando en la consola de RStudio:

```
R.version$version.string
```

"FINAL".doc



FINAL.doc!



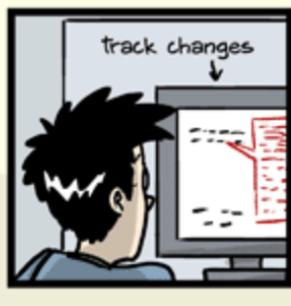
FINAL_rev.2.doc



FINAL_rev.6.COMMENTS.doc



FINAL_rev.8.comments5.
CORRECTIONS.doc



FINAL_rev.18.comments7.
corrections9.MORE.30.doc



FINAL_rev.22.comments49.
corrections.10.#@\$%WHYDID
ICOMETOGRAD SCHOOL????.doc



Figure 3: El dilema de usar nombres como descriptor de versiones.

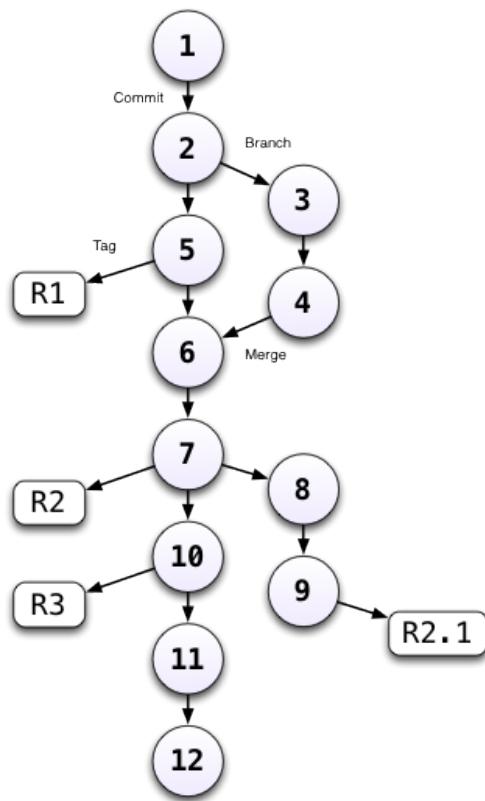


Figure 4: Evolución de las versiones de un archivo.

Versión de RStudio

Vamos a usar la versión de RStudio 1.2.1335 o posterior, la que se puede descargar e instalar desde aquí.

Para verificar que versión de RStudio tiene instalada, ejecute el siguiente comando en la consola de RStudio:

```
RStudio.Version()$version
```

Si el resultado no dice 1.2.1335 (o es una versión anterior), por favor actualice su *RStudio* haciendo clic en “Help -> Check for Updates” y siguiendo las instrucciones.

Instalación de Paquetes de R

Ejecute las siguientes líneas de código para verificar que todos los paquetes necesarios para este curso están instalados en su computador.

```
packages <- c("devtools", "dplyr", "DT", "ggplot2", "leaflet", "roxygen2", "tidyverse")
for (package in packages)
{ if (!(package %in% installed.packages()))
  { install.packages(package) }
}

rm(packages) #Elimina los paquetes desde el entorno de trabajo
```

Si hay algún paquete que no esté instalado, esto va a instalarlo automáticamente. Si ya estuvieran instalados, no verá ninguna salida en o mensaje.

A continuación, cree un nuevo archivo de *R Markdown* (File -> New File -> R Markdown). Si nunca ha creado archivo R Markdown previamente, aparecerá un diálogo preguntando si desea instalar los paquetes requeridos. Haga clic en *Yes*.

Configurando git

Si aun no lo ha hecho, vaya a github.com y cree una cuenta. Si aun no ha bajado e instalado el software de git, descárgelo aquí y proceda a su instalación.

Antes de usar *git*, tiene que decirle quien es usted, para eso se debe establecer las opciones de configuración globales (setting the global options). Lamentablemente, la única forma de hacer esto es por medio de la línea de comando. La versión RStudio que tiene instalada (si es que siguió las instrucciones que le dimos) permite abrir una ventana de terminal en su sesión de RStudio. Para hacer esto seleccione Tools -> Terminal -> New Terminal.

Una pestaña de terminal debería abrirse donde se encuentra normalmente la consola de R. Para establecer las opciones de configuración globales, escriba los siguientes comandos en la pestaña del terminal, reemplazando “*Su Nombre*” por su nombre real y luego presione enter:

```
git config --global user.name "Su Nombre"
```

Luego escriba la siguiente línea de comandos, con la dirección de correo electrónico que usó en la creación de su cuenta en github.com:

```
git config --global user.email "SuEmail@DominioEmail.cl"
```

Por favor note que estas lineas **DEBEN** ser ejecutada una a la vez.

Finalmente, para asegurarse que todo está correcto, escriba este comando que le mostrará las opciones que acaba de definir.

```
git config --global --list
```

Nota para Usuarios de Windows

Si obtienen el mensaje que dice “*comando no encontrado*” o en inglés “*command not found*” (o similar) cuando ejecute estos pasos en la pestaña del terminal de RStudio, puede ser que necesite definir el tipo de terminal que abre RStudio. Bajo algunas instalaciones de git, puede darse que el ejecutable de git no funcione en el terminal que se abre por defecto.

RMarkdown

Objetivos

En este capítulo veremos:

- RMarkdown.
- Introducción a las funciones básicas incluidas en RStudio
- Como usar las páginas de ayuda.

Introducción

Ahora que ya conoce un poco de la sintaxis básica de R, podemos comenzar a aprender sobre RMarkdown. La experiencia nos dice que te vas a volver loco haciendo los análisis o modelos en los cuales trabajes y vas a fallar en generar un flujo de trabajo reproducible. Peor aún, si tratas de desarrollar código directamente en la consola de R. RMarkdown es realmente clave para la investigación colaborativa, es por esto que vamos a comenzar rápidamente a aprender como usarlo y lo seguiremos usando durante el resto del curso.

Un archivo de RMarkdown nos va a ayudar a *tejer* un texto *markdown* con fragmentos de código de R que es evaluado y cuyas salidas, como tablas y gráficos, son incluidas en el documento final.

Para abrir un nuevo documento de RMarkdown siga los siguientes pasos:

File -> New File -> RMarkdown... -> Document of output format HTML, OK.

Puede dar un título como “Mi Proyecto”. Luego clic en OK.

OK, primero que todo: Al abrir el archivo, vamos a ver un 4º panel de la consola de RStudio, este panel es un editor de texto que nos permite abrir varios archivos en un único panel.

Ahora démosle una mirada a este archivo que acabamos de crear - a primera vista se puede ver que no está en blanco; hay un texto inicial que se entrega por defecto. Hay algunas cosas que podemos identificar en el texto:

- Está en inglés (no se preocupe, ¡lo vamos a reescribir!)
- Hay secciones en con texto con el fondo blando y el código R está en secciones plomas (por defecto, estos colores del fondo se pueden cambiar).

Lo primero que vamos a hacer es generar el código HTML (Knit HTML) haciendo clic en la pelota de lana azul en la parte superior del panel con el archivo RMarkdown. Cuando haga clic por primera vez en este botón, RStudio pedirá que guarde el archivo. Cree un nuevo directorio en algún lugar que le sea fácil encontrarlo posteriormente (por ejemplo Escritorio o Mis Documentos), dele un nombre que tenga sentido y le permita recordarlo después (ej: `clase_rmarkdown`).

¿Qué cosas puede notar entre al comparar los dos archivos?

Las **secciones de código R** plomas están rodeadas de 3 tics y `{r LABEL}`. Estas secciones son evaluadas por R, generando una salida de texto en el caso de `summary(cars)` y de un gráfico en el caso de `plot(pressure)`.

Note como el código `plot(pressure)` no se muestra en la salida HTML porque la sección de código R tiene la opción `echo=FALSE`.

Algunos otros detalles...

Este archivo de RMarkdown tiene 2 lenguajes incluidos: **R** y **Markdown**.

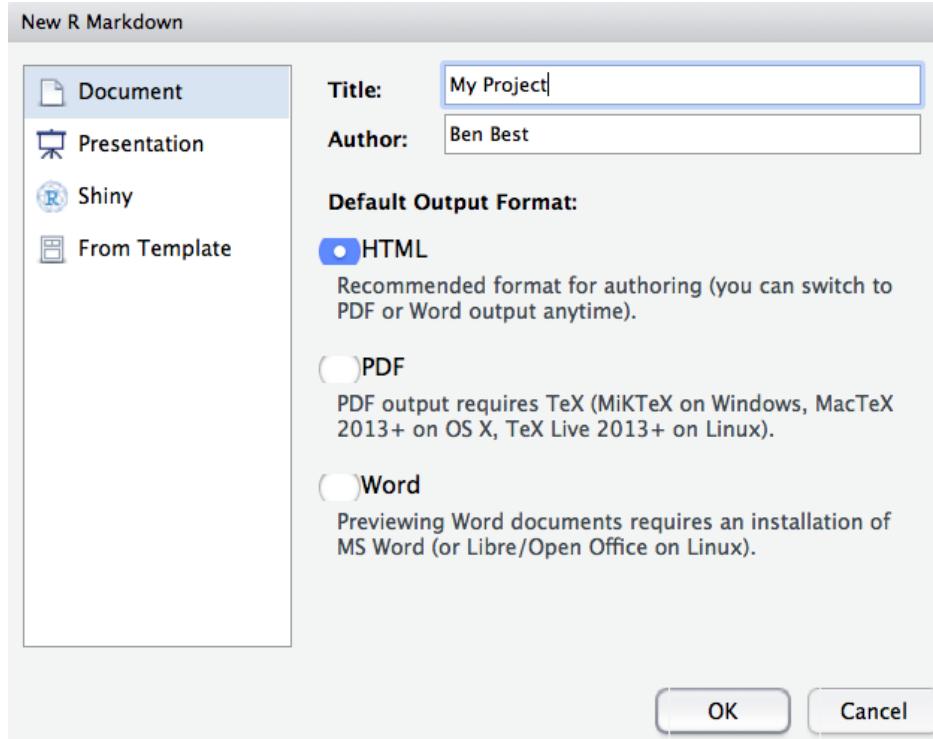


Figure 5: Captura de pantalla de la interfase para la creación de un proyecto nuevo de RStudio.

Aun no sabemos mucho de R, pero ya puede ver que estamos tomando el resumen de los datos llamados ‘cars’ y luego los graficamos. Hay mucho más para aprender sobre R y vamos a trabajar eso en las próximas clases.

El segundo lenguaje que incluye el archivo es Markdown. Este es un lenguaje que para dar formato a texto plano y tiene aproximadamente 15 reglas para conocer.

Vea la siguiente sintaxis:

- **Títulos:** son procesados en múltiples niveles con: #, ##
- **Negrita:** **palabra**

Hay muchas tablas de consejos cheatsheets que te puede utilizar para obtener ayuda, una de esas está incluida en RStudio: para abrirla vaya a Help > Markdown Quick Reference

Importante: El símbolo de gato, número o para los milenials hashtag # tiene un significado diferente al que reconoce R cuando es usado en Markdown:

- en R, un gato indica que es un comentario y no será evaluado. Se pueden usar tantos como se quiera: # es equivalente a #####. Es sólo una cuestión de estilos.
- en Markdown, un gato indica el nivel del título. Y cuantos use tienen un efecto: # es un “título nivel 1”, lo que significa que es la fuente de mayor tamaño y que está al tope de la jerarquía. ### es un título nivel 3 y se muestra anidado bajo el título # y ##.

Más sobre esto en: <http://rmarkdown.rstudio.com/>

Actividad

1. En Markdown, escriba un texto en cursiva (Itálica) y haga una lista numerada. Luego agregue algunos subtítulos. Use la Referencia de Uso Rápido de Markdown (en barra del menú: Help > Markdown Quick Reference).
2. Haga clic en Knit para general el archivo html.

```

1 ---  

2 title: "My Project"  

3 author: "Julie"  

4 date: "11/21/2017"  

5 output: html_document  

---  

7  

8 ````{r setup, include=FALSE}  

9 knitr::opts_chunk$set(echo = TRUE)  

10 ````  

11  

12 ## R Markdown  

13  

14 This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.  

15  

16 When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:  

17  

18 ````{r cars}  

19 summary(cars)  

20 ````  

21  

22 ## Including Plots  

23  

24 You can also embed plots, for example:  

25  

26 ````{r pressure, echo=FALSE}  

27 plot(pressure)  

28 ````  

29  

30 Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.  

31

```

Figure 6: Imagen con el código fuente de un archivo RMarkdown.



The screenshot shows the RStudio interface with two panes. The left pane displays the R Markdown source code, which includes code chunks for `cars` and `pressure` datasets, and a note about the `echo = FALSE` parameter. The right pane shows the generated HTML output. The HTML header includes the project title, author, date, and a note about the Knit button. Below the header is a section titled "Including Plots" containing a scatter plot of "pressure" vs "temperature". The plot shows a positive correlation with several outliers at higher temperatures.

```

1 ---  

2 title: "My Project"  

3 author: "Julie"  

4 date: "11/21/2017"  

5 output: html_document  

---  

7  

8 ````{r setup, include=FALSE}  

9 knitr::opts_chunk$set(echo = TRUE)  

10 ````  

11  

12 ## R Markdown  

13  

14 This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.  

15  

16 When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:  

17  

18 ````{r cars}  

19 summary(cars)  

20 ````  

21  

22 ## Including Plots  

23  

24 You can also embed plots, for example:  

25  

26 ````{r pressure, echo=FALSE}  

27 plot(pressure)  

28 ````  

29  

30 Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.  

31

```

Figure 7: Imagen con el código fuente y pagina resultante de un archivo RMarkdown.

Secciones del código

OK. Ahora practiquemos con algunos comandos de R.

Cree una nueva sección en su RMarkdown siguiendo alguno de estos pasos:

- Clic “Insert > R” en la parte superior del panel de edición.
- Escriba a mano “{r} “
- Si no ha borrado las secciones por defecto que venían con el archivo, editelas!

Ahora escribamos un poco de código R.

```
x <- 4*3  
x
```

Note que al apretar la tecla Enter el comando no se ejecuta; recuerde que esto es sólo un archivo de texto. Para ejecutar las secciones de código R (secciones en plomo) se deben enviar a la consola. ¿Cómo se hace eso?

Hay varias formas de hacerlo:

1. Copiar y pegar los comandos en la consola.
2. Seleccionar la línea (o poner el cursor en ella) y hacer clic en ‘Run’. Esto está disponible desde:
 - a. la barra arriba del archivo (flecha verde).
 - b. la barra menú: Code > Run Selected Line(s).
 - c. combinación de teclas: control-enter en Windows o command-enter en macOS.
3. Clic en la flecha verde en el lado superior derecho de su sección de código.

Actividad

Agregue otros comandos a su archivo. Ejecútelo de las tres formas indicadas más arriba. Finalmente, guarde su archivo RMarkdown.

Funciones de R

Hasta ahora hemos aprendido la sintaxis básica y conceptos de programación en R, como navegar en RStudio y RMarkdown, pero no hemos hecho nada complicado o interesante aún. Aquí es donde las *funciones* de R son útiles.

Una función es una forma de agrupar una serie de comandos para desarrollar una tarea en una forma reusable. Cuando una función es ejecutada, se produce un valor de salida. Nosotros generalmente decimos que “llamamos” una función cuando la ejecutamos. Lo interesante sobre las funciones es que pueden ser creadas por los usuarios y guardadas como un objeto usando el operador de asignación (‘<-’), de esta forma es posible escribir una función para lo que sea que necesitemos. Tenga en cuenta que R tiene una colección gigantesca de funciones pre-existentes, las que además son expandibles con paquetes o librerías. Para comenzar, vamos a usar algunas funciones de R básicas ya incluidas.

Todas las funciones se llaman usando la misma sintaxis: nombre de la función seguido de un paréntesis donde se entrega lo que necesita la función para funcionar. Las piezas de información que la función necesita para hacer su trabajo se conocen como argumentos. De esta forma, la sintaxis para usar una función se ve así: `valor_resultante <- nombre_de_función(argumento1 = valor1, argumento2 = valor2, ...)`.

Ejemplo simple

Para tomar un ejemplo muy simple, vamos a utilizar una función que calcula el promedio, `mean()`. Como se puede imaginar, esta función calcula el promedio de los números entregados como argumentos.

Creemos un vector con pesos:

```
peso_kg <- c(55, 25, 12)
```

y ahora usemos la función `mean` para calcular el peso promedio.

```
mean(peso_kg)
```

```
## [1] 30.66667
```

Páginas de ayuda

Obteniendo ayuda

¿Qué pasa ahora si sabe cuál es la función que necesita pero no cómo funciona? Afortunadamente RStudio entrega una forma muy fácil para acceder a las páginas de documentación y ayuda.

Para acceder a la ayuda para la función `mean`, entre el siguiente comando en la consola:

```
?mean
```

Esto abrirá el panel de ayuda en la sección inferior derecha de RStudio.

Las páginas de ayuda están divididas en secciones, la documentación está en inglés:

- *Description*: Se entrega una descripción extendida de lo que hace la función.
- *Usage*: Los argumentos de la función y sus valores por defecto.
- *Arguments*: Una explicación de los datos que cada argumento espera.
- *Details*: Detalle que sean relevantes de conocer.
- *Value*: Los resultados devuelve la función.
- *See Also*: Otras funciones que podrían resultar útiles en este contexto.
- *Examples*: Algunos ejemplos de como usar la función.

Actividad

Ejercicio: Hable con su(s) compañero(s) y dele una mirada a las ayudas de algunas funciones que usted espera que existan. Aquí les dejo algunas ideas: `?getwd()`, `?plot()`, `?min()`, `?max()`, `?log()`.

También existe ayuda para cuando usted no está seguro del nombre de la función, en esos casos solo es necesario usar doble símbolo de pregunta:

```
??install
```

No todas las funciones tienen (o requieren) argumentos:

```
date()
```

```
## [1] "Tue Oct 8 16:26:20 2024"
```

Leyendo un archivo de datos a R

Hasta ahora hemos aprendido como asignar valores a objetos de R y lo que es una función, pero no hemos hecho nada aún con datos reales. Ahora vamos a introducir la función `read.csv`, que nos permite cargar a R datos reales desde un archivo CSV. Esta función se usa generalmente en las primeras líneas del código que escribamos.

Ya que esta será nuestra primera vez usando esta función, primero leeremos la página de ayuda de `read.csv`. La ayuda tiene mucha información, mostrando que la función acepta muchos argumentos, sin embargo el primero de ellos es el más importante - tenemos que decirle que archivo es el que va a leer.

Descargue un archivo desde el Arctic Data Center

Navegue al set de datos de Craig Tweedie @tweedie_2009 que está publicado en el Arctic Data Center y descargue el primer archivo csv llamado “BGchem2008data.csv”. Mueva este archivo desde su carpeta de

Descargas a un directorio de fácil acceso. Yo recomiendo crear un directorio llamado `datos` en el directorio que creó anteriormente para las otras actividades de esta clase.

Ahora tenemos que indicarle a la función `read.csv` como encontrar el archivo que queremos leer. Para esto se usa el argumento `file` (archivo) que se puede ver en la sección usage (uso) de la página de ayuda. En RMarkdown, se pueden usar path absolutos (que comienzan con su directorio `~/`) o path relativos, que son **relativos a la ubicación del archivo RMarkdown**. RStudio y RMarkdown tiene la capacidad de autocompletar cuando se usan paths relativos, por lo que usaremos ese sistema. Asumiendo que ya movió el archivo a el directorio dentro de la carpeta `archivos_clase_R` llamada `datos`, su función `read.csv` puede ser llamada de la siguiente forma:

```
bg_chem <- read.csv("datos/5.1_BGchem2008data.csv")
## El nombre fue cambiado para hacer referencia que el archivo csv se usa en el capítulo 5.
```

Ahora debería tener un objeto en R llamado `bg_chem` de la clase `data.frame` en su ambiente de trabajo. Verifique `data.frame` existe y contiene los datos.

Note que en las páginas de ayuda hay una gran cantidad de argumentos que no fueron necesarios de utilizar. Algunos argumento en las llamadas a funciones son opcionales y algunos otros son requeridos. Los argumentosopcionales son mostrados en la sección de uso como un par `nombre = valor`, donde se muestra el valor que usa esa opción por defecto. Si usted no especifica el valor para la opción en particular, la función asume el valor por defecto (ejemplo: `header = TRUE` para `read.csv`). Los argumentos requeridos sólo muestran el nombre del argumento, sin un valor asociado. Fíjese que el único argumento requerido para `read.csv` es `file`.

Siempre se puede especificar un argumento de la forma `nombre = valor`. Pero si no lo hace, R intentará identificarlos basado en el orden en que fueron entregados. De esta forma, en el comando de más arriba, se asume que queremos `file = "data/5.1_BGchem2008data.csv"`, ya que `file` es el primer argumento. Si queremos agregar algún otro argumento, por ejemplo `stringsAsFactors`, vamos a necesitar especificarlos usando `nombre = valor`, ya que el segundo argumento es `header`. Para las funciones que llamo comúnmente, uso esto sólo para los primeros 2 argumentos, para los que están en la 3a posición o más alla siempre uso el par `nombre = valor`.

Muchos usuarios de R (incluyéndome) sobrescriben el argumento por defecto de `stringsAsFactors` usando la siguiente llamada:

```
bg_chem <- read.csv("data/5.1_BGchem2008data.csv", stringsAsFactors = FALSE)
```

Esto muy útil, ya que de otra forma las columnas que contiene datos tipo strings son transformados a factores, pudiendo generar esto problemas y confusiones.

Usando `data.frames`

Un `data.frame` en R es una estructura de datos de dos dimensiones, es similar a como se comporta una hoja de cálculos. Un `data.frame` es una colección de filas y columnas de datos, donde cada columna tiene un nombre y representa una variable, siendo cada fila a una medición de esa variable. De esta forma, cuando ejecutamos `read.csv`, el objeto `bg_chem` fue creado como un objeto tipo `data.frame`. Hay muchas formas para explorar un `data.frame` en R y RStudio. A continuación mostramos algunos:

- Clic en la palabra `bg_chem` en el panel *Environment*.
- Clic en la flecha junto a `bg_chem` en el panel de *Environment*
- Ejecute `head(bg_chem)` en la consola.
- Ejecute `View(bg_chem)` en la consola.

Usualmente las funciones se ejecutan en columnas individuales de un `data.frame`. Para llamar a una columna específica se usa el operador `$`. Por ejemplo, digamos que se quiere mirar las primeras filas sólo de la columna `Date`. Para esto se debe ejecutar en la consola:

```
head(bg_chem$Date)

## [1] "2008-03-21" "2008-03-21" "2008-03-21" "2008-03-21" "2008-03-21"
## [6] "2008-03-22"
```

¿Cómo se puede calcular la temperatura promedio de las muestras tomadas por el CTD?

```
mean(bg_chem$CTD_Temperature)
```

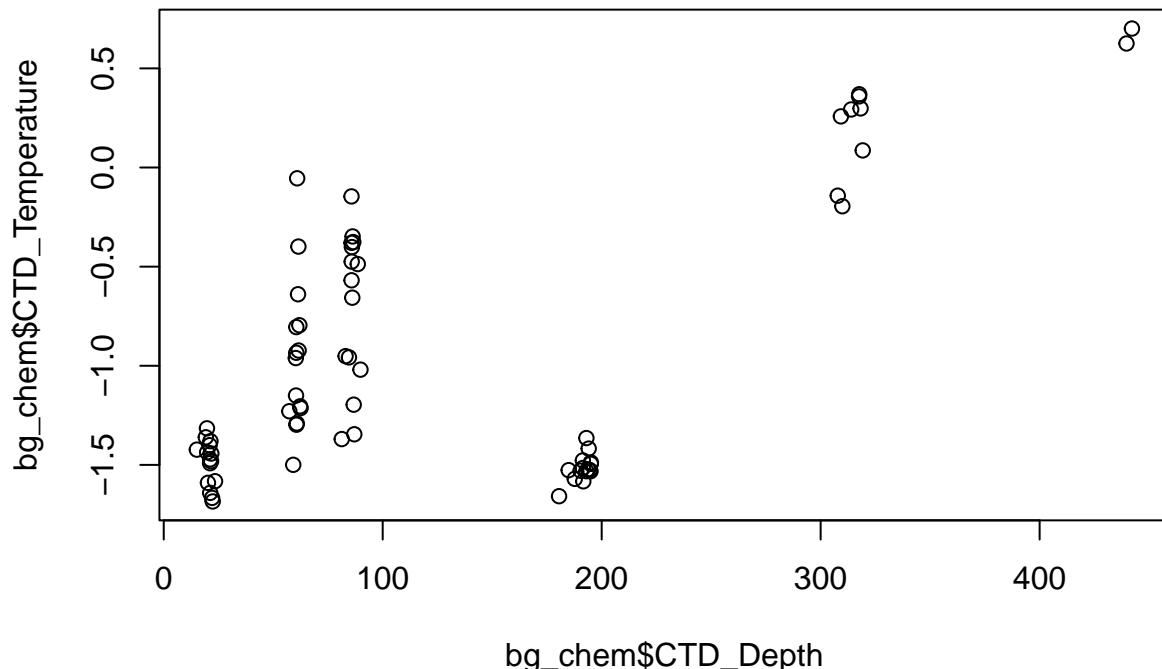
```
## [1] -0.9646915
```

O si quisiéramos guardar este promedio como una variable para ser utilizada después:

```
temp_promedio <- mean(bg_chem$CTD_Temperature)
```

También es posible crear algunos gráficos simples utilizando este operador (\$).

```
plot(bg_chem$CTD_Depth, bg_chem$CTD_Temperature)
```



En R existen muchas herramientas y funciones más avanzadas para crear mejores gráficos (incluso con una sintaxis más simple). Veremos algunas de ellas en capítulos posteriores.

Actividad

Ejercicio: Tome algunos minutos en explorar este set de datos. Pruebe algunas funciones en las columnas utilizando el operador \$ para seleccionar una columna única. Experimente haciendo gráficos y generando resúmenes de los datos.

Solución de problemas

Mi RMarkdown no genera (knit) el PDF

Si se obtiene un error cuando se trata de generar el PDF, donde dice que su computador no tiene la instalación de LaTeX, existen dos posibles problemas:

1. Su computador no tiene LaTeX instalado.
2. Su computador si tiene LaTeX instalado, pero RStudio no lo puede encontrar (no está definida la ruta o path).

Si usted ya usa LaTeX (para escribir papers por ejemplo), seguro que su problema esté en la segunda categoría. Arreglar esto sólo requiere redireccionar RStudio a donde este instalado LaTeX, pero esto no lo explicaremos acá.

Si su problema es que no tiene instalado LaTeX y **está seguro que no lo tiene!** puede usar el paquete de R `tinytex`, este se instala muy fácilmente y es reconocido automáticamente por RStudio. Para hacer esto sólo necesita tener derechos de administrador del computador.

Para instalar `tinytex` ejecute en la consola:

```
install.packages("tinytex")
tinytex::install_tinytex()
```

Acabo de ejecutar un comando pero no pasa nada

Esto puede ocurrir porque no completó el comando: ¿Se muestra un pequeño signo + en su consola? R le esta diciendo que está esperando a que termine de entregar los comandos. En el ejemplo de más abajo falta cerrar el paréntesis.

```
> x <- seq(1, 10
+
```

Se puede simplemente cerrar el paréntesis y apretar enter, o `esc` dos veces, lo que va a cancelar cancelar la ejecución del comando.

R dice que mi objeto no puede ser encontrado

Este error es muy común en usuarios nuevos: `Error in mean(myobject) : object 'myobject' not found`

Esto significa que el objeto llamado `myobject` no existe en el ambiente de trabajo. Las razones más comunes para este error son:

- **Error de tipeo:** asegúrese que el nombre de su objeto esta bien escrito, R hace diferencias entre mayúsculas y minúsculas. Debe estar escrito *exactamente* como fue creado.
- **No fue asignado a una variable:** note que los objetos sólo son guardados en el ambiente de trabajo si fueron asignado con el operador de asignación, ej: `myobject <- read.csv(...)`.
- **No ejecute la linea en RMarkdown:** recuerde que escribir una línea de código en RMarkdown no es lo mismo que escribirla en la consola, usted tiene que ejecutar la línea de código usando `control + enter`, ejecutando la sección de código o por alguna de las otras formas que fueron descritas en la sección @ref(ejecutarCodEnRmd) de este mismo capítulo.

Análisis literal

RMarkdown es una excelente forma de generar análisis literales y flujos de trabajo reproducibles. Aquí hay un ejemplo en inglés de un flujo de trabajo real escrito utilizando RMarkdown.

Control de versiones con git y GitHub

Objetivos

En este capítulo veremos:

- ¿Por qué **git** es útil para análisis reproducibles?
- ¿Cómo usar **git** para registrar los cambios que se hacen en el tiempo?
- ¿Cómo usar **GitHub** para colaborar?
- ¿Cómo estructurar los “*commits*” para que los cambios sean claros para otros?
- ¿Cómo escribir mensajes de “*commits*” que sean efectivos?

El problema con los nombres de archivo

Cada archivo en un proceso científico sufre de cambios. Los manuscritos son editados. Las figuras son revisadas. Los códigos se corrigen cuando se encuentran problemas. Los archivos de datos se combinan, los errores son corregidos, se dividen y combinan nuevamente. En el curso de un análisis simple, uno puede esperar miles de cambios en los archivos. Y aún así, todo lo que usamos para identificas este sinnúmero de cambios son los simples **nombres de archivos** (fig. @ref(fig:finalDoc2)). Teniendo esto en consideración, es lógico pensar que debe existir una forma mejor... Y si la hay, se conoce como **Control de Versiones**.

Un **sistema de control de versiones** ayuda a seguir los cambios que se realizan a nuestros archivos, sin el desastre que resulta de utilizar sólo el nombre de los archivos. En los sistemas de control de versiones como **git**, se registra no sólo el nombre del archivo, si no que además su contenido, de esta forma, cuando el contenido cambia, se puede identificar que partes estaban y donde. El registro además contiene la relaciones entre las versiones, de esta forma se tiene un historial de todos los cambios derivados de las cada una de las versiones y es fácil dibujar un gráfico que muestre los cambios que ha sufrido un archivo, con sus versiones previas y aquellas derivadas (Fig. @ref(fig:figVersiones2))

Los sistemas de control de versiones asignan un identificador a cada versión de cada archivo, manteniendo un registro de como están relacionados entre ellos. Además, estos sistemas permiten generar ramas laterales a una versión, la que puede ser fusionada de regreso a la tronco principal. Es posible además tener *múltiples copias* en múltiples computadores como respaldos y para trabajar colaborativamente. Finalmente, se pueden incluir etiquetas (tags) a versiones particulares, de esta forma es fácil retornar la versión que tenían los archivos cuando fueron etiquetados. Esto es particularmente útil para identificar una versión exacta de los datos, código y texto, por ejemplo, de un manuscrito que fue enviado para ser publicado, este puede ser el caso de la etiqueta R2 en la figura @ref(fig:figVersiones2).

Control de versiones y colaboración usando Git and GitHub

Es importante hacer la distinción entre **git** y **GitHub**.

- **git**: Software para el control de versiones que monitorea los archivos de un directorio (repositorio).
 - git crea el historial de versiones del repositorio.
- **GitHub**: Sitio web que permite a los usuarios almacenar sus repositorios git y compartirlos con otros usuarios.

Veamos un repositorio de GitHub

La captura de pantalla en la figura @ref(fig:webGitHub) muestra una copia de un repositorio almacenado en GitHub, con el listado de archivos y directorios, indicando cuando fueron modificados, incluyendo información acerca de quien hizo los cambios y una pequeña descripción de los cambios realizados.

Si nos metemos en los “*commits*” del repositorio (fig. @ref(fig:githubCommits)), podemos ver la historia de los cambios que se le han realizado. Por ejemplo, se ve que **kellijohnson** y **seananderson** hicieron algunos cambios durante junio y julio:

"FINAL".doc



FINAL.doc!



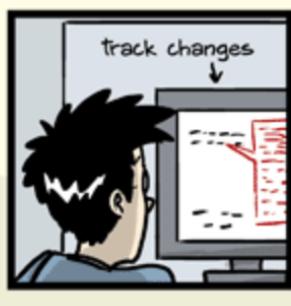
FINAL_rev.2.doc



FINAL_rev.6.COMMENTS.doc



FINAL_rev.8.comments5.
CORRECTIONS.doc



FINAL_rev.18.comments7.
corrections9.MORE.30.doc



FINAL_rev.22.comments49.
corrections.10.#@\$%WHYDID
ICOMETOGRAD SCHOOL????.doc



Figure 8: El dilema de usar nombres como descriptor de versiones.

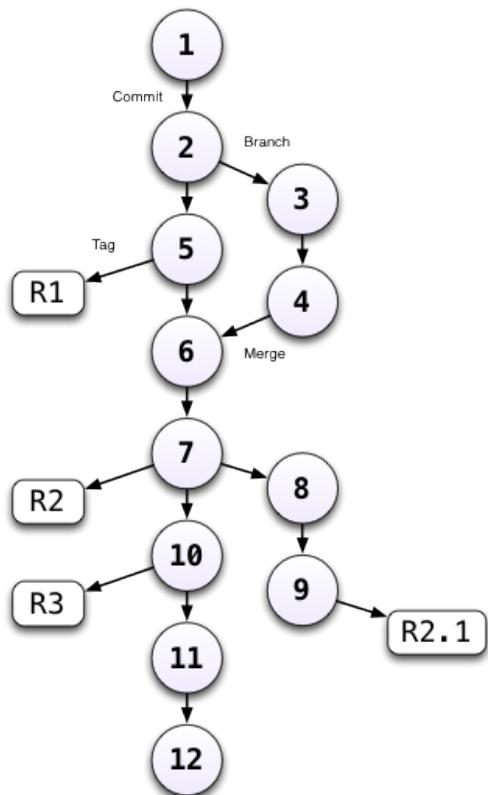


Figure 9: Evolución de las versiones de un archivo.



Figure 10: Diferencia entre git y GitHub.

The screenshot shows a GitHub repository page for the 'ss3sim' organization and the 'ss3sim' repository. At the top, there's a header with a lock icon, the URL 'https://github.com/ss3sim/ss3sim', a progress bar at 133%, and navigation links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. Below the header, the repository name 'ss3sim / ss3sim' is displayed, along with metrics: 18 stars, 10 forks, and 12 open issues. A navigation bar below the repository name includes links for 'Code', 'Issues 13', 'Pull requests 0', 'Projects 0', 'Wiki', and 'Insights'. The main content area is titled 'An R package for stock-assessment simulation with Stock Synthesis' and lists tags: 'r', 'fisheries', and 'stock-synthesis'. It features a summary bar with statistics: 2,762 commits, 14 branches, 9 releases, and 11 contributors. Below this, a list of recent commits is shown, each with a user icon, author, commit message, file changes, and timestamp. The commits are as follows:

Author	Commit Message	File Changes	Date
kellijohnson	Fix example in sample_lcomp	R	Latest commit 23d453b on Jul 6
	Fix example in sample_lcomp	R	5 months ago
	Add microbenchmark tests of parallel options	benchmarks	3 years ago
	Replace data with compressed versions	data	4 years ago
	Add missing case file for vignette to run	inst	5 months ago
	add ESS as argument to sample_comp functions	man-roxygen	2 years ago
	Fix example in sample_lcomp	man	5 months ago
	Tweaks to pass R CMD check on R-devel	tests-extra	2 years ago
	Add unit tests	tests	a year ago
	Add todo note and change scenario to available case files	vignettes	5 months ago

Figure 11: Captura de pantalla de un repositorio en GitHub.

The screenshot shows a GitHub repository page for 'ss3sim / ss3sim'. The top navigation bar includes links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. Below the header, there are buttons for 'Watch' (18), 'Star' (10), and 'Fork' (12). The main content area displays a list of commits, grouped by date. Each commit card includes the author's profile picture, the commit message, the date it was committed, and three action buttons: a copy icon, a link to the commit hash, and a diff icon.

- Commits on Jul 6, 2017:
 - Fix example in sample_lcomp (kellijohnson, Jul 6) - Hash: 23d453b
 - Add todo note and change scenario to available case files (kellijohnson, Jul 6) - Hash: 1b2c0a9
 - Add missing case file for vignette to run (kellijohnson, Jul 6) - Hash: 57242d2
- Commits on Jun 29, 2017:
 - Bump development version (seananderson, Jun 29) - Hash: 5fbfb966
- Commits on Jun 14, 2017:
 - Fix broken link in vignette introduction to github in sect 13. (kellijohnson, Jun 14) - Hash: 0e0cb2d
- Commits on Jun 13, 2017:
 - Fix get-results functions to work with new version of r4ss. (kellijohnson, Jun 13) - Hash: 9971d7a
 - Fixed change_year and change_e to work with the newest version of r4ss. (kellijohnson, Jun 13) - Hash: b864fbf

Figure 12: Captura de pantalla de los Commits de un repositorio en GitHub.

Si entramos ahora a ver los cambios realizados el 13 de julio (fig. @ref(fig:githubDiferencias)), podemos saber exactamente cuales fueron los cambios realizados a cada archivo:

```

Fix get-results functions to work with new version of r4ss.
r4ss now uses Label instead of LABEL in the derived quantities.

Browse files

Commit 9971d7af8610358c55b89d2175ab1686b3e6a5f6
Author: kellijohnson committed on Jun 13
1 parent b864fbf

Showing 1 changed file with 28 additions and 15 deletions.

Unified Split
View ▾

43 R/get-results.r
@@ -440,17 +440,23 @@ get_results_timeseries <- function(report.file){
 440   xx <- xx[xx$Yr %in% years,]
 441   names(xx) <- gsub(":_1","", names(xx))
 442   # Get SPR from derived_quants
 443   - spr <- with(report.file$derived_quants,
 444   -   report.file$derived_quants[grep("SPRratio_", LABEL), ])
 445   - spr$Yr <- sapply(strsplit(spr$LABEL, "_"), "[", 2)
 443   + spr <- report.file$derived_quants[grep("SPRratio_",
 444   +   report.file$derived_quants[,,
 445   +     grep("label", colnames(report.file$derived_quants),
 446   +       ignore.case = TRUE)]), ]
 447   + spr$Yr <- sapply(strsplit(
 448   +   spr[, grep("label", colnames(spr), ignore.case = TRUE)], "_"), "[", 2)
 446   449   colnames(spr)[which(colnames(spr) == "Value")] <- "SPRratio"
 447   450   # Get recruitment deviations
 448   451   dev <- report.file$recruit

```

Figure 13: Captura de pantalla donde se presentan las diferencias entre dos versiones alojadas en un repositorio de GitHub.

Monitorear estas modificaciones, como se relacionan a cada una de las versiones de un software en particular y a los archivos es exactamente para lo que fueron diseñados git y GitHub. Vamos a mostrar como estos sistemas pueden ser realmente efectivos para monitorear las versiones de códigos científicos, figuras y manuscritos y de esta forma tener flujos de trabajo reproducibles.

El ciclo de vida de Git

Como usuario de git usted tiene que entender algunos conceptos básicos asociados a los sets de cambios con versiones y como estos son almacenados y se mueven a través del repositorio. Cualquier repositorio de git puede ser clonado, de esta forma existe en forma local y remota. Pero cada uno de estos repositorios clonados son una copia simple de todos los archivos y del historial de los cambios que se han realizado, lo que son almacenados en un forma repositorio git particular. Para nuestros propósitos, tenemos que considerar un repositorio de git simplemente como un directorio que contiene adicionalmente un set de metadatos relacionado las versiones.

En un directorio local donde git fue habilitado, el directorio contiene la versión actual de todos los archivos del repositorio. Estos archivos de trabajo están unidos a un directorio escondido que contiene el 'Repositorio

local', el que a su vez contiene todos los otros cambios que se han realizado a los archivos y los metadatos sobre las versiones.

De esta forma, cuando se está usando git para trabajar con archivos, se pueden usar los comandos de git para indicar específicamente que cambios a los archivos de trabajo deben ser definidos para las versiones (usando el comando `git add`) y cuando grabar estos cambios como una versión en el repositorio local (usando el comando `git commit`).

Los demás conceptos se relacionan a la sincronización de los cambios en el repositorio local a un repositorio remoto. El comando `git push` se usa para enviar los cambios realizados en forma local a un repositorio remoto (posiblemente en GitHub) y el comando `git pull` es usado para traer los cambios del repositorio remoto y unirlos al repositorio local.

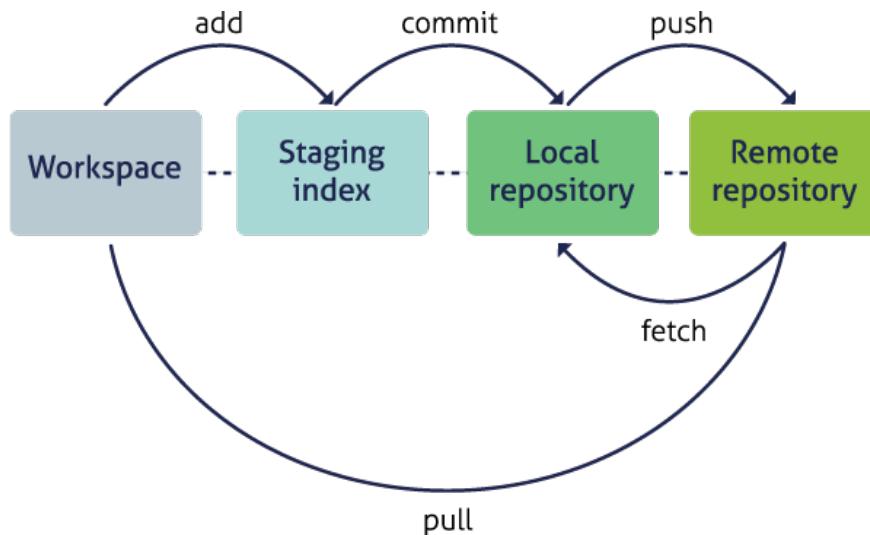


Figure 14: Diagrama de flujo del ciclo de vida de git.

- `git clone`: Copia todo el repositorio remoto a uno local.
- `git add (stage)`: Notifica a git de monitorear un set particular de cambios.
- `git commit`: Almacena los cambios realizados como una versión.
- `git pull`: Combina los cambios de un repositorio remoto a nuestro repositorio local.
- `git push`: Copia los cambios de nuestro repositorio local al repositorio remoto.
- `git status`: Determina el estado de los archivos en el repositorio local.
- `git log`: Imprime el historial de cambios en el repositorio.

Estos siete comandos son la mayoría de los comandos que va a necesitar para utilizar git de forma exitosa. Pero todo esto es demasiado abstracto, mejor exploremos estos conceptos utilizando ejemplos reales.

Cree un repositorio remoto en GitHub

Empecemos creando un repositorio en GitHub, luego editaremos algunos archivos.

- Ingrese al sitio GitHub.
- Clic en el botón de repositorio nuevo (New repository).
- Nómbralo como `sasap-test`.
- Cree un archivo `README.md`.
- Defina la licencia a Apache 2.0.

¡Usted acaba de crear su primer repositorio! Este repositorio fue creado con un par de archivos que GitHub hace por usted, son los archivos `README.md`, `LICENSE` y `.gitignore`.

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner Repository name

 **mbjones** / sasap-test ✓

Great repository names are short and memorable. Need inspiration? How about [fantastic-potato](#).

Description (optional)

This is a test repository

 **Public**
Anyone can see this repository. You choose who can commit.

 **Private**
You choose who can see and commit to this repository.

Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: R ▾ Add a license: Apache License 2.0 ▾ ⓘ

Create repository

Figure 15: Captura de pantalla de la creación de un repositorio nuevo en GitHub.

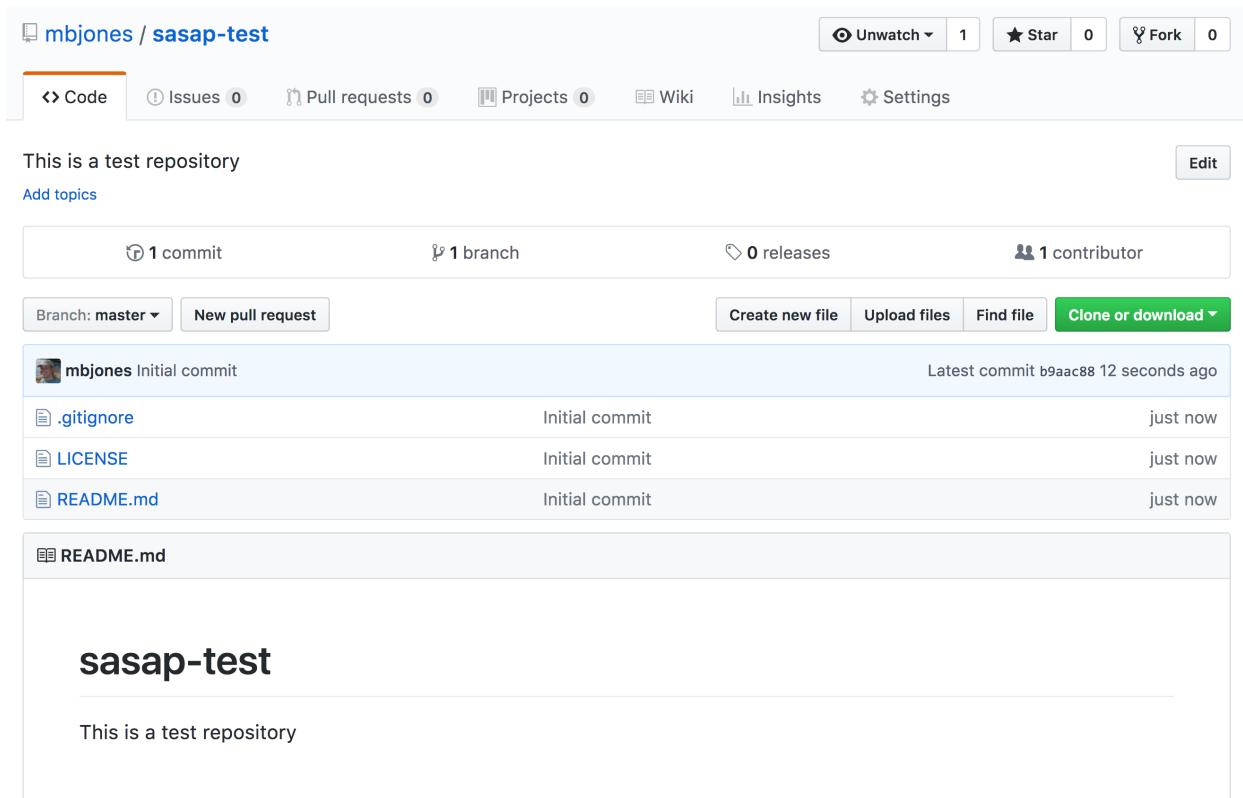


Figure 16: Captura de pantalla con el repositorio recién creado llamado sasap-test.

Para hacer cambios menores a archivos de texto se puede utilizar la interfase web de GitHub. Navegue al archivo README.md en el listado de archivos y habilite la edición haciendo clic en el icono del *lápiz*. Este es un archivo normal con el formato Markdown, ahora se puede editar, agregando o removiendo texto. Cuando haya terminado, incluya un mensaje de commit y luego haga clic en el botón **Commit changes**.

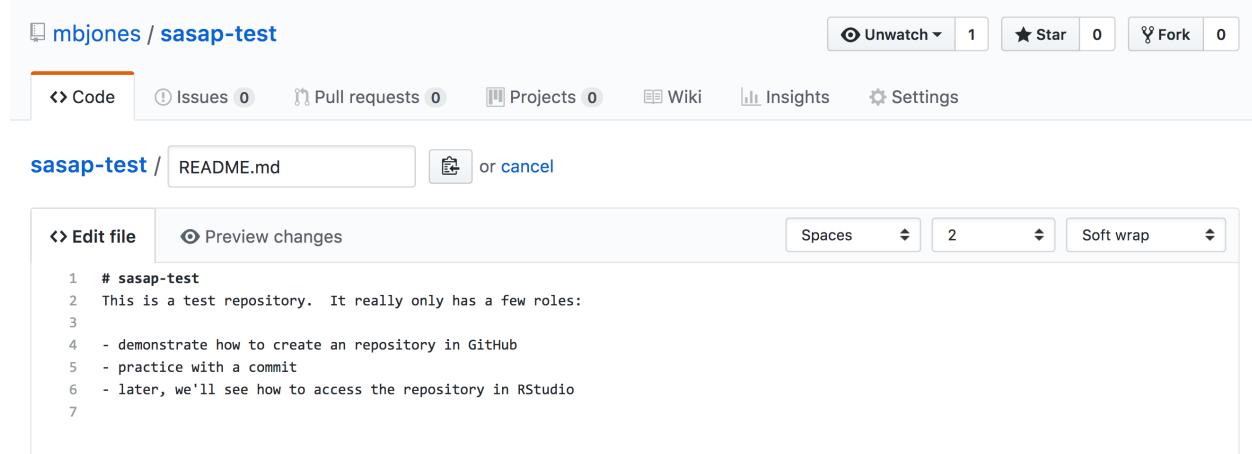


Figure 17: Captura de pantalla con la interfase web de GitHub para la edición de documentos de texto.

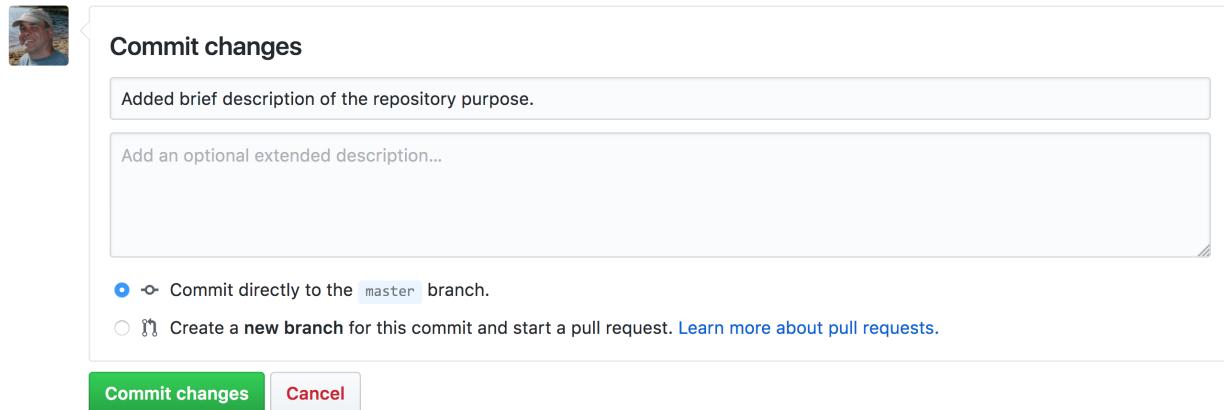


Figure 18: Captura de pantalla con la interfase web de GitHub para la confirmar de los cambios realizados al documento de texto (commit changes).

Felicitaciones, ahora usted acaba de confirmar (commit) los cambios, ahora es el autor de la primera versión de este archivo. Si navega de regreso a la página del repositorio GitHub, puede ver la lista de los cambios confirmados (commits) ahí, así como la visualización del documento generado a partir del archivo README.md.

Expliquemos algunas cosas sobre esta ventana. Representa una vista del repositorio que acaba de crear, hasta ahora mostrando todos sus archivos. Para cada archivo, muestra la fecha de la última modificación y el mensaje asociado al *commit* que se utilizó para describir los cambios realizados. Esta es la razón por qué es tan importante escribir buenos mensajes, que contengan información relevante cuando se hace el *commit*. Además, el título azul sobre el listado de archivos muestra el *commit* más reciente, con su mensaje asociado y su identificado SHA. Ese identificador SHA es la clave para el set de versiones. Si hace clic en el identificador SHA (*810f314*), va a mostrar los cambios que se hicieron en ese *commit* en particular.

En la siguiente sección vamos a usar el URL de GitHub del repositorio que acaba de crear y lo usaremos para clonarlo (*clone*) a un repositorio en nuestra máquina local y así editar los archivos con RStudio. Para

The screenshot shows a GitHub repository page for 'mbjones / sasap-test'. At the top, there are buttons for Unwatch (1), Star (0), and Fork (0). Below the header, there are tabs for Code (selected), Issues (0), Pull requests (0), Projects (0), Wiki, Insights, and Settings. A main message says 'This is a test repository' with an 'Edit' button. There's a link to 'Add topics'. Below this, stats show 2 commits, 1 branch, 0 releases, 1 contributor, and Apache-2.0 license. A dropdown shows 'Branch: master' and a 'New pull request' button. To the right are buttons for 'Create new file', 'Upload files', 'Find file', and a green 'Clone or download' button. The commit history lists three entries: 'mbjones Added brief description of the repository purpose.' (25 seconds ago), '.gitignore' (Initial commit, 14 minutes ago), 'LICENSE' (Initial commit, 14 minutes ago), and 'README.md' (Added brief description of the repository purpose., 25 seconds ago). Below the commit history is a section for 'README.md' with the title 'sasap-test'. The README content is: 'This is a test repository. It really only has a few roles:' followed by a bulleted list: • demonstrate how to create an repository in GitHub • practice with a commit • later, we'll see how to access the repository in RStudio.

Figure 19: Captura de pantalla con la interfase web los cambios corfirmados (commits) y de la visualización de la página README.md.

eso, primero tenemos que copiar el URL de GitHub, que representa la dirección del repositorio:

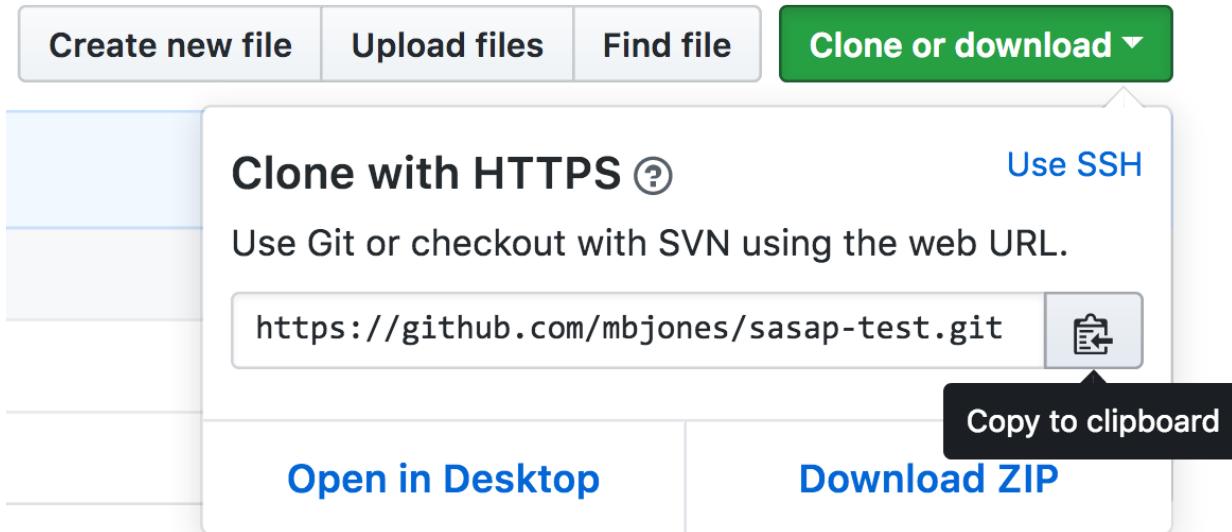


Figure 20: Captura de pantalla para clonar un repositorio GitHub.

Trabajando localmente con Git en RStudio

RStudio incluye soporte para Git como sistema de control de versiones, pero esto **sólo** ocurre si estamos trabajando en un *Proyecto de RStudio* (RStudio project folder). En esta sección vamos a clonar el repositorio que se creó en GitHub y lo vamos a dejar un repositorio local de un proyecto de RStudio.

Esto es lo que vamos a hacer:

1. Crear un proyecto nuevo.
2. Inspeccionar el panel Git y el historial de versiones.
3. Confirmar una modificación (commit) al archivo README.md.
4. Commit las modificaciones que hicieron en RStudio.
5. Inspeccionar el historial de versiones.
6. Crear y commit un archivo Rmd.
7. Enviar (*Push*) estos cambios a GitHub.
8. Ver el historial de cambios en GitHub.

Crear nuevo Proyecto (Create a New Project)

Comience creando un *New Project...* en RStudio, seleccione la opción *Version Control* y pegue el URL de GitHub que copió en el espacio para repositorio remoto (*Repository URL*). Si bien usted puede darle el nombre que quiera al repositorio local, se usa típicamente el mismo nombre que el que se tiene en GitHub, de esta forma se mantiene un grado de correspondencia. Usted puede elegir cualquier directorio para su copia local, en mi caso use el directorio `development` (fig. @ref(fig:githubClone)).

Un vez que haga clic en **Create Project** (crear proyecto), una nueva página de RStudio se abrirá con todos los archivos copiados localmente desde el repositorio remoto. Dependiendo de como esté configurada la versión de RStudio, la posición y tamaño de los paneles puede cambiar, pero generalmente todos van a estar presentes, incluyendo los paneles *Git* y el listado de archivos (*Files*) que fueron creados en el repositorio remoto.

En la figura @ref(fig:githubCloneLocal) puede ver que apareció un archivo llamado `sasap-test.Rproj` y que están los otros 3 archivos que se crearon con el repositorio remoto de GitHub (`.gitignore`, `LICENSE` y `README.md`).

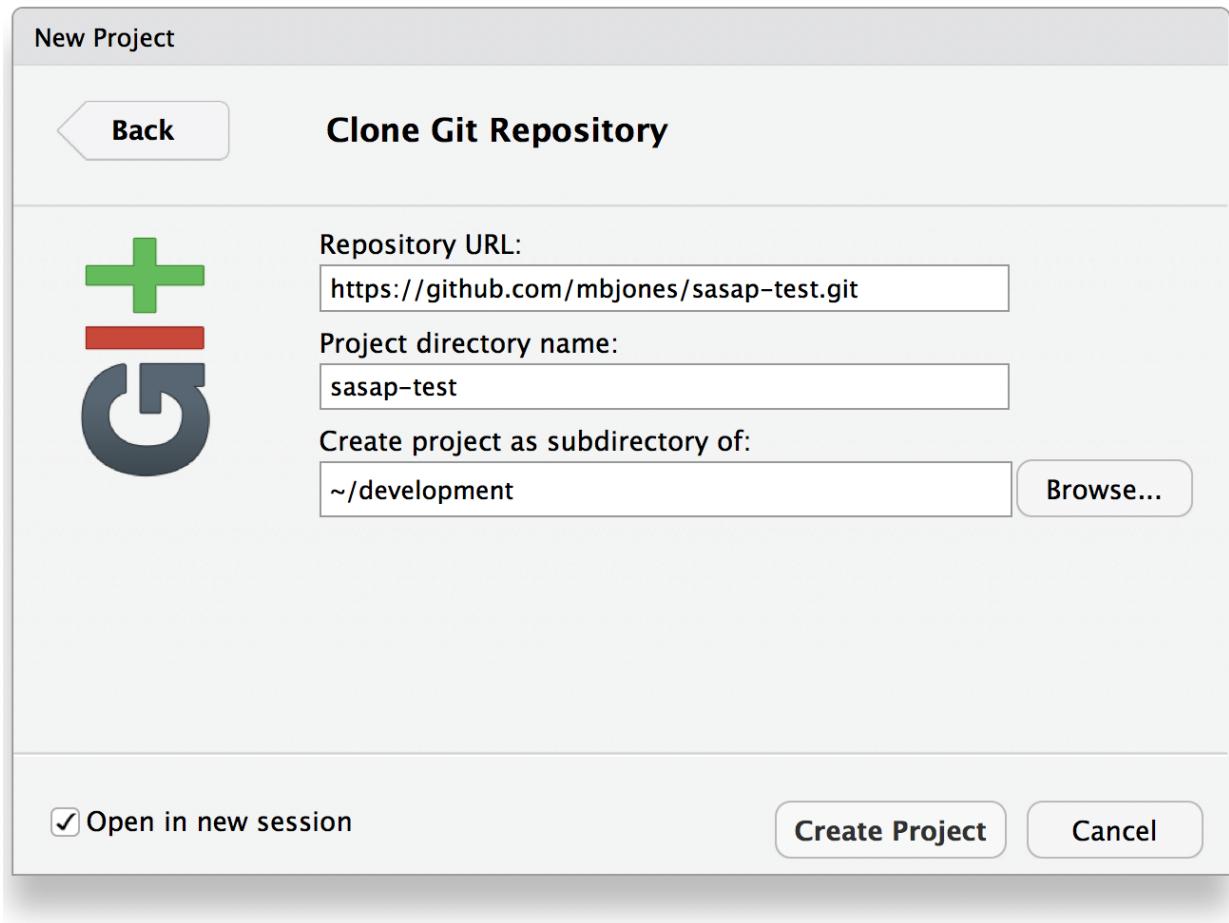


Figure 21: Captura de pantalla para la creación de un proyecto de RStudio clonando un repositorio remoto.

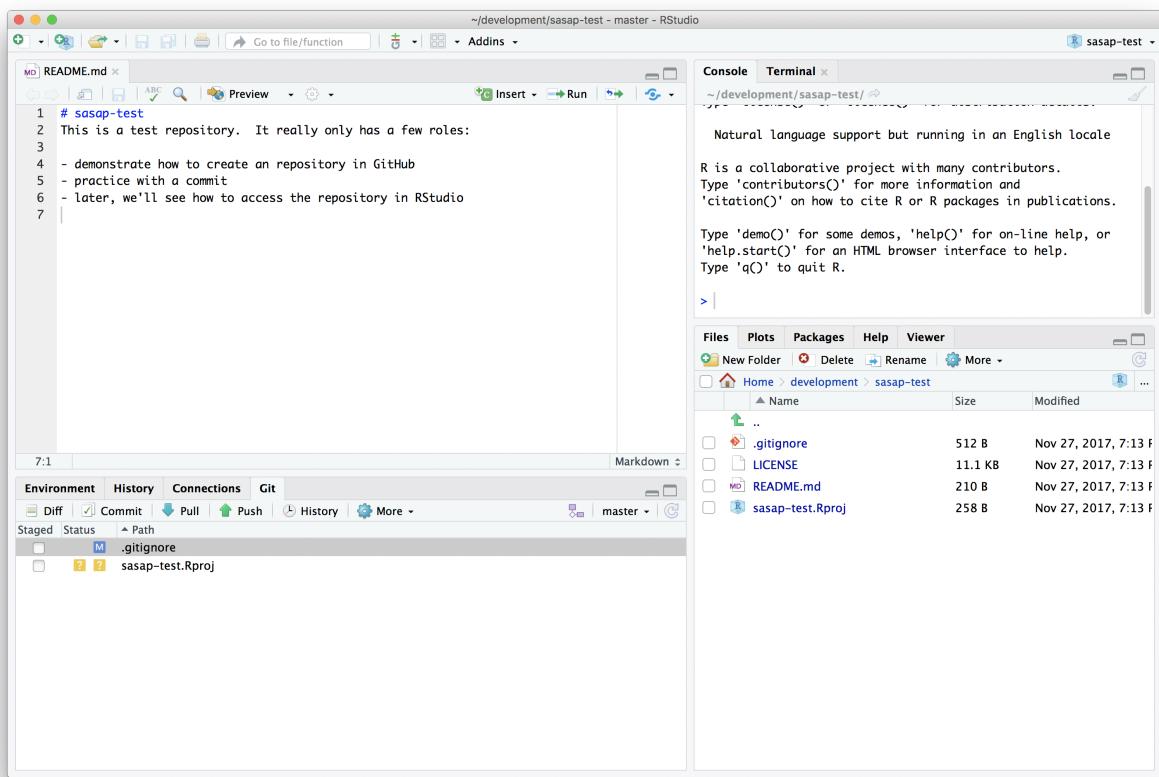


Figure 22: Captura de pantalla de la interfase del proyecto de RStudio con el clon local de repositorio.

En el panel *Git* en RStudio se pueden ver 2 archivos. Este es el panel de estatus donde se muestran todos los archivos del repositorio en los cuales se han realizado modificaciones. En este caso, el archivo `.gitignore` se muestra con una *M* que significa *Modificado* y `sasap-test.Rproj` con un *?* para indicar que este archivo no está siendo monitoreado. Esto significa que git no tiene registro de ninguna versión para este archivo y que no sabe nada acerca de él. A medida que usted vaya tomando decisiones sobre el control de versiones en RStudio, estos iconos van a ir cambiando para reflejar el estatus de la versión actual de cada uno de los archivos.

Inspeccionar el historial (history)

A continuación vamos a hacer clic en el botón *History* (historial, es el reloj que aparece en la primera fila al interior del panel ed Git), esto despliega una ventana con el registro de los cambios que se han realizado y, en este caso, deben ser idénticos a lo que usted vio en GitHub. Al hacer clic en cada fila del historial, podrá ir viendo exactamente qué fue agregado y cambiado en cada uno de los commits en este repositorio.

Confirme cambios haciendo clic en *commit* al archivo README.md (Commit a README.md change)

Ahora hagamos algunos cambios al archivo README.md en RStudio. Agregue una sección nueva con un block de *markdown* como este:

Una vez que los haya guardado, podrá ver en forma inmediata el archivo *README.md* en el panel Git (fig. @ref(fig:rstudioPanelStatus)), marcado con una **M** de modificación. Ahora usted puede seleccionar este archivo en el panel de Git y hacer clic en *Diff* para ver los cambios (comparando las diferencias) que guardó (note que estos cambios no se han confirmado (*commit*) aun a su repositorio local).

En la figura @ref(fig:rstudioDiferencias) se muestra cómo se ven los cambios comparados con el archivo original. Las líneas nuevas se destacan en color verde y en rojo las que fueron eliminadas.

Commit los cambios hechos en Rstudio

Para confirmar los cambios que se acaban de hacer en el archivo README.md, seleccione la caja de selección *Staged* a lado del nombre del archivo, este le dice a Git cuáles son los cambios que quiere sean incluidos en el commit, escriba un mensaje describiendo qué cambios se hicieron y por qué y finalmente haga clic en botón *Commit* (fig. @ref(fig:rstudioCommit1)).

Note que algunos de los cambios en el repositorio, `.gitignore` y `sasap-test.Rproj`, aun no se han confirmado y no serán parte del *commit*. En otras palabras, aun existen cambios pendientes para ser registrados en el repositorio. Usted verá una notificación que indica (en inglés):

Your branch is ahead of 'origin/master' by 1 commit.

Lo que se traduce a: Su rama está más avanzada que el 'maestro/origen' por una confirmación

Esto significa que hemos commit **1** cambio en el repositorio local, pero que este no se está en el repositorio de origen (`origin`), no se ha hecho push, donde origin es el nombre que se usa típicamente para el repositorio en GitHub. Entonces, confirmemos los cambios pendientes, para esto seleccione la caja *staged* y luego escriba el mensaje describiendo el commit (fig. @ref(fig:rstudioCommit2)).

Cuando haya terminado no habrá más cambios pendientes en el panel *Git* y el repositorio estará completamente limpio.

Inspeccionando el historial

Fíjese que ahora el mensaje dice:

Your branch is ahead of 'origin/master' by 2 commits.

Su rama está más avanzada que el 'maestro/origen' por 2 confirmación

RStudio: Review Changes

Changes History master (all commits) Pull

Subject	Author	Date	SHA
HEAD -> refs/heads/master origin/master origin/HEAD Added brief des Matt Jones <gitcode@magisa.org> 2017-11-28 810f314f	Matt Jones <gitcode@magisa.org>	2017-11-28	810f314f
Initial commit	Matt Jones <gitcode@magisa.org>	2017-11-28	b9aac88c

Commits 1–2 of 2

SHA 810f314f
Author Matt Jones <gitcode@magisa.org>
Date 2017-11-28 03:19
Subject **Added brief description of the repository purpose.**
Parent b9aac88c

README.md

View file @ 810f314f

```
@@ -1,2 +1,6 @@
1 | 1 # sasap-test
2 | This is a test repository
2 | This is a test repository. It really only has a few roles:
3 |
4 | - demonstrate how to create an repository in GitHub
5 | - practice with a commit
6 | - later, we'll see how to access the repository in RStudio
```

Figure 23: Historial de los cambios realizados en el repositorio local.

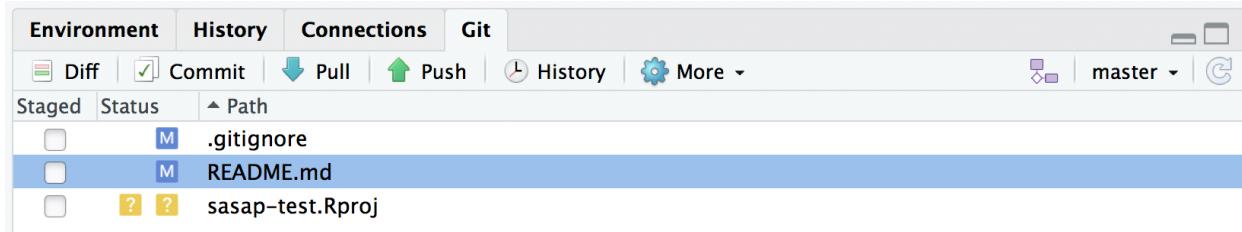


Figure 24: Panel de estado de los cambios en los archivos.

Estas 2 confirmaciones son las dos que acabamos de hacer y no se han empujado (push) aun a GitHub. Haciendo clic en el botón *History* (historial), podemos ver que hay un total de 4 commit en el repositorio local, mientras hay sólo 2 en GitHub (fig. @ref(fig:rstudioCommit3)).

Enviar (Push) cambios a GitHub

Ahora que se han hecho todos los cambios deseados en el repositorio local, usted puede empujar (*push*) los cambios a GitHub usando el botón *Push*. Se abrirá una ventana donde se pregunta por su usuario y password de GitHub, luego los cambios serán enviados. Esto dejará su repositorio local en un estado totalmente limpio y sincronizado con el repositorio remoto. Terminado esto, en el historial (en GitHub) se muestran todos los commits, incluyendo los 2 que fueron hechos en GitHub y los 2 que se hicieron en forma local con RStudio.

Ahora puede ver que las etiquetas del repositorio local (`HEAD`) y del remoto (`origin/HEAD`) están apuntando a la misma versión en el historial. De esta forma, si miramos el historial de los commits en GitHub serán iguales a los que tenemos en forma local.

Sobre (buenos) mensajes de confirmación (commit)

Es claro que una buena documentación es crítica para hacer del historial de versiones significativo y útil. Es tentador saltarse la escritura del mensaje asociado al commit o escribir algo por defecto como ‘Actualización!’. Sin embargo es importante escribir mensajes que sean para entender en el futuro que se hizo y por qué. Ademas, los mensajes que se usan en commits son en general más fáciles de entender si usan una convención de verbos activos. Por ejemplo, se puede ver que los mensajes de commit en las capturas de pantallas comienzan siempre con un verbo en pasado (en inglés!) y que explican que fue lo que se cambió.

Si bien muchos de los cambios que aquí se hicieron son simples y se explican por si mismos, para cambios más complejos, es mejor entregar un mensaje completo y auto-contenido. La convención, sin embargo, es tratar de usar mensajes cortos, con una sentencia breve, seguido de una explicación más detallada y racional para el cambio. Esto permite que el nivel de detalles sea legible en el registro de las versiones (version log). No puedo contar el número de veces que he visto los registro de commits de hace 2, 3 o 10 años y agradecido el nivel de diligencia de los colaboradores que se tomaron el tiempo de describir el trabajo que se realizó.

Flujos de trabajo colaborativos y libres de conflictos (relacionados con Git)

Hasta ahora nos hemos enfocado al uso de Git y GitHub para el uso personal, como ya se demostró esto es extremadamente útil. Sin embargo donde git y GitHub brilla es cuando se comparte un repositorio GitHub con colaboradores, de esta forma se puede trabajar en un código, análisis y modelos en forma colaborativa. Cuando se trabaja de esta manera con otros investigadores, es muy importante poner atención al estado del repositorio remoto para evitar potenciales conflictos al combinar el trabajo. Un *merge conflict* ocurre cuando dos colaboradores hacen 2 commits en forma separada donde se ha(n) cambiado(s) la(s) misma(s) línea(s) de código de un archivo. Cuando esto ocurre, git no puede combinar los cambios en forma automática y arroja un error preguntando como resolver el conflicto. Esto no es grave, no es necesario tenerle miedo a los *merge conflicts* ya que son muy fáciles resolver y aquí hay algunas guías geniales. de como hacerlo (por ahora sólo en inglés).

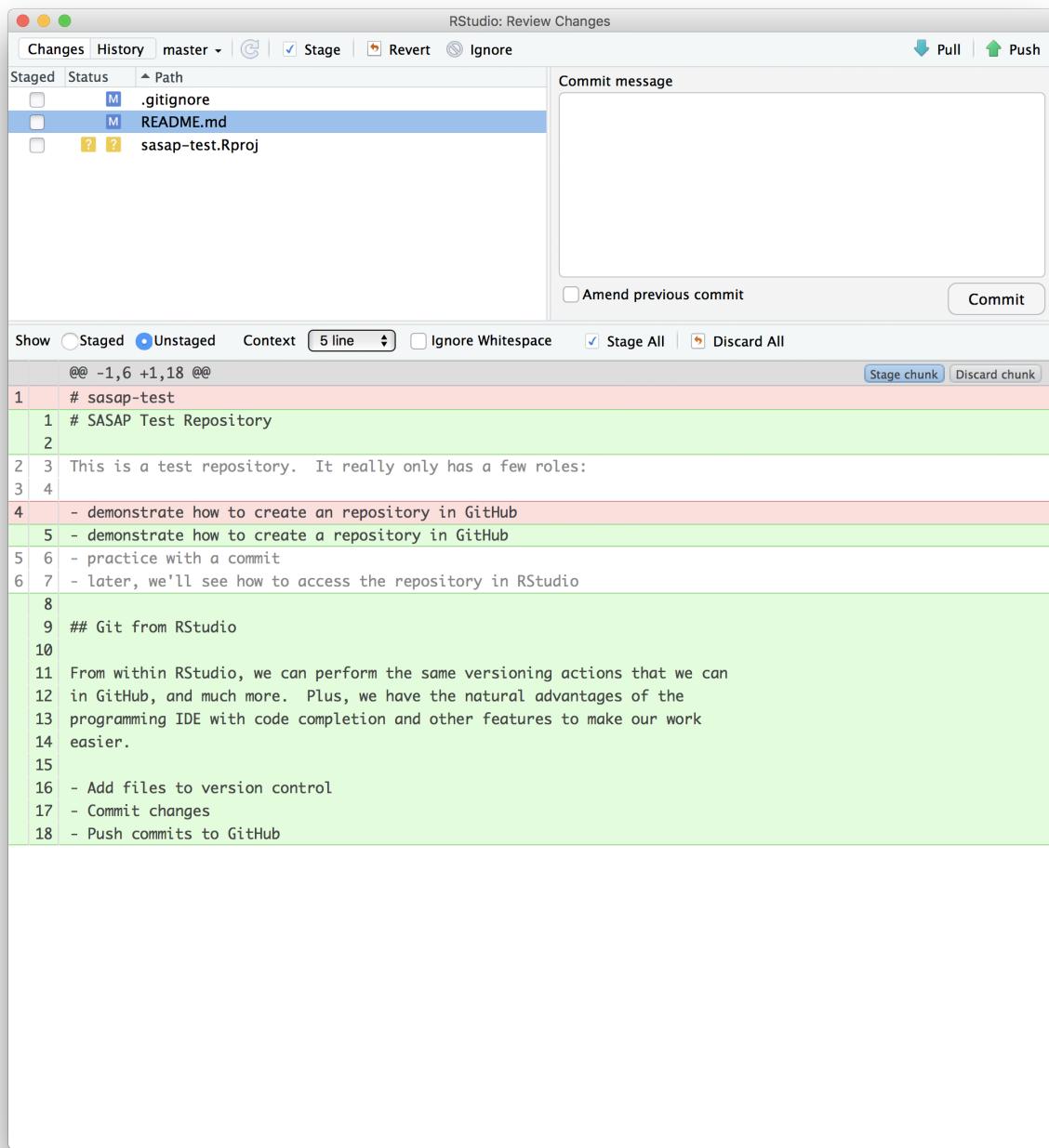


Figure 25: Ventana que presenta las diferencias entre la versión almacenada en el repositorio y los últimos cambios realizados.

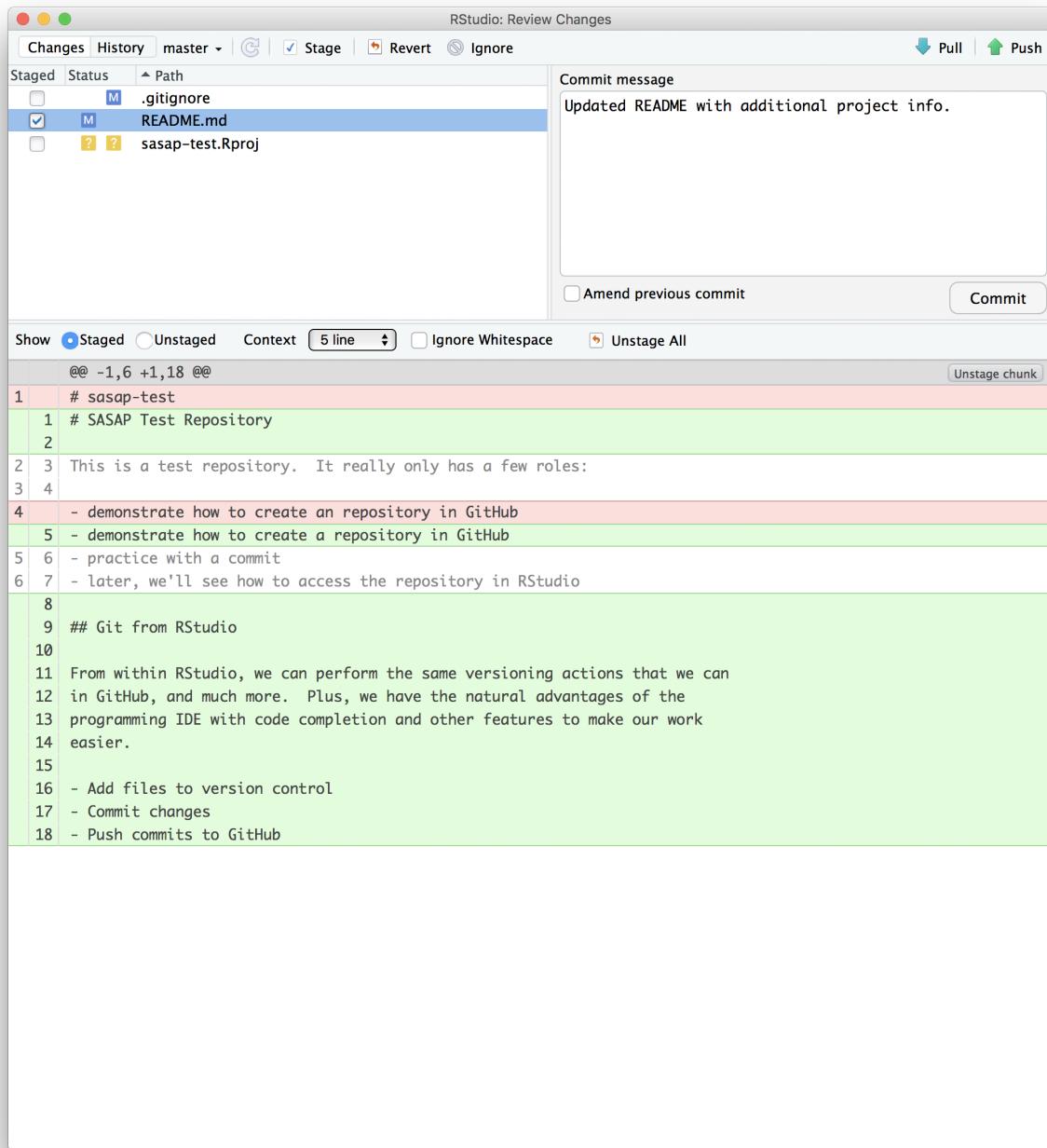


Figure 26: Captura de pantalla con un mensaje describiendo la confirmación (commit) que se realizará.

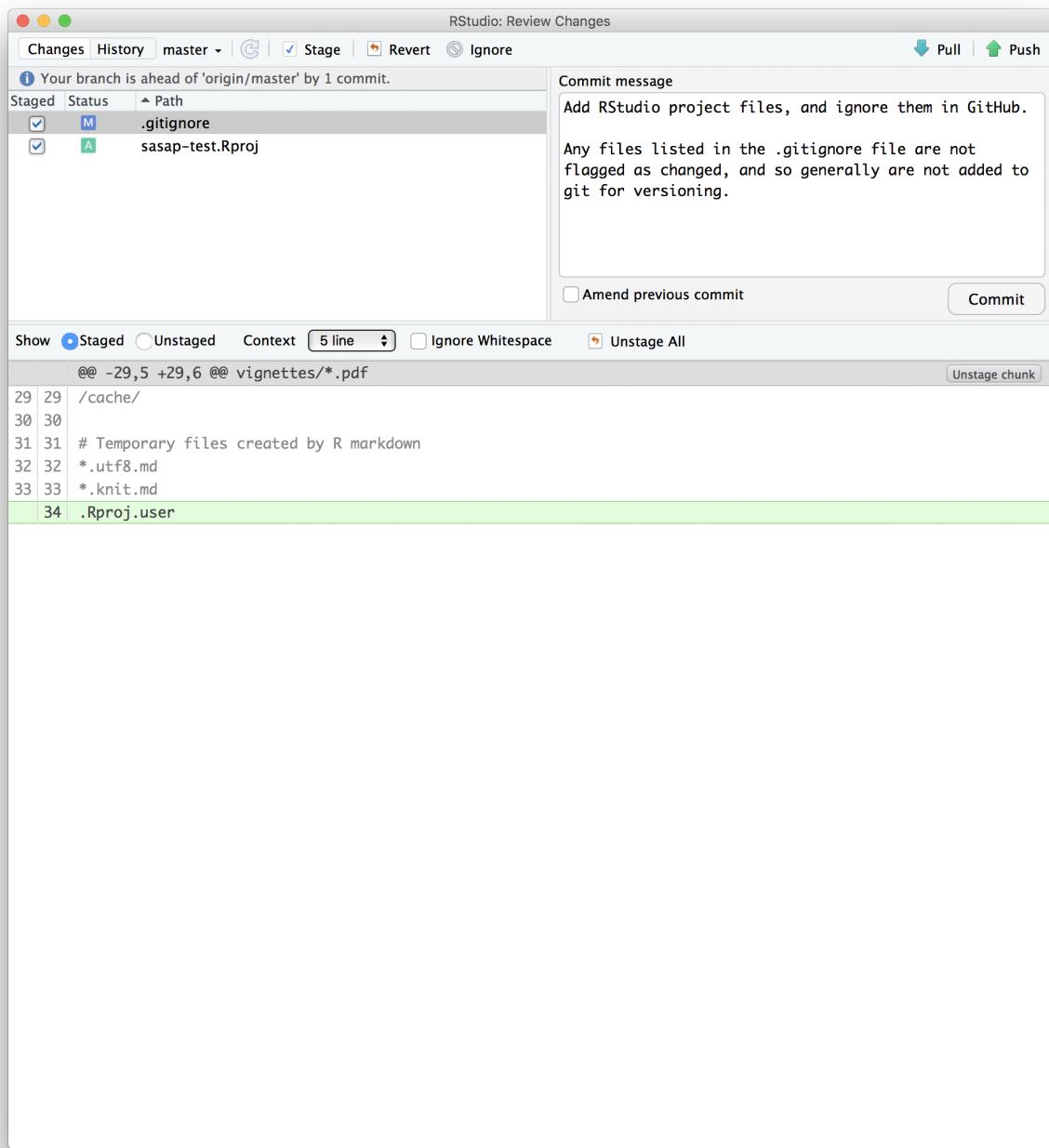


Figure 27: Captura de pantalla con los archivos a confirmar sus cambios y el texto descriptivo de la confirmación.

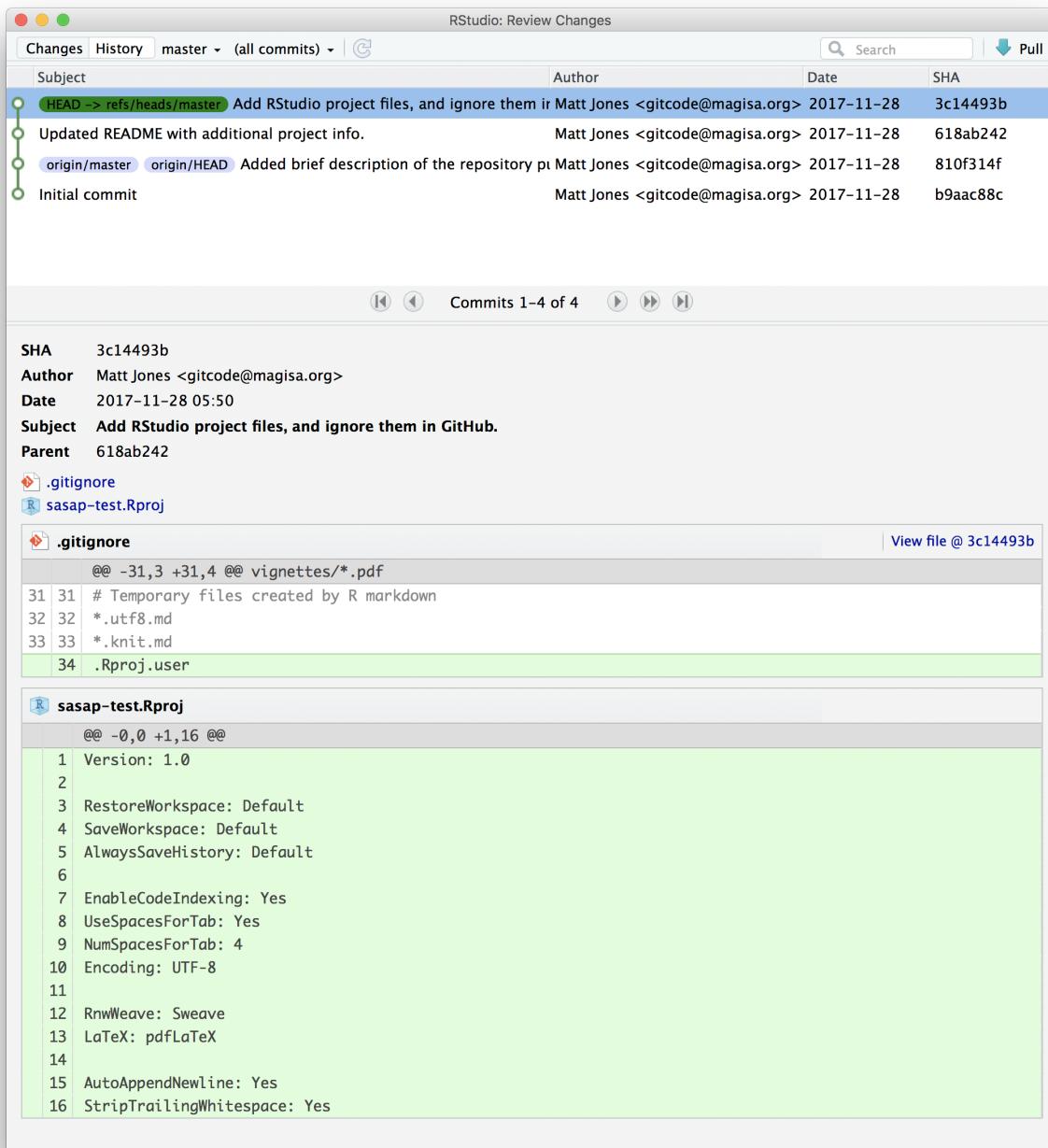


Figure 28: Captura de pantalla con 4 commits.

Subject	Author	Date	SHA
HEAD -> refs/heads/master origin/master origin/HEAD Add RStudio project files, and ignore them	Matt Jones <gitcode@magisa.org>	2017-11-28	3c14493b
Updated README with additional project info.	Matt Jones <gitcode@magisa.org>	2017-11-28	618ab242
Added brief description of the repository purpose.	Matt Jones <gitcode@magisa.org>	2017-11-28	810f314f
Initial commit	Matt Jones <gitcode@magisa.org>	2017-11-28	b9aac88c

Figure 29: Captura de pantalla con los 4 modificaciones confirmadas.

mbjones / sasap-test

Code Issues 0 Pull requests 0 Projects 0 Wiki Insights Settings

Branch: master ▾

Commits on Nov 27, 2017

- Add RStudio project files, and ignore them in GitHub. ... mbjones committed 26 minutes ago
- Updated README with additional project info. mbjones committed 28 minutes ago
- Added brief description of the repository purpose. mbjones committed 3 hours ago
- Initial commit mbjones committed 3 hours ago

Figure 30: Captura de pantalla con el historial de cambios en github.

Dicho esto, es siempre mejor evadir este tipo de conflictos, lo que se pueden minimizar siguiendo las siguientes sugerencias:

- Asegúrese de traer (*pull down*) todos los cambios antes de confirmarlos (*commit*).
 - Asegurese que tiene los cambios más recientes.
 - Pero puede ser que necesite arreglar su código si ocurren conflictos.
- Coordíñese con sus colaboradores con quien va a trabajar.
 - Usted **tiene** que comunicarse para colaborar.

Actividad

Use RStudio para agregar un nuevo archivo al repositorio `sasap-test`, desarrolle una estructura básica y guárdela.

A continuación *stage* y *commit* el archivo en forma local y luego *push it* a GitHub.

Tópicos avanzados

Hay mucho que no hemos visto en este tutorial. Existen tutoriales muy completos que cubren tópicos más avanzados. Algunos de estos temas son:

- Usando git en la linea de comandos.
- Resolviendo conflictos.
- *Branching* y *merging*.
- *Pull requests* versus contribuciones directas por colaboradores.
- Usando `.gitignore` para proteger datos sensativos.
- *GitHub Issues* y por que son útiles.
- y más, mucho más.
- Try Git es un tutorial interactivo muy bueno y completo.
- Software Carpentry Version Control with Git

- Codecademy Learn Git (some paid)

Referencias