

Al ser un proyecto en equipo tenemos que tener en cuenta varias cosas. Las más importantes: vamos a estar trabajando en git para ir integrado lo que programa cada uno, esto significa que vamos a requerir de cuidado y comunicación y por otra parte al estar programando en equipo los distintos puntos a desarrollar en el proyecto se pueden hacer en paralelo, lo que significa que puede que estemos desarrollando cierta función en la que se necesite unos inputs que todavía no estén desarrollados. Por último, alguien tendrá que hacer de master del proyecto, que se encargará de minimizar todos estos conflictos.

Integración en Git

El master tendrá que crear un repositorio en git.

1. Crear nuevo ("new") repositorio en tu github

Create a new repository

A repository contains all project files, including the revision history. Already have a repository you'd like to import?

Import a repository.

Owner: aarana95 / Repository name: GapProject ✓

Great repository names: GapProject is available. Need inspiration? How about...

Description (optional)

☒ Public
Anyone on the internet can see this repository. You choose who can commit to it.

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

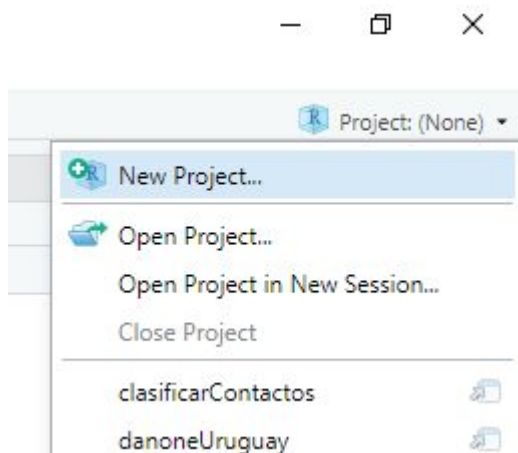
Skip this step if you're importing an existing repository.

☐ Initialize this repository with a README
This will let you immediately clone the repository to your computer.

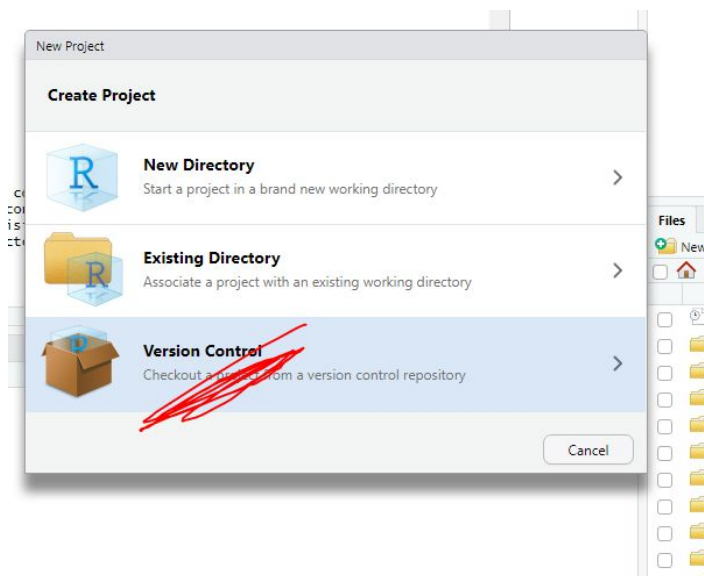
Add .gitignore: None | Add a license: None ⓘ

Create repository

2. Creamos un nuevo proyecto en RStudio (esto lo debemos hacer todos los participantes)

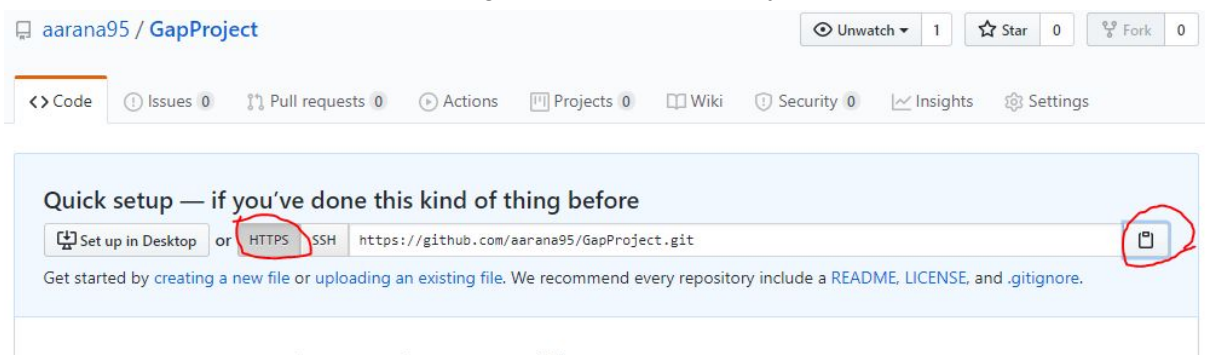


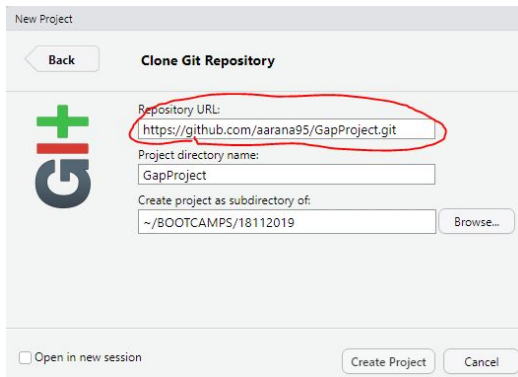
3. Lo creamos desde Version Control



4. Y desde Git

5. Copiamos la URL de nuestro github en el Repository URL de Rstudio



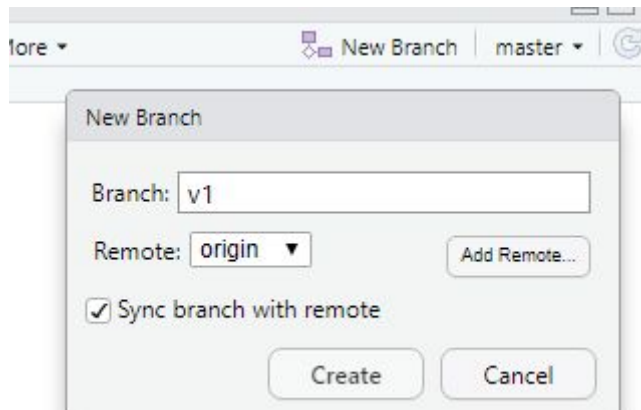


6. Ya tenemos el proyecto descargado. Ahora el master debería crear una rama master. Copiando el código de github en la Terminal de RStudio

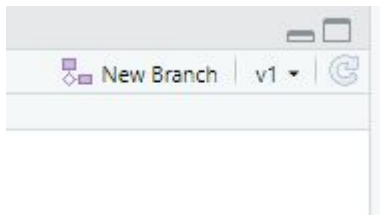
...or create a new repository on the command line

```
echo "# GapProject" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/aarana95/GapProject.git
git push -u origin master
```

7. Ahora si somos el master veremos que estamos en la rama (branch) "master", si no somos el master podremos hacer un "Pull" y nos podremos colocar en "master".
8. Como master vamos a tener que gestionar el git y no es moco de pavo... Como norma general a "master" solo vamos a subir lo que esté testado y vemos que funciona. El desarrollo lo tendremos que ir haciendo en otras ramas



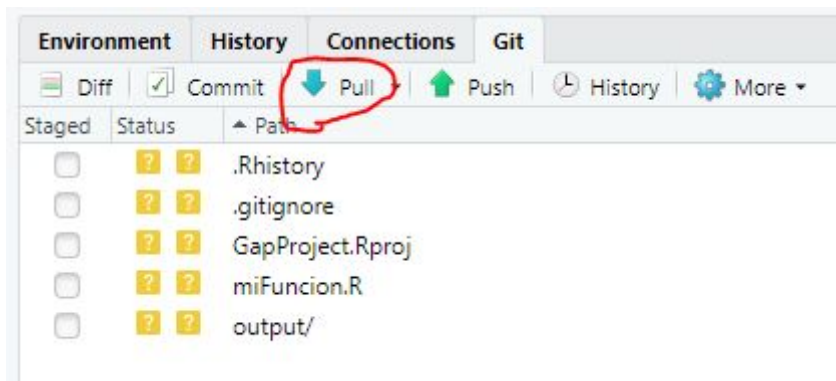
9. Esto significa que para programar nos tendremos que colocar en v1. ESTO ES MUY IMPORTANTE, SI NO ESTAMOS TODOS EN LA MISMA RAMA PUEDEN PASAR DOS COSAS, UNA MALA Y UNA PEOR. LA MALA: GENERAMOS CONFLICTOS EN GIT QUE SON UN DOLOR. LA PEOR: PERDEMOS TODO NUESTRO CÓDIGO.



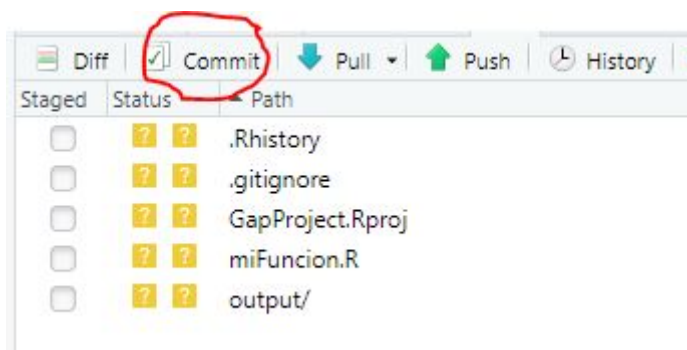
10. Llegará un momento en el que los códigos que vayamos creando habrá que subirlos a master, incluso habrá que crear nuevas versiones (v1.1, v2...). Cuando llegue el momento me llamaís y vamos viéndolo individualmente.

Instrucciones para ir subiendo nuestro código a Git

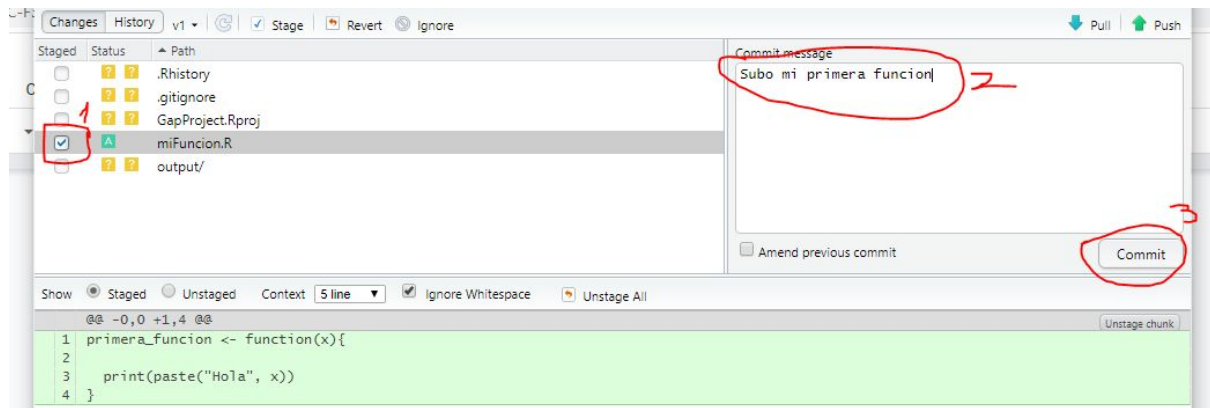
1. Antes de subir cualquier cosa SIEMPRE, hay que hacer un PULL para comprobar que no este subido nada que no tengamos. Si no hacemos pull seguramente generaremos conflictos



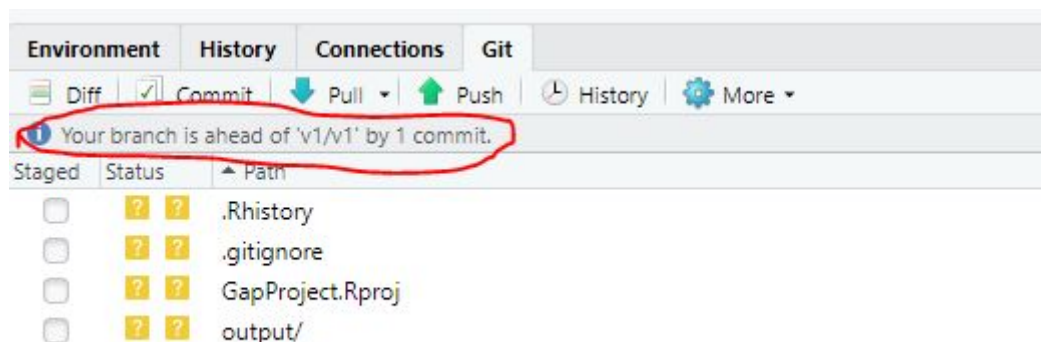
2. Hacemos commit



3. Seleccionamos el archivo que queremos subir, escribimos la descripción de los cambios que vamos a subir y commiteamos



4. Hacemos PUSH. Si vemos este mensaje significa que hemos hecho “commits” que no hemos pusheado



5. Una vez hemos hecho el Push los demás miembros del equipo podrán hacer un Pull y descargarse lo que hemos subido.

Estructura de carpetas

Al comenzar cualquier proyecto en R vamos a construir siempre la misma estructura de carpetas en las que cada una tendrá unos archivos concretos. Las carpetas serán las siguientes:

- **config:** Archivos de configuración, típicamente serán archivos de tipo xml o json.
- **data:** Estarán los datos de entrada para el proyecto, idealmente serán archivos csv, pero los archivos pueden ser de cualquier otro tipo.
- **log:** Aquí irá un archivo .log que se generará automáticamente durante la ejecución del programa. Más adelante hablaremos de esto.
- **miscellaneous:** Aquí irán los archivos que no encajan en ninguna de las demás carpetas, como por ejemplo el script main.R que se utilizará para hacer la llamada de la aplicación, más adelante hablaremos del main.R.
- **output:** En esta carpeta es donde tendremos que guardar cualquier output que genere nuestra aplicación.
- **R:** En esta carpeta irán todos nuestros scripts, archivos .R.

Estructura del código

El proyecto tiene unas partes que están muy definidas:

1. Leemos el archivo de configuración
2. Leemos los datos
3. Pre-procesamos los datos
4. Generamos el modelo
5. Generamos la predicción

Por lo tanto la función principal de nuestra aplicación deberá ser la unión de esas partes, debería parecerse a este código (el código solo tiene desarrollado una parte):

```
funcionPrincipal <- function(directorioProyecto){  
  tryCatch(expr = {  
  
    #Activamos el log  
  
    |  
    #Leemos el archivo de configuracion  
    config <- leerConfig(directorioProyecto)  
  
    #Leemos los datos  
    datos <- leerDatos(config, directorioProyecto)  
  
    #Preprocesamos los datos  
  
    #Generamos el modelo  
  
    #Generamos el output  
  }, error = function(e){  
    logerror("La aplicacion ha petado...", logger = 'log')  
    stop()  
  }, finally = {  
    loginfo("Fin de la ejecucion.", logger = 'log')  
  })  
}
```

En esta fracción de código podemos ver dos de las cosas que tenemos que controlar en el proyecto y que están explicadas más adelante: por un lado los “log” de los que ya hemos hablado antes y por otro lado el control de excepciones que haremos con las funciones tryCatch.

En informática se conoce como código espagueti a los códigos que no hacen uso de funciones o clases y acabas teniendo un script de una barbaridad de líneas de código que son difíciles de manejar. Procuraremos que nuestros códigos tengan una extensión de no más de 50 líneas (idealmente no más de 20), si nuestras funciones tienen más líneas seguramente este código tenga “bloques” y cada uno de estos “bloques” se podrá sustituir por una función. Por lo tanto nuestro código tendrá muchas funciones que habrá que ordenar de una forma lógica. Una opción es tener todas nuestras funciones en un mismo script, pero ya hemos dicho que eso no es buena idea. Otra opción es tener un script para cada función, esto haría que acabásemos con un montón de scripts sin saber de un vistazo cuales son los que son más importantes y cuales son funciones muy concretas que no tienen importancia para el flujo general de la aplicación. Lo que es más recomendable es crear script por bloques. Cada uno de los cinco bloques de los que hemos hablado arriba tendrían su propio script y en el meteríamos la función principal (i.e. leerDatos) y todas las funciones auxiliares que fuesen necesarias para ejecutar la función principal del script. En el caso de encontrarnos con funciones auxiliares que se utilizarán en más de un bloque de código tendríamos que valorar en cuál de los scripts la definiremos o podríamos crear un script de funciones auxiliares.

Archivo de configuración

Los archivos de configuración por norma general son archivos con datos con la estructura clave:valor (como los diccionarios de python). Típicamente se pueden utilizar archivos json o xml. Os voy a explicar los xml pero si algún grupo quiere utilizar un archivo json tiene la libertad de hacerlo.



```
1 <config>
2
3 <datos>
4   <train>paro.csv, bbc.csv</train>
5   <target>pib.csv</target>
6 </datos>
7
8 <testRate>0.2</testRate>
9 <outputFile>Nueva Carpeta</outputFile>
10
11 </config>
```

Las etiquetas son la clave, y lo que ponemos dentro será el valor. En el ejemplo: config es la clave y tiene como valor las claves datos, testRate y outputFile, la clave datos tiene a su vez como valor las claves train y target, la clave target tiene el valor “pib.csv”, etc... Esto es solo un ejemplo para vuestro proyecto tendréis que definir que estructura de config necesitáis. Como config vais a necesitar todos los datos que sean parametrizables, por ejemplo si no tenéis el campo “target” en vuestro archivo de configuración estaríais obligando a el usuario a llamar siempre de la misma forma a la target.

Lectura del archivo de configuración

Leer el archivo de configuración en vuestro código va a tener dos partes: la primera consta de parsear el archivo xml y la segunda que es pasar ese dato parseado a una lista de R. Para hacer estas dos instrucciones vais a necesitar instalar la librería XML, es vuestro trabajo investigar que funciones son y como utilizarlas.

Validaciones del archivo de configuración

Hay tres validaciones que vamos a tener que hacer para validar que nuestro archivo de configuración esta como debería de estar, tened en cuenta que esto es importante, pues una vez este definido el archivo de configuración a lo largo del código siempre lo vamos a llamar de la misma manera. En el ejemplo que hemos visto antes, cuando yo quiera acceder al valor de la target, siempre lo haré de la misma forma: `config$datos$target`, si mi archivo de configuración no es exactamente igual, al acceder a `config$datos$target` me generará un error.

1. La primera validación es de lectura. Tenemos que controlar que no nos de un error al leer nuestro archivo de configuración
2. Si hemos leído bien nuestro archivo de configuración, tenemos que controlar que siempre tenga la misma estructura, esto lo haremos verificando los nombres de las

lista, ejemplo: `name(config$datos) == c("train", "target")` si esa igualdad no es cierta significará que dentro de la etiqueta datos no están las etiquetas train y target (o al menos no en ese orden) por lo que el archivo de configuración no tiene exactamente la forma que le hemos pedido

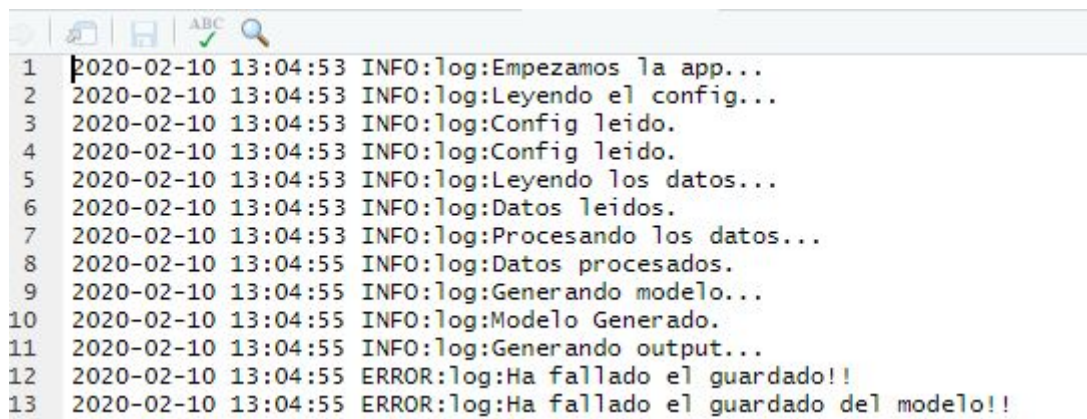
3. Por último si tenemos exactamente la estructura de archivo de configuración que queremos tendremos que validar que los valores que hemos metido tienen sentido: Ej. si en `testRate` le estamos pidiendo el ratio para hacer un train-test split no tiene sentido si le metemos un campo de texto o un número mayor que 1. Así tendríamos que validar cada uno de los campos para que haya únicamente lo que pueda aceptar.

Una vez hemos leído y validado el archivo de configuración ya podremos utilizarlo en el resto del código.

Registro de acontecimientos (log)

Para saber como va ejecutandose nuestra aplicación vamos a crear un archivo que registre todos los hitos importantes, en adelante a este archivo le llamaremos *logfile*. Hay dos motivos principales para tener un logfile:

- Tener un registro de los tiempos de ejecución que tarda cada bloque de nuestro código (lectura de los datos, generar el modelo...). Por ejemplo habría que escribir un mensaje diciendo que vas a leer el config antes de leerlo y otro mensaje de "config leído" al terminar, de esta forma tendremos un registro de cuánto tiempo ha tardado en leer el config y además si vemos que no escribe el último mensaje de "config leído" sabemos que algo raro esta pasando en el config.
- Y ese es el segundo motivo: tener un registro de los errores que pueda dar nuestra aplicación para saber que ha fallado y corregirlo. Siguiendo con el ejemplo del config si en `testRate` tenemos escrito "diez" nos debería devolver un mensaje que diga "Error en el config: testRate tiene que ser un número entre 0 y 1"



```
1 2020-02-10 13:04:53 INFO:log:Empezamos la app...
2 2020-02-10 13:04:53 INFO:log:Leyendo el config...
3 2020-02-10 13:04:53 INFO:log:Config leído.
4 2020-02-10 13:04:53 INFO:log:Config leído.
5 2020-02-10 13:04:53 INFO:log:Leyendo los datos...
6 2020-02-10 13:04:53 INFO:log:Datos leídos.
7 2020-02-10 13:04:53 INFO:log:Procesando los datos...
8 2020-02-10 13:04:55 INFO:log:Datos procesados.
9 2020-02-10 13:04:55 INFO:log:Generando modelo...
10 2020-02-10 13:04:55 INFO:log:Modelo Generado.
11 2020-02-10 13:04:55 INFO:log:Generando output...
12 2020-02-10 13:04:55 ERROR:log:Ha fallado el guardado!!
13 2020-02-10 13:04:55 ERROR:log:Ha fallado el guardado del modelo!!
```

Para generar el logfile utilizaremos la librería **logging**. Para utilizar los logs vamos a necesitar en primer lugar crear un *controlador*, esto es una función en la que le vamos a indicar tres cosas: Que tipo de escritura queremos, como se va a llamar nuestro controlador (o *logger*) y donde va a escribir el log. Esta función se llama **addHandler**. Estos controladores se quedan funcionando a no ser que los desactivemos, si no los desactivamos pueden llevarnos a confusiones, por lo tanto aunque la aplicación falle nosotros vamos a querer desactivar el controlador, más adelante en el control de errores veremos como podemos hacer esto. La función para desactivar el controlador es **removeHandler**.

Por otro lado vamos a querer escribir los mensajes propiamente dichos, estos pueden ser de error (logerror) o de información (loginfo). A las funciones les pasaremos el mensaje que queremos escribir y el nombre del logger que hemos activado.

Control de errores

En cualquier proyecto que queramos poner en producción tiene que haber un control de errores que este estructurado. Esto no significa que la aplicación no pueda fallar, sino que cuando la aplicación falle el usuario sepa que es lo que esta fallando. El mismo ejemplo que hemos puesto antes en el config nos sirve ahora como ejemplo. El usuario tiene libertad para meter cualquier campo en el xml del config, pero si introduce un valor que no es válido para ese campo concreto el programa debe petar, y el usuario debe saber que es lo que ha impedido que se realice la ejecución completa. Cuando queramos detener el flujo de ejecución porque hay algo que vaya a impedir el correcto funcionamiento de la aplicación utilizaremos la función **stop**, en cambio si lo que queremos controlar es errores que a priori no deberían suceder, como por ejemplo la lectura de los datos, debemos meter el código a controlar en una estructura **tryCatch**.

Depurar código (debuggear)

Para depurar nuestro código y saber donde se cometen los errores debemos de *debuggear* el código. Esto es importante porque no todos los errores de nuestro código harán que nuestra aplicación pete. Podemos encontrar tres tipos de errores:

- **Error bloqueante:** Error que impide la ejecución del código. El error detiene la ejecución antes de lo previsto y no permite realizar el flujo completo. Cualquier error de sintaxis en el código generará un error de este tipo.
- **Error crítico:** Este tipo de errores son los más difíciles de detectar y los peores errores que nos podemos encontrar. La aplicación nos devolverá un resultado pero este no concuerda con la lógica del problema. Por ejemplo, si aplicamos una transformación a nuestros datos para hacer el modelo, y para hacer la predicción metemos los datos sin hacer la misma transformación, es posible que el modelo me

devuelva una predicción y que mi aplicación no falle en ningún momento pero la solución será errónea. Para detectar estos errores habrá que depurar el código línea a línea para comprobar que en todo momento nuestros datos estén exactamente como queremos que estén.

- **Error leve:** Este tipo de errores se refiere a errores menores como algún fallo de escritura en los logs o en los outputs.

En R hay dos formas de debuggear: introduciendo la función que queremos debuggear en la función **debug**, si queremos dejar de debuggear una función la meteremos en la función **undebg**. La segunda forma es poniendo **browser()** en el lugar en el que queramos empezar a debuggear y si el flujo de ejecución pasa por ahí se abrirá el entorno de debuggeo (si hemos puesto un browser y la ejecución no se detiene en ese lugar significará que no pasa por ese lugar, esto también nos puede dar pistas del error que estamos buscando).

```
20
21   loginfo("Leyendo el config...", logger = "log")
22   config <- leerConfig(path)
23   loginfo("Config leído.", logger = "log")
24
25
26   loginfo("Leyendo los datos...", logger = "log")
27   datos <- leerDatos(config, path)
28   loginfo("Datos leídos.", logger = "log")
29
30
31   loginfo("Procesando los datos...", logger = "log")
32   splitDatos <- preProcesarDatos(datos)
33   loginfo("Datos procesados.", logger = "log")
34
35
36   loginfo("Generando modelo...", logger = "log")
37   output <- generarModelo(splitDatos, config)
38   loginfo("Modelo Generado.", logger = "log")
39
40
41   loginfo("Generando output...", logger = "log")
42   generarOutput(output, config, path)
43   loginfo("Output generado.", logger = "log")
44
45 }, error = function(e){
46
47   logerror("La aplicacion ha petado...", logger = "log")
48   stop()
49
50 }, finally = {
51
54:7 clasificarContactosApp(path) ↕
```

Console Terminal x

1 BOOTCAMP/PS/44112019/ClasificarContactos/ 5

23 4

Next { } ⏪ ⏩ Continue Stop

```
generarOutput(output, config, path)
loginfo("Output generado.", logger = "log")
}, error = function(e) {
  logerror("La aplicacion ha petado...", logger = "log")
  stop()
}, finally = {
  loginfo("Fin de la ejecucion.", logger = "log")
  removeHandler(writeToFile, logger = "log")
})
Browse[2]> n
```

Aquí tenemos un ejemplo de como funciona la consola en modo debug. La flecha verde y el fondo amarillo nos indica que ahora mismo la ejecución está en la línea 22. Nos encontramos cinco opciones distintas:

1. **Next**: Salta a la siguiente línea de código, en este caso pasaría a la línea 23. Hay que tener en cuenta que si la función falla se saldrá de la ejecución. Para evitar esto podemos ejecutar la parte *leerConfig(path)* en la consola (copiando y pegando el código).
2. El segundo botón sirve para introducirse dentro de la función, en este ejemplo empezariamos a debuggear la función *leerConfig*. R al ser un lenguaje de código abierto podremos debuggear cualquier función aunque sean de una librería de terceros, por lo tanto podemos ver lo que nos falla aunque no lo hayamos programado nosotros.
3. El tercer botón sirve para salir de un bucle, si estamos en un bucle for y no queremos ver las muchas iteraciones que se tienen que ejecutar y con ver las primeras nos basta podemos salir del bucle con este botón.
4. El **Continue** ejecutara todo el código hasta que se encuentre con otro *browser()* o bien hasta que finalice la ejecución.
5. El **stop** finaliza la ejecución.

Testear código (QA)

Para poner una aplicación en producción esta tiene que estar a prueba de bombas. Para eso tenemos que controlar todas las opciones posibles de las partes que son susceptibles de interpretación humana. En este proyecto habría que evaluar todas las combinaciones posibles en el config (por ejemplo dejar el config vacío). Por otro lado hay que evaluar los datos de entrada: ¿Que pasa si hay columnas repetidas?, ¿Si hay una única columna funciona?, ¿Y si hay cero filas?, ¿Si hay muchísimas columnas funciona?, ¿Que ocurre si una fila son todo NAs?, ¿NA, "NA" o "" los trata por igual?

Empaquetar

Para empaquetar un proyecto tenemos que documentarlo, eso se hace poniendo una cabecera a cada una de las funciones:

```
#' @title clasificarContactosApp
#' @description Funcion principal del paquete de clasificarContactos
#'
#' @param path, string
#'
#' @export
#' @import logging
#'
#' @author Ander
clasificarContactosApp <- function(path){
  # ...
}
```

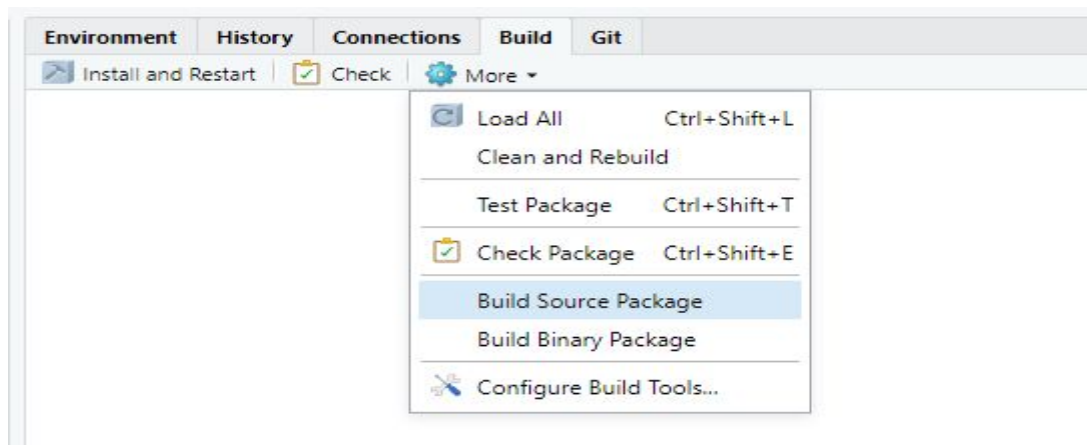
- **title:** nombre de la función
- **description:** Que hace la función
- **param:** parámetro de entrada, hay tantos param como parámetros de entrada tenga la función, en este caso como solo tiene el path, solo hay uno
- **export:** Esta línea solo la pondremos si queremos que a esta función se pueda acceder desde el paquete, en nuestro caso solo llevará export la función principal
- **import:** Librerías que son necesarias para la ejecución de la función, como en param habrá tantos import como librerías fuesen necesarias.
- **return:** si la función devuelve algo habrá que ponerlo en return, misma estructura que param.
- **author:** autor de la función.

Una vez tengamos toda la documentación tenemos que ejecutar en la consola esta función. **roxygen2::roxygenize()** (asegurate de que la ejecutas en la misma ruta que está el proyecto). Esta función te generará por un lado la carpeta **man** en la que está toda la documentación del proyecto y por otro lado un archivo **NAMESPACE** en la que aparecerán las librerías que son necesarias importar y las funciones que vamos a exportar.

Por otro lado tenemos que generar otro archivo que se llamara DESCRIPTION y estará en la raíz del proyecto y tiene que tener la siguiente estructura:

```
Package: clasificarContactos
Title: What The Package Does (one line, title case required)
Version: 0.1
Authors@R: person("Ander", "Arana", email = "first.last@example.com",
                  role = c("aut", "cre"))
Description: What the package does (one paragraph)
Depends: R (>= 3.1.0)
License: What license is it under?
LazyData: true
RoxygenNote: 7.0.2
Encoding: UTF-8
```

Una vez hemos hecho todo esto cerramos Rstudio y lo volvemos a abrir. Nos debería aparecer una pestaña nueva en la ventana del environment: **Build**.



Y le damos a “Build Source Package”. Esto debería crearnos el paquete (un archivo tar.gz) en la carpeta que está por encima de la raíz del proyecto.

Este archivo ya podremos instalarlo y utilizarlo fácilmente desde la pestaña de Packages:

