

Some good practices for research with R

Etienne Bacher
LISER

March 8, 2023

1. Validate our data with {validate}
2. Make our R environment reproducible with {renv}
3. Make our paths reproducible with {here}
4. Keep a clean session

Validate our data with {validate}

Why?

Cleaning data can take hundreds or thousands of lines.

Sometimes we do some mistakes that can have **big consequences**.

Retraction of: Growing up in a Recession

The Review of Economic Studies, rdac085, <https://doi.org/10.1093/restud/rdac085>

Published: 11 January 2023 **Article history** ▼

This is a retraction to: *The Review of Economic Studies*, Volume 81, Issue 2, April 2014, Pages 787–817, <https://doi.org/10.1093/restud/rdt040>



PDF

Split View

“ Cite

Permissions

Share ▼

Issue Section: [Retraction](#)

This is a retraction of: Paola Giuliano, Antonio Spilimbergo, Growing up in a Recession, *The Review of Economic Studies*, Volume 81, Issue 2, April 2014, Pages 787–817, <https://doi.org/10.1093/restud/rdt040>

The authors and editorial team are retracting this article because the original findings cannot be replicated, likely as a result of an inadvertent coding error. While the original codes and data sets are no longer available, new analysis with a markedly similar data set does not support the original results.

Paper published in the JPE (error found in replication led by I4R):

**Note on: Cooperative Property Rights and Development:
Evidence from Land Reform in El Salvador**

Eduardo Montero*

University of Chicago

March 1, 2023

First and foremost, I want to thank Anders Kjelsrud, Andreas Kotsadam, and Ole Rogeberg for their careful and thoughtful replication of [Montero \(2022\)](#). In their replication efforts, documented in [Kjelsrud et al. \(2023\)](#), they uncover a data mistake which affects the results reported in Table 5 and Figure 6 of [Montero \(2022\)](#). This table and figure had presented evidence that land reform cooperatives had lower earnings inequality compared to *haciendas*. Once the data merging error is corrected, these results are no longer valid. Below I discuss in greater detail the data merging error and, to motivate future research, present an alternative, correlational analysis that explores whether collective ownership is associated with lower inequality using more recent data.

What is {validate}?

{validate} is an R package whose goal is to ensure that our code has produced the expected output.

It should be used on the final and on the intermediate datasets (basically anytime we do some big modifications).

How to use {validate}?

1. Define a series of expectations, or *rules*, with `validator()`
2. Pass our dataset through these rules with `confront()`
3. Check that all rules are respected.

Example

Let's take the dataset `gapminder`, modify it a bit, and assume it's our output:

```
1 library(gapminder)
2 library(countrycode)
3
4 gapminder$iso <- countrycode(gapminder$country, "country.name", "iso3c")
5
6 head(gapminder)
```

```
# A tibble: 6 × 7
  country      continent  year lifeExp      pop gdpPercap iso
  <fct>        <fct>    <int>  <dbl>    <int>    <dbl> <chr>
1 Afghanistan Asia      1952   28.8  8425333    779. AFG
2 Afghanistan Asia      1957   30.3  9240934    821. AFG
3 Afghanistan Asia      1962   32.0 10267083    853. AFG
4 Afghanistan Asia      1967   34.0 11537966    836. AFG
5 Afghanistan Asia      1972   36.1 13079460    740. AFG
6 Afghanistan Asia      1977   38.4 14880372    786. AFG
```

1. Define a series of expectations, or **rules**, with `validator()`:

```
1 library(validate)
2
3 rules <- validator(
4   # Ensure that all ISO-3 codes have 3 letters
5   field_length(iso, n = 3),
6
7   # Ensure that there are no duplicated combination of iso-year
8   is_unique(iso, year),
9
10  # Ensure that year doesn't have any missing values
11  !is.na(year)
12 )
```

2. Pass our dataset through these rules with `confront()`:

```
1 x <- confront(gapminder, rules)
2 x <- summary(x)
3
4 x
```

	name	items	passes	fails	nNA	error	warning	expression
1	V1	1704	1704	0	0	FALSE	FALSE	field_length(iso, n = 3)
2	V2	1704	1704	0	0	FALSE	FALSE	is_unique(iso, year)
3	V3	1704	1704	0	0	FALSE	FALSE	!is.na(year)

3. Check that all rules are respected (or generate an error if there's a failing test):

```
1 stopifnot(unique(x$fails) == 0)
```



Writing rules can be tedious, for example if we have a list of variables that should be positive (GDP, population, etc.).

Instead of writing `var1 >= 0, var2 >= 0, ...`, we can use `var_group()`:

```
1 rules <- validator(  
2   positive_vars := var_group(lifeExp, pop, gdpPercap),  
3   positive_vars >= 0  
4 )  
5  
6 x <- confront(gapminder, rules)  
7 x <- summary(x)  
8  
9 head(x)
```

	name	items	passes	fails	nNA	error	warning	expression
1	V2.1	1704	1704	0	0	FALSE	FALSE	lifeExp - 0 >= -1e-08
2	V2.2	1704	1704	0	0	FALSE	FALSE	pop - 0 >= -1e-08
3	V2.3	1704	1704	0	0	FALSE	FALSE	gdpPercap - 0 >= -1e-08

There are a lot of other helpers:

- `in_range()`: useful for e.g percentages
- `field_format()` for regular expressions
- `is_linear_sequence()`: useful to check if there are some gaps in time series
- many others...

See more details in the [The Data Validation Cookbook.](#)

**Make our R environment
reproducible with {renv}**

Packages in R

Packages make our life simpler by not having to reinvent the wheel.

But ***packages evolve!*** Between two versions of a same package:

- functions can be removed or renamed;
- function outputs can change in terms of results or display;
- function arguments can be moved, removed or renamed.

Moreover, packages can disappear if they are not supported anymore.

Personal experience

I did my Master's thesis with R using ~30 packages in total.

Two months later, I couldn't run my code anymore because a package I used to extract some results slightly changed one of its arguments.

→ Two lessons:

1. choose our packages wisely: better to use popular and actively developed packages;
2. use some tools to keep the version of the packages we used.

Packages in R

Even the most used packages in R can change a lot over the years (e.g `tidyverse`).

It is ***our*** responsibility to make sure that our scripts are reproducible. If I take our script 4 years later, I should be able to run it.

Problem: how to deal with evolving packages?

Solution

Take a snapshot of packages version using `{renv}`.

Idea: create a *lockfile* that contains the version of all the packages we used in a project, as well as their dependencies.

When we give the project to someone else, they will be able to restore it with the exact same package versions.

How does it work?

1. Initialize `{renv}` whenever we want with `init()`;
2. Work as usual;
3. Run `snapshot()` from time to time to update the lockfile;
4. If we come back to this project later, or if we share this project, run `restore()` to get the packages as they were when we used them.

Example

Let's take the example of `gapminder` again. We import two packages, `gapminder` and `countrycode`:

```
1 library(gapminder)
2 library(countrycode)
3
4 gapminder$iso <- countrycode(gapminder$country, "country.name", "iso3c")
5
6 head(gapminder)
```

```
# A tibble: 6 × 7
  country      continent  year lifeExp      pop gdpPercap iso
  <fct>        <fct>    <int>  <dbl>    <int>    <dbl> <chr>
1 Afghanistan Asia      1952   28.8  8425333    779. AFG
2 Afghanistan Asia      1957   30.3  9240934    821. AFG
3 Afghanistan Asia      1962   32.0 10267083    853. AFG
4 Afghanistan Asia      1967   34.0 11537966    836. AFG
5 Afghanistan Asia      1972   36.1 13079460    740. AFG
6 Afghanistan Asia      1977   38.4 14880372    786. AFG
```

1. Initialize {renv} whenever we want with `init()`:

```
1 renv::init()
```

```
1 * Initializing project ...
2 * Discovering package dependencies ... Done!
3 * Copying packages into the cache ... Done!
4 The following package(s) will be updated in the lockfile:
5
6 # CRAN =====
7 - R6                [* -> 2.5.1]
8 - base64enc         [* -> 0.1-3]
9 - bslib             [* -> 0.4.2]
10 - cachem           [* -> 1.0.6]
11 - cli              [* -> 3.5.0]
12 - countrycode      [* -> 1.4.0]
13 - digest           [* -> 0.6.31]
14 - ellipsis         [* -> 0.3.2]
15 - evaluate         [* -> 0.19]
16 - fansi            [* -> 1.0.3]
17 - fastmap          [* -> 1.1.0]
```

This will create:

- a file called `renv.lock`
- a folder called `renv`

→ don't touch these files!

2. Work as usual. Let's import another package:

```
1 library(dplyr)
```



```
1 Error in library(dplyr) : there is no package called 'dplyr'
```



Hum... weird, `dplyr` was installed on my laptop.

`{renv}` creates a sort of “local library” in our project, so we need to reinstall `dplyr` first:

```
1 install.packages("dplyr")
```



```
1 library(dplyr)
```



Now that we imported a new package, let's see the status of `{renv}`:

```
1 renv::status()
```



```
1 The following package(s) are installed but not recorded in the lockfile:
2
3   withr           [2.5.0]
4   dplyr           [1.0.10]
5   generics       [0.1.3]
6   tidyselect     [1.2.0]
7
8 Use `renv::snapshot()` to add these packages to your lockfile.
```



3. Run `snapshot()` from time to time to update the lockfile;

```
1 renv::snapshot()
```

```
1 The following package(s) will be updated in the lockfile:
2
3 # CRAN =====
4 - dplyr          [* -> 1.0.10]
5 - generics      [* -> 0.1.3]
6 - tidyselect    [* -> 1.2.0]
7 - withr         [* -> 2.5.0]
8
9 Do you want to proceed? [y/N]: Y
10 * Lockfile written to 'C:/Users/etienne/Desktop/Divers/good-practices/renv.'
```

Good to know

`{renv}` is not a panacea for reproducibility.

If we use some packages that depend on external software (e.g `R Selenium` uses Java), `{renv}` cannot install this software for us.

Learn more about `{renv}` capabilities and limitations on [the package's website](#).

Make our paths reproducible with
{here}

Paths

Absolute path: path that is specific to our computer because it starts *at the root of a computer*.

Ex: "C:/Users/etienne/Desktop/myproject/mydata/WDI"

Relative path: path that is specific to a project because it starts *at the root of the project*.

Ex: "mydata/WDI"

Relative paths in R

Use the package [{here}](#):

- to know the working directory:

```
1 here::here()
```

```
[1] "C:/Users/etienne/Desktop/Divers/good-practices"
```

- to use some data, script, etc.:

```
1 mydata <- read.csv(here::here("data/WDI/gapminder.csv"))
```

Relative paths in R

The only path in my script is “data/WDI/gapminder.csv”.

Therefore, if I give the folder “good-practices” to someone else:

- the output of `here::here()` will change because the location of the folder on the computer changed.
- but my code will still run because the path to the data inside the folder didn't change.

Relative paths in R

More advantages:

- `{here}` will also work if we open the script outside of an RStudio project
- `{here}` will work on all operating systems (e.g no paths problems because of Windows or Mac).

Get more info on `{here}` on [the package's website](#).

Keep a clean session

Remove all objects

Last but not least: how do we ensure our code will run in a fresh session on another laptop?

If you already use `rm(list = ls())` at the beginning of your script...

... you're wrong

Problem

What does `rm(list = ls())` do?

- `rm()`: remove a list of objects from the environment
- `ls()`: list all objects in the environment

So `rm(list = ls())` removes all the objects from the environment: datasets, variables, etc.

What about loaded packages? What about options set with `options()`?

Problem

`rm(list = ls())` does NOT create a fresh R session. Try it yourself:

1. load any package, e.g `dplyr`
2. use it, e.g `filter(iris, Species == "setosa")`
3. run `rm(list = ls())`
4. try again `filter(iris, Species == "setosa")`

This will still work, meaning that the package was not unloaded.

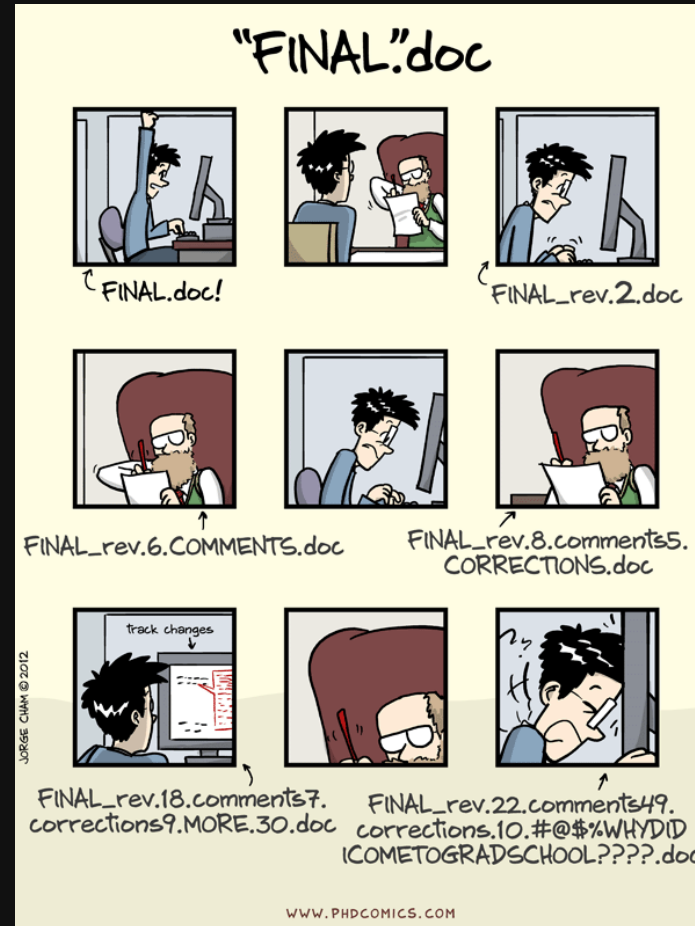
Solution

Instead of using `rm(list = ls())`, you should completely restart the session to be sure your code can run in a fresh session:

- Session > Restart R;
- or Ctrl + Shift + F10;
- or `rstudioapi::restartSession()`.

Bonus: version control

If this is familiar...



... you should (maybe) use version control!

Version control

Most famous version control tool: Git.

Difference between Git and Github:

- Git: core tool
- Github: web interface that makes it much easier to use Git

Version control

Idea: you are able to go back to your project at any point in time.

Workflow:

- put your project on a repository
- write code, write drafts, etc.
- once in a while (at the end of the day, or after a big coding session), **commit** and **push** your changes to the repository
- the repository keeps track of what has changed and allows you to go back to your code at any point in time.

Personal example

Commits on Jan 18, 2023

move the creation of all combinations, clarify a line



etiennebacher committed on Jan 18



76b4a0e



Commits on Jan 17, 2023

fix duplicated names for terrorism, add check for this in validate()



etiennebacher committed on Jan 17



5223aa0



fix some checks



etiennebacher committed on Jan 17



5ba767c



add some checks for own computations of instrument relative to ESS in... ...



etiennebacher committed on Jan 17



f626367



* ESS: remove interviews that happen more than 1.5 years after beginning ...



etiennebacher committed on Jan 17



6af4541



Commits on Jan 16, 2023

wrong code but useful to keep















etiennebacher committed on Jan 16



d574da4



Personal example

Commits on Jan 18, 2023
<div>move the creation of all combinations, clarify a line</div> <div> etiennebacher committed on Jan 18</div> <div> 76b4a0e <></div>
Commits on Jan 17, 2023
<div>fix duplicated names for terrorism, add check for this in validate()</div> <div> etiennebacher committed on Jan 17</div> <div> 5223aa0 <></div>
<div>fix some checks</div> <div> etiennebacher committed on Jan 17</div> <div> 5ba767c <></div>
<div>add some checks for own computations of instrument relative to ESS in...</div> <div> etiennebacher committed on Jan 17</div> <div> f626367 <></div>
<div>* ESS: remove interviews that happen more than 1.5 years after beginning</div> <div> etiennebacher committed on Jan 17</div> <div> 6af4541 <></div>
Commits on Jan 16, 2023
<div>wrong code but useful to keep</div> <div> etiennebacher committed on Jan 16</div> <div> d574da4 <></div>

Commits I made: important to add a useful message (unlike some commits here)

Personal example

The screenshot displays a commit history interface with three sections, each representing a date. Each commit entry includes a description, the committer's name, and a code icon. Red circles highlight these code icons.

Date	Commit Description	Committer	Code Icon
Commits on Jan 18, 2023	move the creation of all combinations, clarify a line	etiennebacher committed on Jan 18	76b4a0e
Commits on Jan 17, 2023	fix duplicated names for terrorism, add check for this in validate()	etiennebacher committed on Jan 17	5223aa0
	fix some checks	etiennebacher committed on Jan 17	5ba767c
	add some checks for own computations of instrument relative to ESS in...	etiennebacher committed on Jan 17	f626367
	* ESS: remove interviews that happen more than 1.5 years after beginning	etiennebacher committed on Jan 17	6af4541
Commits on Jan 16, 2023	wrong code but useful to keep	etiennebacher committed on Jan 16	d574da4

Browse the repository when these commits were made (aka time-travel machine).

Personal example

Version control

Git & Github are also very useful for collaboration (if everyone knows how to use it). It is also possible to link Overleaf and Github.

But not easy to learn and takes time to be efficient (maybe a future training?)

Great resource for Git + Github + R: <https://happygitwithr.com>

Thanks!

Source code for slides and examples:

<https://github.com/etiennebacher/good-practices>

Comments, typos, etc.:

<https://github.com/etiennebacher/good-practices/issues>