onboarding  
reviews,  
that ensure that packages contributed by the community undergo a  
transparent, constructive, non adversarial and open review process, take  
place in the issue tracker of a GitHub repository. Development of the  
packages we onboard also takes place in the open, most often in GitHub  
repositories.

So, how did I collect data?

**A side-note about GitHub**

In the following, I’ll mention repositories. All of them are git  
repositories, which means they’re folders under version control, where  
roughly said all changes are saved via commits and their messages (more  
or less) describing what’s been changed in the commit. Now, on top of  
that these repositories live on GitHub which means they get to enjoy  
some infratructure such as issue trackers, milestones, starring by  
admirers, etc. If that ecosystem is brand new to you

**Package review processes: weaving the threads**

Each package submission is an issue thread in our onboarding repository. The first  
comment in that issue is the submission itself, followed by many  
comments by the editor, reviewers and authors. On top of all the data  
that’s saved there, mostly text data, we have a private  
[Airtable](https://airtable.com/) workspace where we have a table of  
reviewers and their reviews, with direct links to the issue comments  
that are reviews.

**Getting issue threads**

Unsurprisingly, the first step here was to “get issue threads”. What do  
I mean? I wanted a table of all issue threads, one line per comment,  
with columns indicating the time at which something was written, and  
columns digesting the data from the issue itself, e.g. guessing the role  
from the commenter from other information: the first user of the issue  
is the “author”.

I used to use GitHub API V3 and then heard about GitHub API  
V4 which blew my mind. As if I  
weren’t impressed enough by the mere existence of this API

* I discovered the ghql  
  package allows one to interact  
  with such an API and that its docs actually use GitHub API V4 as an  
  example!

I have nothing against GitHub API V3 and  
gh and purrr workflows, but I was  
curious and really enjoyed learning these new tools and writing this  
code. I had written a gh/purrr code for getting the same information  
and it felt clumsier, but it might just be because I wasn’t  
perfectionist enough when writing it! I achieved writing the correct  
GitHub V4 API query to get *just* what I needed I then succeeded  
in transforming the JSON output into a rectangle by reading Carl’s post  
but also by taking advantage of another online explorer, [jq  
play](https://jqplay.org/) where I pasted my output via  
writeClipboard. That’s nearly always the way I learn about query  
tools: using some sort of explorer and then pasting the code into a  
script. When I am more experienced, I can skip the explorer part.

The first function I wrote was one for getting the issue number of the  
last onboarding issue, because then I looped/mapped over all issues.

library("ghql")

library("httr")

library("magrittr")

# function to get number of last issue

get\_last\_issue <- function(){

query = '{

repository(owner: "ropensci", name: "onboarding") {

issues(last: 1) {

edges{

node{

number

}

}

}

}

}'

token <- Sys.getenv("GITHUB\_GRAPHQL\_TOKEN")

cli <- GraphqlClient$new(

url = "https://api.github.com/graphql",

headers = add\_headers(Authorization = paste0("Bearer ", token))

)

## define query

### creat a query class first

qry <- Query$new()

qry$query('issues', query)

last\_issue <-cli$exec(qry$queries$issues)

last\_issue %>%

jqr::jq('.data.repository.issues.edges[].node.number') %>%

as.numeric()

}

get\_last\_issue()

## [1] 201

Then I wrote a function for getting all the precious info I needed from  
an issue thread. At the time it lived on its own in an R script, now  
it’s gotten included in ghrecipes  
package as  
[get\_issue\_thread](https://github.com/ropenscilabs/ghrecipes/blob/master/R/get_issue_thread.R)  
so you can check out the code there, along with other useful recipes for  
analyzing GitHub data.

Then I launched this code to get all data! It was very satisfying.

Library(ghrecipes)

#get all threads

issues <- purrr::map\_df(1:get\_last\_issue(), get\_issue\_thread)

# for the one(s) with 101 comments get the 100 last comments

long\_issues <- issues %>%

dplyr::count(issue) %>%

dplyr::filter(n == 101) %>%

dplyr::pull(issue)

issues2 <- purrr::map\_df(long\_issues, get\_issue\_thread, first = FALSE)

all\_issues <- dplyr::bind\_rows(issues, issues2)

all\_issues <- unique(all\_issues)

readr::write\_csv(all\_issues, "data/all\_threads\_v4.csv")

**Digesting them and complementing them with Airtable data**

In the previous step we got a rectangle of all threads, with information  
from the first issue comment (such as labels) distributed to all the  
comments of the threads.

issues <- readr::read\_csv("data/all\_threads\_v4.csv")

issues <- janitor::clean\_names(issues)

issues <- dplyr::rename(issues, user = author)

issues <- dplyr::select(issues, - dplyr::contains("topic"))

issues %>%

head() %>%

dplyr::select(- body) %>%

knitr::kable()

| **title** | **author\_association** | **assignee** | **created\_at** | **closed\_at** | **user** | **comment\_url** | **package** | **pulled** | **issue** | **meta** | **x6\_approved** | **out\_of\_scope** | **x4\_review\_s\_in\_awaiting\_changes** | **x0\_presubmission** | **question** | **x3\_reviewer\_s\_assigned** | **holding** | **legacy** | **x1\_editor\_checks** | **x5\_awaiting\_reviewer\_s\_response** | **x2\_seeking\_reviewer\_s** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| rrlite | OWNER | sckott | 2015-03-10 23:22:45 | 2015-03-31 00:16:28 | richfitz | NA | TRUE | TRUE | 1 | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Now we need a few steps more:

* transforming NA into FALSE for variables corresponding to labels,
* getting the package name from Airtable since the titles of issues  
  are not uniformly formatted,
* knowing which comment is a review,
* deducing the role of the user writing the comment  
  (author/editor/reviewer/community manager/other).

Below binary variables are transformed and only rows corresponding to  
approved packages are kept.

# labels

replace\_1 <- function(x){

!is.na(x[1])

}

# binary variables

ncol\_issues <- ncol(issues)

issues <- dplyr::group\_by(issues, issue) %>%

dplyr::arrange(created\_at) %>%

dplyr::mutate\_at(9:(ncol\_issues-1), replace\_1) %>%

dplyr::ungroup()

# keep only issues that are finished

issues <- dplyr::filter(issues, package, !x0\_presubmission,

!out\_of\_scope, !legacy,

!x1\_editor\_checks, x6\_approved)

issues <- dplyr::select(issues, - dplyr::starts\_with("x"),

- package, - out\_of\_scope, - legacy,

- meta, - holding, - pulled, - question)

Then, thanks to the airtabler package we can add the name of the  
package, and identify review comments.

# airtable data

airtable <- airtabler::airtable("appZIB8hgtvjoV99D", "Reviews")

airtable <- airtable$Reviews$select\_all()

airtable <- dplyr::mutate(airtable,

issue = as.numeric(stringr::str\_replace(onboarding\_url,

".\*issues\\/", "")))

# we get the name of the package

# and we know which comments are reviews

reviews <- dplyr::select(airtable, review\_url, issue, package) %>%

dplyr::mutate(is\_review = TRUE)

issues <- dplyr::left\_join(issues, reviews, by = c("issue", "comment\_url" = "review\_url"))

issues <- dplyr::mutate(issues, is\_review = !is.na(is\_review))

Finally, the non elegant code below attributes a role to each user  
(commenter is its more precise version that differentiates reviewer 1  
from reviewer 2). I could have used dplyr case\_when.

# non elegant code to guess role

issues <- dplyr::group\_by(issues, issue)

issues <- dplyr::arrange(issues, created\_at)

issues <- dplyr::mutate(issues, author = user[1])

issues <- dplyr::mutate(issues, package = unique(package[!is.na(package)]))

issues <- dplyr::mutate(issues, assignee = assignee[1])

issues <- dplyr::mutate(issues, reviewer1 = ifelse(!is.na(user[is\_review][1]), user[is\_review][1], ""))

issues <- dplyr::mutate(issues, reviewer2 = ifelse(!is.na(user[is\_review][2]), user[is\_review][2], ""))

issues <- dplyr::mutate(issues, reviewer3 = ifelse(!is.na(user[is\_review][3]), user[is\_review][3], ""))

issues <- dplyr::ungroup(issues)

issues <- dplyr::group\_by(issues, issue, created\_at, user)

# regexp because in at least 1 case assignee = 2 names glued together

issues <- dplyr::mutate(issues, commenter = ifelse(stringr::str\_detect(assignee, user), "editor", "other"))

issues <- dplyr::mutate(issues, commenter = ifelse(user == author, "author", commenter))

issues <- dplyr::mutate(issues, commenter = ifelse(user == reviewer1, "reviewer1", commenter))

issues <- dplyr::mutate(issues, commenter = ifelse(user == reviewer2, "reviewer2", commenter))

issues <- dplyr::mutate(issues, commenter = ifelse(user == reviewer3, "reviewer3", commenter))

issues <- dplyr::mutate(issues, commenter = ifelse(user == "stefaniebutland", "community\_manager", commenter))

issues <- dplyr::ungroup(issues)

issues <- dplyr::mutate(issues, role = commenter,

role = ifelse(stringr::str\_detect(role, "reviewer"),

"reviewer", role))

issues <- dplyr::select(issues, - author, - reviewer1, - reviewer2, - reviewer3, - assignee,

- author\_association, - comment\_url)

readr::write\_csv(issues, "data/clean\_data.csv")

The role “other” corresponds to anyone chiming in, while the community  
manager role is planning blog posts with the package author.

Here is the final table. I unselect “body” because formatting in the  
text could break the output here, but I do have the text corresponding  
to each comment.

issues %>%

dplyr::select(- body) %>%

head() %>%

knitr::kable()

| **title** | **created\_at** | **closed\_at** | **user** | **issue** | **package** | **is\_review** | **commenter** | **role** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| rrlite | 2015-03-31 00:25:14 | 2015-04-13 23:26:38 | richfitz | 6 | rrlite | FALSE | author | author |
| rrlite | 2015-04-01 17:30:51 | 2015-04-13 23:26:38 | sckott | 6 | rrlite | FALSE | editor | editor |
| rrlite | 2015-04-01 17:36:03 | 2015-04-13 23:26:38 | karthik | 6 | rrlite | FALSE | other | other |
| rrlite | 2015-04-02 03:36:09 | 2015-04-13 23:26:38 | jeroen | 6 | rrlite | FALSE | reviewer2 | reviewer |
| rrlite | 2015-04-02 03:50:43 | 2015-04-13 23:26:38 | gaborcsardi | 6 | rrlite | FALSE | other | other |
| rrlite | 2015-04-02 03:53:57 | 2015-04-13 23:26:38 | richfitz | 6 | rrlite | FALSE | author | author |

There are 2521 comments, corresponding to 70 onboarded packages.

**Submitted repositories: down to a few metrics**

As mentioned earlier, onboarded packages are most often developped on  
GitHub. In any case, their being on GitHub  
means it’s possible to get their history to have a glimpse at work  
represented by onboarding!

**Getting all onboarded repositories**

Using rOpenSci git2r package I  
cloned all onboarded repositories in a “repos” folder. Since I didn’t  
know which package was in ropensci or ropenscilabs, I tried both.

Library(git2r)

airtable <- airtabler::airtable("appZIB8hgtvjoV99D", "Reviews")

airtable <- airtable$Reviews$select\_all()

safe\_clone <- purrr::safely(git2r::clone)

# github link either ropensci or ropenscilabs

clone\_repo <- function(package\_name){

print(package\_name)

url <- paste0("https://github.com/ropensci/", package\_name, ".git")

local\_path <- paste0(getwd(), "/repos/", package\_name)

clone\_from\_ropensci <- safe\_clone(url = url, local\_path = local\_path,

progress = FALSE)

if(is.null(clone\_from\_ropensci$result)){

url <- paste0("https://github.com/ropenscilabs/", package\_name, ".git")

clone\_from\_ropenscilabs <- safe\_clone(url = url, local\_path = local\_path,

progress = FALSE)

if(is.null(clone\_from\_ropenscilabs$result)){

message("OUILLE")

}

}

}

pkgs <- unique(airtable$package)

pkgs <- pkgs[!pkgs %in% fs::dir\_ls()]

pkgs <- pkgs[pkgs != "rrricanes"]

purrr::walk(pkgs, clone\_repo)

I didn’t clone “rrricanes” because it was too big!

**Getting commits reports**

I then got the commit logs of each repo for various reasons:

* commits themselves show how much code and documentation editing was  
  done during review
* I wanted to be able to git reset hard the repo at its state at  
  submission, for which I needed the commit logs.

I used the gitsum  
package to get commit  
logs because its dedicated high-level functions made it easier than with  
git2r.

library("magrittr")

library(gitsum)

get\_report <- function(package\_name){

message(package\_name)

local\_path <- paste0(getwd(), "/repos/", package\_name)

if(length(fs::dir\_ls(local\_path)) != 0){

gitsum::init\_gitsum(local\_path, over\_write = TRUE)

report <- gitsum::parse\_log\_detailed(local\_path)

report <- dplyr::select(report, - nested)

report$package <- package\_name

if(!"datetime" %in% names(report)){

report <- dplyr::mutate(report,

hour = as.numeric(stringr::str\_sub(timezone, 1, 3)),

minute = as.numeric(stringr::str\_sub(timezone, 4, 5)),

datetime = date + lubridate::hours(-1 \* hour) + lubridate::minutes(-1 \* minute))

report <- dplyr::select(report, - hour, - minute, - timezone)

}

report <- dplyr::select(report, - date)

return(report)

}else{

return(NULL)

}

}

packages <- fs::dir\_ls("repos")

packages <- stringr::str\_replace\_all(packages, "repos\\/", "")

purrr::map\_df(packages, get\_report) %>%

readr::write\_csv("output/gitsum\_reports.csv")

**Getting repositories as at submission**

Crossing information from the issue threads and from commit logs, I  
could find the latest commit before submission and create a copy of each  
repo before resetting it at this state. This is the closest to a  
[Time-Turner](http://harrypotter.wikia.com/wiki/Time-Turner) that I  
have!

library("magrittr")

# get issues opening datetime

issues <- readr::read\_csv("data/clean\_data.csv")

issues <- dplyr::group\_by(issues, package)

issues <- dplyr::summarise(issues, opened = min(created\_at))

# now for each package keep only commits before that

commits <- readr::read\_csv("output/gitsum\_reports.csv")

commits <- dplyr::left\_join(commits, issues, by = "package")

commits <- dplyr::group\_by(commits, package)

commits <- dplyr::filter(commits, datetime <= opened)

# and from them keep the latest one,

# that's the latest commit before submission!

commits <- dplyr::filter(commits, datetime == max(datetime), !is\_merge)

commits <- dplyr::summarize(commits, hash = hash[1])

# small helper function

get\_sha <- function(commit){

commit@sha

}

set\_archive <- function(package\_name, commit){

message(package\_name)

# copy the entire repo to another location

local\_path <- paste0(getwd(), "/repos/", package\_name)

local\_path\_archive <- paste0(getwd(), "/repos\_at\_submission/", package\_name)

fs::dir\_copy(local\_path, local\_path\_archive)

# get all commits -- it's fast which is why I don't use gitsum report here

commits <- git2r::commits(git2r::repository(local\_path\_archive))

# get their sha

sha <- purrr::map\_chr(commits, get\_sha)

# all of this to extract the commit with the sha of the latest commit before submission

# in other words the latest commit before submission

commit <- commits[sha == commit][[1]]

# do a hard reset at that commit

git2r::reset(commit, reset\_type = "hard")

}

purrr::walk2(commits$package, commits$hash, set\_archive)

**Outlook: getting even more data? Or analyzing this dataset**

There’s more data to be collected or prepared! From GitHub issues,  
using GitHub  
archive one could  
get the labelling history: when did an issue go from “editor-checks” to  
“seeking-reviewers” for instance? It’d help characterize the usual speed  
of the process. One could also try to investigate the formal and less  
formal links between the onboarded repository and the review: did  
commits and issues mention the onboarding review (with words), or even  
actually put a link to it? Are actors in the process little or very  
active on GitHub for other activities, e.g. could we see that some  
reviewers create or revive their GitHub account especially for  
reviewing?

Rather than enlarging my current dataset, I’ll present its analysis in  
two further blog posts answering the questions “How much work is  
rOpenSci onboarding?” and “How to characterize the social weather of  
rOpenSci onboarding?”.