library(tidyverse) library(rlang) library(tidyxl) library(brotools) mtcars\_plot <- mtcars %>%

group\_nest(am) %>% #shortcut for group\_by(am) %>% nest() mutate(plots = map2(.y = am, .x = data, ~{ggplot(data = .x) +

geom\_smooth(aes(y = mpg, x = hp), colour =

"#82518c") +

ggtitle(paste0("Miles per gallon as a function

of horse power for am = ", .y)) +

theme\_blog()})

This creates a new data frame that looks like this:

mtcars\_plot

## # A tibble: 2 x 3

## am data plots ## >

|  |  |  |  |
| --- | --- | --- | --- |
| ## | 1 | 0 | [19 × 10] |
| ## | 2 | 1 | [13 × 10] |

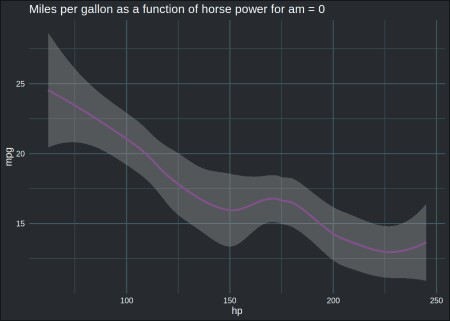
In three lines of code, I grouped the mtcars dataframe by the variable am and then created two plots, which are contained in a new column called plots. If you’re unfamiliar with R, it is quite likely that you’ve never seen anything like this. If you have experience with functional programming languages though, you might recognize what’s going on. Essentially, map2() *loops* over two variables, am and data (this variable is not in the original data frame, but gets created as a result of the group\_nest(am) call) and applies a function, in this case a call to ggplot(), to generate two plots…

Let’s take a look at the plots:

mtcars\_plot %>% pull(plots

## [[1]]

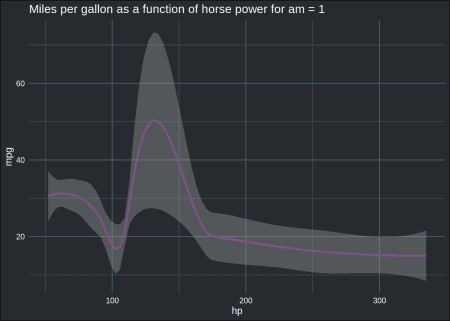
## `geom\_smooth()` using method = 'loess' and formula 'y ~ x'



##

## [[2]]

## `geom\_smooth()` using method = 'loess' and formula 'y ~ x'



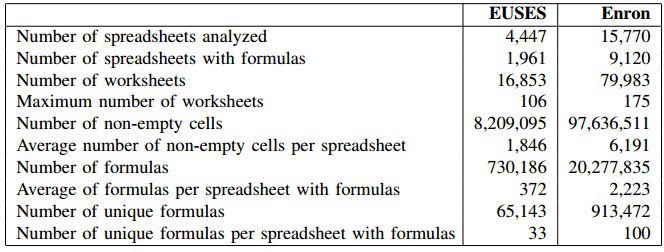
The advantage of this workflow is that you don’t have to think much about anything -once you understand how it works-. The alternative would be two create two separate data frames, and create two separate plots. That’s a totally valid solution, unless you need to create hundreds of plots. With the workflow above, it doesn’t matter if the am variable has 2 or 2000 levels. The code would look exactly the same.

This workflow is very flexible. You can even use this approach to read in, and analyze, many, many files. As many as, for instance, 15931 Excel files from an American oil company that went bust in the early 2000’s, Enron.

# The Enron Corpus

I won’t go into much detail about the Enron Corpus, but to make a long story short: Big evil American oil company went bust, company emails got released for research purposes after being purchased for 10000USD by a computer scientist, and many of these emails had Excel spreadsheets attached to them. Other computer scientist released spreadsheets for research purposes.

On Felienne Hermans’ blog post, you can spot the following table:



I’m going to show how this table could be replicated using R and the mutate()–map() workflow above. First, let’s load one single spreadsheet with {tidyxl} and get some of the code ready that we will need.

Let’s get all the paths to all the files in a vector:

list\_paths <- list.files(path = "~/six\_to/spreadsheets",

pattern = ".xlsx" full.names = TRUE

Let’s work with the first one. Let’s read it in with {tidyxl}:

(example\_xlsx <- xlsx\_cells(list\_paths[1])) ## # A tibble: 19,859 x 21

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ##  ## | sheet | address | row | col | is\_blank | data\_type | error | logical | numeric |
| ## | 1 Preschedule | A1 | 1 | 1 | FALSE | date | NA |  | NA |
| ## | 2 Preschedule | B1 | 1 | 2 | TRUE | blank | NA |  | NA |
| ## | 3 Preschedule | C1 | 1 | 3 | TRUE | blank | NA |  | NA |
| ## | 4 Preschedule | D1 | 1 | 4 | TRUE | blank | NA |  | NA |
| ## | 5 Preschedule | E1 | 1 | 5 | TRUE | blank | NA |  | NA |
| ## | 6 Preschedule | F1 | 1 | 6 | TRUE | blank | NA |  | NA |
| ## | 7 Preschedule | G1 | 1 | 7 | TRUE | blank | NA |  | NA |
| ## | 8 Preschedule | H1 | 1 | 8 | TRUE | blank | NA |  | NA |
| ## | 9 Preschedule | I1 | 1 | 9 | TRUE | blank | NA |  | NA |
| ## | 10 Preschedule | J1 | 1 | 10 | TRUE | blank | NA |  | NA |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ## | # | … | with 19,849 more rows, and 12 more variables: date , character | , |
| ## | # |  | character\_formatted , formula , is\_array , |  |
| ## | # |  | formula\_ref , formula\_group , comment , height , |  |
| ## | # |  | width , style\_format , local\_format\_id |  |

The beauty of {tidyxl} is that it can read in a very complex and ugly Excel file without any issues. Each cell of the spreadsheet is going to be one row of the data set, the contents of all cells is now easily accessible. Let’s see how many sheets are in there:

example\_xlsx %>%

summarise(n\_sheets = n\_distinct(sheet) ## # A tibble: 1 x 1

## n\_sheets ##

## 1 11

11… that’s already quite a lot. How many formulas are there per sheet?

example\_xlsx %>%

mutate(is\_formula = is.na(formula)) %>% group\_by(sheet) %>%

summarise(n\_formula = sum(is\_formula)) %>% arrange(desc(n\_formula)

## # A tibble: 11 x 2

## sheet n\_formula ##

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ## | 1 | Preschedule |  |  | 2651 |
| ## | 2 | Deals |  |  | 324 |
| ## | 3 | Economics |  |  | 192 |
| ## | 4 | Balancing |  |  | 97 |
| ## | 5 | Fuel |  |  | 70 |
| ## | 6 | Comp |  |  | 0 |
| ## | 7 | EPEData |  |  | 0 |
| ## | 8 | HeatRate |  |  | 0 |
| ## | 9 | spin reserve | log | sheet | 0 |
| ## | 10 | Top |  |  | 0 |

## 11 Unit Summary 0

There’s a sheet in there with 2651 formulas. This is insane. Anyways, as you can see, {tidyxl} makes analyzing what’s inside such Excel files quite simple. Let’s now create functions that will compute what we need. I won’t recreate everything from the table, but you’ll very quickly get the idea. Let’s start with a function to count spreadsheets that contain at least one formula:

at\_least\_one\_formula <- function(x){ (any( is.na(x$formula)))

}

Let’s get the number of worksheets:

n\_sheets <- function(x){

x %>

summarise(n\_sheets = n\_distinct(sheet)) %>% pull(n\_sheets

}

And how many formulas are contained in a spreadsheet:

n\_formulas <- function(x){

x %>

mutate(is\_formula = is.na(formula)) %>% summarise(n\_formula = sum(is\_formula)) %>% pull(n\_formula

}

Let’s stop here. We could of course continue adding functions, but that’s enough to illustrate what’s coming. Let’s just define one last function. This function will call all three functions defined above, and return the result in a dataframe. You’ll see why soon enough:

get\_stats <- function(x){

tribble(~has\_formula, ~n\_sheets, ~n\_formulas at\_least\_one\_formula(x), n\_sheets(x), n\_formulas(x)

}

Let’s try it out on our single spreadsheet:

get\_stats(example\_xlsx) ## # A tibble: 1 x 3

## has\_formula n\_sheets n\_formulas ##

## 1 TRUE 11 3334

Neat.

Now, let’s see how we can apply these function to 15k+ Excel spreadsheets.

# No loops ever allowed

10 years ago, I was confronted to a similar problem. I had a pretty huge amount of files on a computer that I

needed to analyze for a chapter of my Phd thesis. The way I solved this issue was by writing a loop that looked horrible and did what I needed on each file. It did the job, but it did not look good, and was a nightmare whenever I needed to modify it, which I needed to do often. I had to think about a structure to hold the results; it was a nested list with I think 4 or 5 levels, and I had to keep track of the dimensions in my head to make sure I was writing the right result in the right spot. It wasn’t pleasant. Until this week, I thought that such a loop was the only real solution to such a problem.

But a comment on one of my youtube video changed this:



The comment was made on this video in which I create a data set like in the introduction to this blog post, but instead of having 2 groups (and thus 2 datasets), I had 100. Now, in the video this wasn’t an issue, but what if instead of having 100 datasets, I had 15k+? And what if these datasets were quite huge? For example, the largest spreadsheet in the Enron Corpus is 40MiB. Loading it with {tidyxl} returns a tibble with 17 million rows, and needs 2GiB of RAM in a clean R session. If you want to read in all the 15k+, you’re simply going to run out of memory even before you could analyze anything. As I’ve written above, the solution would be to loop over each file, do whatever I need done, and save the results in some kind of structure (very likely some complex nested list). Or is it the only solution? Turns out that I tried some things out and found a solution that does not require changing my beloved mutate()-map() workflow.

Let’s first start by putting the paths in a data frame:

(enron <- enframe(list\_paths, name = NULL, value = "paths")) ## # A tibble: 15,871 x 1

## paths ##

|  |  |  |
| --- | --- | --- |
| ## | 1 | /home/cbrunos/six\_to/spreadsheets/albert\_meyers 1 1-25act.xlsx |
| ## | 2 | /home/cbrunos/six\_to/spreadsheets/albert\_meyers 2 1-29act.xlsx |
| ## | 3 | /home/cbrunos/six\_to/spreadsheets/andrea\_ring 10 ENRONGAS(1200).xlsx |
| ## | 4 | /home/cbrunos/six\_to/spreadsheets/andrea\_ring 11 ENRONGAS(0101).xlsx |
| ## | 5 | /home/cbrunos/six\_to/spreadsheets/andrea\_ring 12 ENRONGAS(1200).xlsx |
| ## | 6 | /home/cbrunos/six\_to/spreadsheets/andrea\_ring 13 Trader & Products |

5-15-01…

## 7 /home/cbrunos/six\_to/spreadsheets/andrea\_ring 14 Trader & Products 5-16-01…

## 8 /home/cbrunos/six\_to/spreadsheets/andrea\_ring 15 IFERCnov.xlsx

## 9 /home/cbrunos/six\_to/spreadsheets/andrea\_ring 16 ifercdec.xlsx

## 10 /home/cbrunos/six\_to/spreadsheets/andrea\_ring 17 IFERCJan.xlsx ## # … with 15,861 more rows

For the purposes of this blog post, let’s limit ourselves to 30 spreadsheets. This won’t impact how the code is going to work, nor memory usage. It’s just that I won’t my post to compile quickly while I’m writing:

(enron <- head(enron, 30)) ## # A tibble: 30 x 1

## paths ##

## 1 /home/cbrunos/six\_to/spreadsheets/albert\_meyers 1 1-25act.xlsx

## 2 /home/cbrunos/six\_to/spreadsheets/albert\_meyers 2 1-29act.xlsx

## 3 /home/cbrunos/six\_to/spreadsheets/andrea\_ring 10 ENRONGAS(1200).xlsx

## 4 /home/cbrunos/six\_to/spreadsheets/andrea\_ring 11 ENRONGAS(0101).xlsx

## 5 /home/cbrunos/six\_to/spreadsheets/andrea\_ring 12 ENRONGAS(1200).xlsx ## 6 /home/cbrunos/six\_to/spreadsheets/andrea\_ring 13 Trader & Products

5-15-01…

## 7 /home/cbrunos/six\_to/spreadsheets/andrea\_ring 14 Trader & Products 5-16-01…

## 8 /home/cbrunos/six\_to/spreadsheets/andrea\_ring 15 IFERCnov.xlsx

## 9 /home/cbrunos/six\_to/spreadsheets/andrea\_ring 16 ifercdec.xlsx

## 10 /home/cbrunos/six\_to/spreadsheets/andrea\_ring 17 IFERCJan.xlsx ## # … with 20 more rows

Ok, so now, in order to read in all these files, I would write the following code:

enron %>%

mutate(datasets = map(paths, xlsx\_cells)

This would create a new column called datasets where each element would be a complete data set. If I run this in my 30 examples, it might be ok. But if I run it on the full thing, there’s no way I’m not going to run out of RAM. So how to solve this issue? How to run my neat get\_stats() function on all datasets if I cannot read in the data? The solution is to only read in the data when I need it, and only one dataset at a time. The solution is to build a *lazy* tibble. And this is possible using quo(). To quickly grasp what quo() does, let’s try calling the following expression once with, and once without quo():

runif(10)

## [1] 0.98342755 0.13500737 0.06196822 0.61304269 0.30600919 0.48015570

## [7] 0.05747049 0.04535318 0.37880304 0.70647563

This runs runif(10) returning 10 randomly generated numbers, as expected.

quo(unif(10)) ##

## expr: ^unif(10) ## env: global

This instead returns a quosure, which to be honest, is a complex beast. I’m not sure I get it myself. The definition, is that quosures are *quoted expressions that keep track of an environment*. For our practical purposes, we can use that to delay when the data gets read in, and that’s all that matters:

(enron <- enron %>%

mutate(datasets = map(paths, ~quo(xlsx\_cells(.)))) ## # A tibble: 30 x 2

## paths datasets ##

## 1 /home/cbrunos/six\_to/spreadsheets/albert\_meyers 1 1-25act.xlsx

This takes less than a second to run, and not just because I only have 30 paths. Even if I was working on the complete 15k+ datasets, this would run in an instant. That’s because we’re actually not reading in anything yet. We’re only setting the scene.

The magic happens now: we’re going to now map our function that computes the stats we need. We only need to change one thing. Let’s see:

get\_stats <- function(x){ x <- eval\_tidy(x

tribble(~has\_formula, ~n\_sheets, ~n\_formulas at\_least\_one\_formula(x), n\_sheets(x), n\_formulas(x)

}

I’ve added this line:

x <- eval\_tidy(x)

This evaluates the quosure, thus instantiating the dataset, and then proceeds to make all the computations. Let’s see what happens when we run this on our lazy tibble:

(enron <- enron %>%

mutate(stats = map(datasets, get\_stats)) ## # A tibble: 30 x 3

## paths datasets stats

##

## 1 /home/cbrunos/six\_to/spreadsheets/albert\_meyers 1 1…

What happened here is nothing short of black magic: one by one, each quosure was instantiated, and the required stats were computed, then the dataset was thrown into the garbage before moving

on to the next quosure. This means that RAM usage was kept to a minimum, and I could have run this over my 15k+ spreadsheets without any issue. You can watch me run similar code in

my video here; I show how my RAM usage does not move even though I’m mapping over all the Excel sheets.

The column stats now holds one dataframe with one row and three columns for each Excel file. Because stats is a list-column of dataframes, we can use unnest() to get to the data.

Let’s take a closer look on one dataframe:

enron %>% head(1) %>%

select(paths, stats) %>% unnest(cols = stats

## # A tibble: 1 x 4

## paths has\_formula n\_sheets

n\_formulas ##

## 1 /home/cbrunos/six\_to/spreadsheets/albert\_meye… TRUE 11

3334

We see that by using unnest(), the two columns inside the nested dataframe get expanded and become columns of the “main” dataframe.

We’re done, but let’s clean up the dataset a little bit and take a look at the results:

(

enron <- enron %>

mutate(excel\_file = str\_remove(paths, "/home/cbrunos/six\_to/

spreadsheets/")) %>%

select(-paths, -datasets) %>% unnest(cols = stats

)

## # A tibble: 30 x 4

## has\_formula n\_sheets n\_formulas excel\_file ##

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ## | 1 | TRUE | 11 | 3334 | albert\_meyers 1 1-25act.xlsx |
| ## | 2 | TRUE | 11 | 3361 | albert\_meyers 2 1-29act.xlsx |
| ## | 3 | TRUE | 4 | 550 | andrea\_ring 10 ENRONGAS(1200).xlsx |
| ## | 4 | TRUE | 4 | 549 | andrea\_ring 11 ENRONGAS(0101).xlsx |
| ## | 5 | TRUE | 4 | 550 | andrea\_ring 12 ENRONGAS(1200).xlsx |
| ## | 6 | FALSE | 0 | 0 | andrea\_ring 13 Trader & Products |
| 5-15-01 E…  ## 7 FALSE | | | 0 | 0 | andrea\_ring 14 Trader & Products |
| 5-16-01 E… | | |  |  |  |
| ## | 8 | TRUE | 1 | 169 | andrea\_ring 15 IFERCnov.xlsx |
| ## | 9 | TRUE | 1 | 177 | andrea\_ring 16 ifercdec.xlsx |
| ## | 10 | TRUE | 1 | 162 | andrea\_ring 17 IFERCJan.xlsx |
| ## | # … with | | 20 more rows | | |

Getting some statistics is now easy:

enron %>%

summarise(average\_n\_formulas = mean(n\_formulas) max\_sheets = max(n\_sheets)

## # A tibble: 1 x 2

## average\_n\_formulas max\_sheets ##

## 1 490. 11

By the way, now that we see that the code works, we can run it on all the spreadsheets simply by not running the following line:

(enron <- head(enron, 30))

Also, we can quite easily run all of this in parallel using {furrr}:

library(furrr)

## Loading required package: future plan(multiprocess, workers = 12

enron <- enframe(list\_paths, name = NULL, value = "paths")

enron <- head(enron, 1200) #just to compile the document faster, I only consider 1200 Excel spreadsheets

enron <- enron %>%

mutate(datasets = map(paths, ~quo(xlsx\_cells(.)))

start <- Sys.time() enron <- enron %>%

mutate(stats = future\_map(datasets, get\_stats)

Sys.time() - start

## Time difference of 36.86839 secs

Same code, no parallelization (it takes longer, obviously):

enron <- enframe(list\_paths, name = NULL, value = "paths") enron <- head(enron, 1200)

enron <- enron %>%

mutate(datasets = map(paths, ~quo(xlsx\_cells(.)))

start <- Sys.time() enron <- enron %>%

mutate(stats = map(datasets, get\_stats)

Sys.time() - start

## Time difference of 1.217199 mins

I think this is pretty neat.