One topic I haven't discussed in my previous posts about automating tasks with loops or doing simulations is how to deal with errors. If we have unanticipated errors a `map()` or `lapply()` loop will come to a screeching halt with no output to show for the time spent. When your task is time-consuming, this can feel pretty frustrating, since the whole process has to be restarted.

How to deal with errors? Using functions `try()` or `tryCatch()` when building a function is the traditional way to catch and address potential errors. In the past I've struggled to remember how to use these, though, and functions `possibly()` and `safely()` from package **purrr** are convenient alternatives that I find a little easier to use.

In this post I'll show examples on how to use these two functions for handling errors. I'll also demonstrate the use of the related function `quietly()` to capture other types of output, such as warnings and messages.

Table of Contents

R packages

The functions I'm highlighting today are from package **purrr**. I'll also use **lme4** for fitting simulated data to linear mixed models.

```
library(purrr) # v. 0.3.4
library(lme4) # v. 1.1-23
```

Using possibly() to return values instead of errors

When doing a repetitive task like fitting many models with a `map()` loop, an error in one of the models will shut down the whole process. We can anticipate this issue and bypass it by defining a value to return if a model errors out via `possibly()`.

I created the very small dataset below to demonstrate the issue. The goal is to fit a linear model of `y` vs `x` for each `group`. I made exactly two groups here, *a* and *b*, to make it easy to see what goes wrong and why. Usually we have many more groups and potential problems can be harder to spot.

```
dat = structure(list(group = c("a", "a", "a", "a", "a", "a", "b", "b", "b"),
                     x = c("A", "A", "A", "B", "B", "B", "A", "A", "A"),
                     y = c(10.9, 11.1, 10.5, 9.7, 10.5, 10.9, 13, 9.9, 10.3)),
                class = "data.frame",
                row.names = c(NA, -9L))
dat
```

```
#   group x    y
# 1     a A 10.9
# 2     a A 11.1
# 3     a A 10.5
# 4     a B  9.7
# 5     a B 10.5
# 6     a B 10.9
```

```
# 7     b A 13.0
# 8     b A  9.9
# 9     b A 10.3
```

I'll first split the dataset by `group` to get a list of data.frames to loop through.

```
dat_split = split(dat, dat$group)
```

Then I'll loop through each dataset in the list with `map()` and fit a linear model with `lm()`. Instead of getting output, though, I get an error.

```
map(dat_split, ~lm(y ~ x, data = .x) )
```

```
Error in `contrasts<-`(`*tmp*`, value = contr.funs[1 + isOF[nn]]): contrasts
can be applied only to factors with 2 or more levels
```

What's going on? If you look at the dataset again, you'll see that `x` from group *b* contains only a single value. Once you know that you can see the error actually is telling us what the problem is: we can't use a factor with only one level.

Model *a* fits fine, since `x` has two values.

```
lm(y ~ x, data = dat, subset = group == "a")
```

```
#
# Call:
# lm(formula = y ~ x, data = dat, subset = group == "a")
#
# Coefficients:
# (Intercept)           xB
#     10.8333      -0.4667
```

It is the *b* model that fails.

```
lm(y ~ x, data = dat, subset = group == "b")
```

```
Error in `contrasts<-`(`*tmp*`, value = contr.funs[1 + isOF[nn]]): contrasts
can be applied only to factors with 2 or more levels
```

You can imagine that the problem of having only a single value for the factor in some groups could be easy to miss if you working with a large number groups. This is where `possibly()` can help, allowing us to keep going through all groups regardless of errors. We can then find and explore problem groups.

Wrapping a function with possibly()

The `possibly()` function is a *wrapper* function. It wraps around an existing function. Other than defining the function to wrap, the main argument of interest is `otherwise`. In `otherwise` we define what value to return if we get an error from the function we are wrapping.

I make a new wrapped function called `posslm1()`, which wraps `lm()` and returns "Error" if an error occurs when fitting the model.

```
posslm1 = possibly(.f = lm, otherwise = "Error")
```

When I use `posslm1()` in my model fitting loop, you can see that loop now finishes. Model *b* contains the string "Error" instead of a model.

```
map(dat_split, ~posslm1(y ~ x, data = .x) )
```

```
# $a
#
# Call:
```

```
# .f(formula = ..1, data = ..2)
#
# Coefficients:
# (Intercept)           xB
#     10.8333      -0.4667
#
#
# $b
# [1] "Error"
```

Here's another example of `possibly()` wrapped around `lm()`, this time using `otherwise = NULL`. Depending on what we plan to do with the output, using `NULL` or `NA` as the return value can be useful when using `possibly()`.

Now group *b* is `NULL` in the output.

```
posslm2 = possibly(.f = lm, otherwise = NULL)
( mods = map(dat_split, ~posslm2(y ~ x, data = .x) ) )
```

```
# $a
#
# Call:
# .f(formula = ..1, data = ..2)
#
# Coefficients:
# (Intercept)           xB
#     10.8333      -0.4667
#
#
# $b
# NULL
```

Finding the groups with errors

Once the loop is done, we can examine the groups that had errors when fitting models. For example, I can use `purrr::keep()` to keep only the results that are `NULL`.

```
mods %>%
     keep(~is.null(.x) )
```

```
# $b
# NULL
```

This allows me to pull out the names for the groups that had errors. Getting the names in this way is one reason I like that `split()` returns named lists.

```
group_errs = mods %>%
     keep(~is.null(.x) ) %>%
     names()
group_errs
```

```
# [1] "b"
```

Once I have the names of the groups with errors, I can pull any problematic groups out of the original dataset or the split list to examine them more closely. (I use `%in%` here in case `group_errs` is a vector.)

```
dat[dat$group %in% group_errs, ]
```

```
#   group x    y
# 7     b A 13.0
# 8     b A  9.9
# 9     b A 10.3
```

```
dat_split[group_errs]
```

```
# $b
#   group x    y
# 7     b A 13.0
# 8     b A  9.9
# 9     b A 10.3
```

Using compact() to remove empty elements

You may come to a point where you've looked at the problem groups and decide that the models with errors shouldn't be used in further analysis. In that case, if all the groups with errors are `NULL`, you can use `purrr::compact()` to remove the empty elements from the list. This can make subsequent loops to get output more straightforward in some cases.

```
compact(mods)
```

```
# $a
#
# Call:
# .f(formula = ..1, data = ..2)
#
# Coefficients:
# (Intercept)           xB
#     10.8333      -0.4667
```

Using safely() to capture results and errors

Rather than replacing the errors with values, `safely()` returns both the results and the errors in a list. This function is also a wrapper function. It defaults to using `otherwise = NULL`, and I generally haven't had reason to change away from that default.

Here's an example, wrapping `lm()` in `safely()` and then using the wrapped function `safelm()` to fit the models.

```
safelm = safely(.f = lm)
mods2 = map(dat_split, ~safelm(y ~ x, data = .x) )
```

The output for each group is now a list with two elements, one for results (if there was no error) and the other for the error (if there was an error).

Here's what this looks like for model *a*, which doesn't have an error. The output contains a result but no error.

```
mods2[[1]]
```

```
# $result
#
# Call:
# .f(formula = ..1, data = ..2)
#
# Coefficients:
# (Intercept)           xB
#     10.8333      -0.4667
#
#
# $error
# NULL
```

Model *b* didn't work, of course, so the results are `NULL` but the error was captured in `error`.

```
mods2[[2]]
```

```
# $result
# NULL
#
# $error
#
```

Exploring the errors

One reason to save the errors using `safely()` is so we can take a look at what the errors were for each group. This is most useful with informative errors like the one in my example.

Errors can be extracted with a `map()` loop, pulling out the "error" element from each group.

```
map(mods2, "error")
```

```
# $a
# NULL
#
# $b
#
```

Extracting results

Results can be extracted similarly, and, if relevant, `NULL` results can be removed via `compact()`.

```
mods2 %>%
     map("result") %>%
     compact()
```

```
# $a
#
# Call:
# .f(formula = ..1, data = ..2)
#
# Coefficients:
# (Intercept)           xB
#     10.8333      -0.4667
```

Using quietly() to capture messages

The `quietly()` function doesn't handle errors, but instead captures other types of output such as warnings and messages along with any results. This is useful for exploring what kinds of warnings come up when doing simulations, for example.

A few years ago I wrote a post showing a simulation for a linear mixed model. I use the following function, pulled from that earlier post.

```
twolevel_fun = function(nstand = 5, nplot = 4, mu = 10, sigma_s = 1, sigma = 1) {
     standeff = rep( rnorm(nstand, 0, sigma_s), each = nplot)
     stand = rep(LETTERS[1:nstand], each = nplot)
     ploteff = rnorm(nstand*nplot, 0, sigma)
     resp = mu + standeff + ploteff
     dat = data.frame(stand, resp)
     lmer(resp ~ 1 + (1|stand), data = dat)
}
```

One thing I skipped discussing in that post were the messages returned for some simulations. However, I can certainly picture scenarios where it would be interesting and important to capture warnings and messages to see, e.g., how often they occur even when we know the data comes from the model.

Here I'll set the seed so the results are reproducible and then run the function 10 times. You see I get two messages, indicating that two of the ten models returned a message. In this case, the message indicates that the random effect variance is estimated to be exactly 0 in the model.

```
set.seed(16)
sims = replicate(10, twolevel_fun(), simplify = FALSE )

# boundary (singular) fit: see ?isSingular
# boundary (singular) fit: see ?isSingular
```

It turns out that the second model in the output list is one with a message. You can see at the bottom of the model output below that there is 1 lme4 warning.

```
sims[[2]]

# Linear mixed model fit by REML ['lmerMod']
# Formula: resp ~ 1 + (1 | stand)
#    Data: dat
# REML criterion at convergence: 45.8277
# Random effects:
#  Groups   Name        Std.Dev.
#  stand    (Intercept) 0.0000
#  Residual             0.7469
# Number of obs: 20, groups:  stand, 5
# Fixed Effects:
# (Intercept)
#       10.92
# convergence code 0; 0 optimizer warnings; 1 lme4 warnings
```

The **lme4** package stores warnings and messages in the model object, so I can pull the message out of the model object.

```
sims[[2]]@optinfo$conv$lme4

# $messages
# [1] "boundary (singular) fit: see ?isSingular"
```

But I think `quietly()` is more convenient for this task. This is another wrapper function, and I'm going to wrap it around `lmer()`. I do this because I'm focusing specifically on messages that happen when I fit the model. However, I could have wrapped `twolevel_fun()` and captured any messages from the entire simulation process.

I use my new function `qlmer()` inside my simulation function.

```
qlmer = quietly(.f = lmer)
qtwolevel_fun = function(nstand = 5, nplot = 4, mu = 10, sigma_s = 1, sigma = 1) {
     standeff = rep( rnorm(nstand, 0, sigma_s), each = nplot)
     stand = rep(LETTERS[1:nstand], each = nplot)
     ploteff = rnorm(nstand*nplot, 0, sigma)
     resp = mu + standeff + ploteff
     dat = data.frame(stand, resp)
     qlmer(resp ~ 1 + (1|stand), data = dat)
}
```

I set the seed back to 16 so I get the same models and then run the function using `qlmer()` 10 times. Note this is considered *quiet* because the messages are now captured in the output by `quietly()` instead of printed.

The wrapped function returns a list with 4 elements, including the results, any printed output, warnings, and messages. You can see this for the second model here.

```
set.seed(16)
sims2 = replicate(10, qtwolevel_fun(), simplify = FALSE)
```

```
sims2[[2]]
```

```
# $result
# Linear mixed model fit by REML ['lmerMod']
# Formula: resp ~ 1 + (1 | stand)
#    Data: ..2
# REML criterion at convergence: 45.8277
# Random effects:
#  Groups   Name        Std.Dev.
#  stand    (Intercept) 0.0000
#  Residual             0.7469
# Number of obs: 20, groups:  stand, 5
# Fixed Effects:
# (Intercept)
#       10.92
# convergence code 0; 0 optimizer warnings; 1 lme4 warnings
#
# $output
# [1] ""
#
# $warnings
# character(0)
#
# $messages
# [1] "boundary (singular) fit: see ?isSingular\n"
```

In a simulation setting, I think seeing how many times different messages and warnings come up could be pretty interesting. It might inform how problematic a message is. If a message is common in simulation we may feel more confident that such a message from a model fit to our real data is not a big issue.

For example, I could pull out all the `messages` and then put the results into a vector with `unlist()`.

```
sims2 %>%
     map("messages") %>%
     unlist()
```

```
# [1] "boundary (singular) fit: see ?isSingular\n"
# [2] "boundary (singular) fit: see ?isSingular\n"
```

If I wanted to extract multiple parts of the output, such as keeping both messages and warnings, I can use the extract brackets in `map()`.

These results don't look much different compared to the output above since there are no warnings in my example. However, note the result is now in a named vector so I could potentially keep track of which are `messages` and which are `warnings` if I needed to.

```
sims2 %>%
     map(`[`, c("messages", "warnings") ) %>%
     unlist()
```

```
#                                    messages
# "boundary (singular) fit: see ?isSingular\n"
#                                    messages
# "boundary (singular) fit: see ?isSingular\n"
```

I showed only fairly simple way to use these three functions. However, you certainly may find yourself using them for more complex tasks. For example, I've been in situations in the past where I wanted to keep only models that didn't have errors when building parametric bootstrap confidence intervals. If they had existed at the time, I could have used `possibly()` or `safely()` in a `while()` loop, where the bootstrap data would be redrawn until a model fit without error. Very useful! 😉

Just the code, please

Here's the code without all the discussion. Copy and paste the code below or you can download an R script of uncommented code from here.

```
library(purrr) # v. 0.3.4
library(lme4) # v. 1.1-23

dat = structure(list(group = c("a", "a", "a", "a", "a", "a", "b", "b", "b"),
                     x = c("A", "A", "A", "B", "B", "B", "A", "A", "A"),
                     y = c(10.9, 11.1, 10.5, 9.7, 10.5, 10.9, 13, 9.9, 10.3)),
                class = "data.frame",
                row.names = c(NA, -9L))
dat

dat_split = split(dat, dat$group)
map(dat_split, ~lm(y ~ x, data = .x) )

lm(y ~ x, data = dat, subset = group == "a")
lm(y ~ x, data = dat, subset = group == "b")

posslm1 = possibly(.f = lm, otherwise = "Error")
map(dat_split, ~posslm1(y ~ x, data = .x) )

posslm2 = possibly(.f = lm, otherwise = NULL)
( mods = map(dat_split, ~posslm2(y ~ x, data = .x) ) )

mods %>%
     keep(~is.null(.x) )

group_errs = mods %>%
     keep(~is.null(.x) ) %>%
     names()
group_errs

dat[dat$group %in% group_errs, ]
dat_split[group_errs]

compact(mods)

safelm = safely(.f = lm)
mods2 = map(dat_split, ~safelm(y ~ x, data = .x) )
mods2[[1]]
mods2[[2]]

map(mods2, "error")

mods2 %>%
     map("result") %>%
     compact()

twolevel_fun = function(nstand = 5, nplot = 4, mu = 10, sigma_s = 1, sigma = 1) {
     standeff = rep( rnorm(nstand, 0, sigma_s), each = nplot)
     stand = rep(LETTERS[1:nstand], each = nplot)
     ploteff = rnorm(nstand*nplot, 0, sigma)
     resp = mu + standeff + ploteff
     dat = data.frame(stand, resp)
     lmer(resp ~ 1 + (1|stand), data = dat)
```

```
}

set.seed(16)
sims = replicate(10, twolevel_fun(), simplify = FALSE )
sims[[2]]
sims[[2]]@optinfo$conv$lme4

qlmer = quietly(.f = lmer)
qtwolevel_fun = function(nstand = 5, nplot = 4, mu = 10, sigma_s = 1, sigma = 1) {
     standeff = rep( rnorm(nstand, 0, sigma_s), each = nplot)
     stand = rep(LETTERS[1:nstand], each = nplot)
     ploteff = rnorm(nstand*nplot, 0, sigma)
     resp = mu + standeff + ploteff
     dat = data.frame(stand, resp)
     qlmer(resp ~ 1 + (1|stand), data = dat)
}

set.seed(16)
sims2 = replicate(10, qtwolevel_fun(), simplify = FALSE)
sims2[[2]]

sims2 %>%
     map("messages") %>%
     unlist()

sims2 %>%
     map(`[`, c("messages", "warnings") ) %>%
     unlist()
```