How to use `tryCatch` for robust R scripting

Rahul Sangole

Why would you want to read this?

Using `tryCatch` to write robust R code can be a bit confusing. I found the help file dry to read.

Over the years, I have developed a few programming paradigms which I've repeatedly found useful. A quick introduction to `tryCatch`, followed by three use-cases I use on a regular basis.

What is `tryCatch` and why should you use it?

`tryCatch` allows us to write "robust" R code. What does this mean?

When R scripts or functions execute, one of three things happen upon execution of each line…

- It runs correctly as expected
- It returns some sort of warning… not enough to stop execution, but a warning to the user that something could be going on that needs attention
- It returns an error and completely stops the program

Without `tryCatch` the warnings are lost in the void, and the errors simply halt execution completely.

But what happens if you're doing a for loop, looping over 140,000 "things" and only ONE of those things causes an error? Without `tryCatch` it'll just stop the whole program and all your calculations in the for loop will be lost! Wouldn't it be great if, we could tell R — if there's an error in one of the loops, just ignore it and keep going?

Welcome to the beauty of `tryCatch`

`tryCatch` syntax

`tryCatch` has a slightly complex syntax structure. However, once we understand the 4 parts which constitute a complete `tryCatch` call as shown, it becomes easy to remember:

- `expr` : [Required] R code(s) to be evaluated
- `error` : [Optional] What should run if an error occured while evaluating the codes in `expr`
- `warning` : [Optional] What should run if a warning occured while evaluating the codes in `expr`
- `finally` : [Optional] What should run just before quitting the `tryCatch` call, irrespective of if `expr` ran succcessfuly, with an error, or with a warning

```
tryCatch(
        expr = {
                # Your code...
                # goes here...
                # ...
        },
        error = function(e){
                # (Optional)
                # Do this if an error is caught...
        },
        warning = function(w){
                # (Optional)
                # Do this if an warning is caught...
        },
        finally = {
                # (Optional)
                # Do this at the end before quitting the tryCatch structure...
        }
)
```

`tryCatch` within loops

There are cases at work where I have quite large datasets to pre-process before model building can begin. The sources of these data can be varied and thus the quality of these data can vary. While each dataset *should* conform to our data quality standards (datatypes, data dictionaries, other domain-specific constraints), very often these isn't the case. As a result, common data preprocessing functions might fail on few datasets. We can use `tryCatch` within the `for` loop to catch errors without breaking the loop.

Say, we have a nested dataframe of the `mtcars` data, nested on the cylinder numbers, and say, we had a few character values in `mpg` which is our response variable.

```
# Example nested dataframe
(df_nested <- mtcars %>% as_tibble() %>% tidyr::nest(-cyl))
## # A tibble: 3 x 2
##     cyl data
##   <dbl> <list>
## 1     6 <tibble [7 × 10]>
## 2     4 <tibble [11 × 10]>
## 3     8 <tibble [14 × 10]>

df_nested$data[[2]][c(4,8),"mpg"] <- "Missing"
```

We wish to run a few custom preprocessors, including taking the log of `mpg`.

```
convert_gear_to_factors <- function(df){
        df %>%
                mutate(
                        gear = factor(gear, levels = 1:5,
                                      labels = paste0("Gear_",1:5))
                        )
        }
transform_response_to_log <- function(df){
        df %>%
                mutate(log_mpg = log(mpg)) %>%
                select(-mpg)
        }
```

How do we run our preprocessors over all the rows without error-ing out?

```
for (indx in 1:nrow(df_nested)) {
        tryCatch(
                expr = {
                        df_nested[[indx, "data"]] <-  df_nested[[indx, "data"]] %>%
                                convert_gear_to_factors() %>%
                                transform_response_to_log()
                        message("Iteration ", indx, " successful.")
                },
                error = function(e){
                        message("* Caught an error on itertion ", indx)
                        print(e)
                }
        )
}
```

Output >>

```
## Iteration 1 successful.

## * Caught an error on itertion 2
## <Rcpp::eval_error in mutate_impl(.data, dots): Evaluation error: non
-numeric argument to mathematical function.>

## Iteration 3 successful.
```

`tryCatch` to catch and log issues early & often

An important component of preparing 'development' code to be 'production' ready is implementation of good defensive programming and logging practices. I won't go into details of either here, except to showcase the style of programs I have been writing to prepare code before it goes to our production cluster.

```
preprocess_data <- function(df, x, b, ...){
        message("-- Within preprocessor")
        df %>%
                assertive::assert_is_data.frame() %>%
                assertive::assert_is_non_empty()
        x %>%
                assertive::assert_is_numeric() %>%
                assertive::assert_all_are_greater_than(3.14)
        b %>%
                assertive::assert_is_a_bool()

        # Code here...
        # ....
        # ....

        return(df)
}
build_model <- function(...){message("-- Building model...")}
eval_model  <- function(...) {message("-- Evaluating model...")}
save_model  <- function(...) {message("-- Saving model...")}
```

Each utility function starts with checking arguments. There are plenty of packages which allow run-time testing. My favorite one is assertive. It's easy to read the code, and it's pipe-able. Errors and warnings are handled using `tryCatch` - they are printed to the console if running in interactive mode, and then written to log files as well. I have written my own custom logging functions, but there are packages like logging and log4r which work perfectly fine.

```
main_executor <- function(...){
        tryCatch(
                expr = {
                        preprocess_data(df, x, b, more_args,...) %>%
                                build_model() %>%
                                eval_model() %>%
                                save_model()
                },
                error = function(e){
                        message('** ERR at ', Sys.time(), " **")
                        print(e)
                        #Custom logging function
                        write_to_log_file(e, logger_level = "ERR")
                },
                warning = function(w){
                        message('** WARN at ', Sys.time(), " **")
                        print(w)
                        #Custom logging function
                        write_to_log_file(w, logger_level = "WARN")
                },
                finally = {
                        message("--- Main Executor Complete ---")
                }
        )
}
```

`tryCatch` while model building

`tryCatch` is quite invaluable during model building. This is an actual piece of code I wrote for a kaggle competition as part of the *413-Time Series* midterm. Github link here. The details of what's going on isn't important. At a high level, I was fitting `stlf` models using `forecast` for each shop, among 60 unique shop-ID numbers. For various reasons, for some shops, an `stlf` model could not be be fit, in which case a default seasonal naive model using `snaive` was to be used. `tryCatch` is a perfect way to handle such exceptions as shown below. I used a similar approach while building models at an "item" level: the number of unique items was in the 1000s; manually debugging one at a time is impossible. `tryCatch` allows us to programmatically handle such situations.

```
stlf_yhats <- vector(mode = 'list', length = length(unique_shops))
for (i in seq_along(unique_shops)) {
        cat('\nProcessing shop', unique_shops[i])
        tr_data <- c6_tr %>% filter(shop_id == unique_shops[i])
        tr_data_ts <-
                dcast(
                        formula = yw ~ shop_id,
                        data = tr_data,
                        fun.aggregate = sum,
                        value.var = 'total_sales',
                        fill = 0
                )
        tr_data_ts <- ts(tr_data_ts[, -1], frequency = 52)

        ###################
        # <--Look here -->
        fit <- tryCatch(
                expr = {tr_data_ts %>% stlf(lambda = 'auto')},
                error = function(e) { tr_data_ts %>% snaive()}
        )
        ###################

        fc <- fit %>% forecast(h = h)
        stlf_yhats[[i]] <- as.numeric(fc$mean)
        stlf_yhats[[i]] <- ifelse(stlf_yhats[[i]] < 0, 0, stlf_yhats[[i]])
}
```

More resources

- https://www.rdocumentation.org/packages/R.oo/versions/1.2.7/topics/trycatch
- https://www.r-bloggers.com/careful-with-trycatch/
- http://adv-r.had.co.nz/Exceptions-Debugging.html