StARIS Report

Jonah McArthur

Aim

During summer 2022, I completed a project working with the Distance for Windows software, with the aim of writing a function in R which would allow more direct comparison between analyses performed using the MRDS and MCDS engines within the software. This function took the input in the format it is provided to MRDS, and used that input to create a command file that would run an equivalent analysis in MCDS. The aim of this project was to work with the function I previously created and improve its readability and modularity.

Method

In preparation for the coding segment of this project, I read through resources on efficiency and modularity provided by Dr Miller, in order to improve my understanding of the methods by which the code could be improved.

Based upon these methods and preliminary discussions with Dr Miller, I then proceeded to plan the next steps for implementing these ideas into the existing code.

The command file that the code is designed to create is split into four sections: the header, the OPTIONS section, the DATA section, and the ESTIMATE section, each of which had its own section within the `create_command_file()` function. Within this function, there was also a large section dedicated to reformatting the data to resemble the MCDS data input more closely. Each of these sections was relatively self contained, so the first step of the planning was to create functions that contained the code for each section, with the aim of allowing each section to be tested individually and making the `create_command_file()` function much more readable.

Within the overall function, there are also many cases of repeated processes. I identified these and designed functions that could be called multiple times, to remove the unnecessary repetition and allow for more consistency throughout the code.

Splitting the code into sections

The HEADER section only contained six lines, so was not separated out into its own function. The four remaining sections were written into their own functions: `reformat_data()`, `options_section()`, `data_section()`, and `estimate_section()`.

When analysing the inputs and outputs of each function, it became clear that in almost every case the functions solely required inputs from within the environment of the `create_command_file()` function, and did not output anything but instead simply performed processes. The only exception of this was the reformat data section, which output the new data frame to be used in all future functions, and defined the variables `covar_pres` and `covar_fields`, which contained information gleaned about the presence of covariates within the model. These variables were defined in the local environment of `reformat_data` but were required in both `data_section` and `estimate_section`. In order to ensure that these functions could access the values of `covar_pres` and `covar_fields`, their values were returned from `reformat_data` alongside the new data frame and used to define the variables within the wider environment.

The return line from `reformat_data` and the lines used to call the function and use its output to define variables within the wider environment is included below:

```
  output <- list(data, covar_pres, covar_fields)
  return(output)
```

```
  # Reformatting the data
  data_info <- reformat_data(data, dsmodel)
  data <- data_info[1]

  # extracting the information about covariates so that it can be used in other functions
  covar_pres <- data_info[2]
  covar_fields <- data_info[3]
```

Defining new functions

There were three processes that were found to be repeated throughout the code that were then written into their own functions.

The first was the R function `cat()` used throughout the code to concatenate relevant strings to the command file:

```
cat("OPTIONS;", file=command.file.name, "\n", append=TRUE)
```

Each instance of the function being called shared at least two inputs - `file=command.file.name` and `append=TRUE` - and many included `"\n"` to signal a line break, therefore the only distinct part of each `cat()` call was the string itself. For this reason, I wrote the `cat_file()` function, which had only two inputs: a string of text to be concatenated to the file, and a boolean to determine whether there should be a line break following the text, which had default `TRUE`. The other two inputs of the `cat()` function were already specified in `cat_file()`, so this reduced most `cat()` calls to one or at most two inputs, removing repetition and making the code considerably easier to read. The above line can now be written simply as:

```
cat_file("OPTIONS;")
```

The second function created was the `cat_conditions()` function, which could be used where a variable had multiple specified potential values, and the text concatenated to the command file depended upon which value the variable had. This situation occurred frequently within the OPTIONS and ESTIMATE sections:

```
  if(cluster == TRUE){
    cat("OBJECT=CLUSTER;", file=command.file.name, "\n", append=TRUE)
  }else{
    cat("OBJECT=SINGLE;", file=command.file.name, "\n", append=TRUE)
  }
```

The `cat_conditions()` function has four inputs:

- `switch_input`: the variable whose value was being checked
- `conditions`: a vector of the potential values that `switch_input` was expected to take; this has the default `c(TRUE, FALSE)` as multiple inputs were found to have these specific conditions
- `results`: a vector of strings to be concatenated to the command file depending on the value `switch_input` was found to have, with the nth entry corresponding to the nth entry of `conditions`
- `new_line`: a boolean specifying whether there should be a line break following the string

The `cat_conditions()` function compares `switch_input` to `conditions` to find the index at which it occurs, and then identifies the correct response string, and uses `cat_file()` to concatenate the response. This was written with the following code:

```
cat_conditions <- function(switch_input, results, conditions=c(TRUE,FALSE), new_line=TRUE) {
  # checking that the input is present
```

```
  if(!is.null(switch_input)){
    # find the index of the condition which is found to be met
    result_index <- grep(switch_input, conditions)
    # find the corresponding response
    fin_result <- results[[result_index]]
    # write the correct command to the command file
    cat_file(text=fin_result, new_line=new_line)
  }
}
```

This then allowed the `options_section()` code to be reduced to a list containing each set of conditions to check, and a `for` loop performing the `cat_conditions()` function for each set of conditions:

```
  # creating a list with all the options and results
  opt_list <- list(point1=list(var=meta.data$point,
                               results=list("TYPE=POINT;","TYPE=LINE;")),
                   point2=list(var=meta.data$point,
                               results=list(c("DISTANCE=RADIAL /UNITS='Meters' /WIDTH=",
                                              meta.data$width, ";"),
                                            c("DISTANCE=PERP /UNITS='Meters' /WIDTH=",
                                              meta.data$width, ";"))),
                   cluster=list(var=cluster,
                                results=list("OBJECT=CLUSTER;","OBJECT=SINGLE;")),
                   debug=list(var=control$debug,
                              results=list("DEBUG=ON;","")))

  # looping through each of the options and concatenating the relevant commands
  for(i in 1:length(opt_list)) {
    # inputting the relevant data to the cat_conditions function
    cat_conditions(opt_list[[i]][[1]],opt_list[[i]][[2]])
  }
```

The final function was `id_fields()`, used in `reformat_data()` to identify columns of the data frame based on a vector of potential names it is suspected to have. This is to deal with the situation where the column names of the data frame do not match those recognised by MCDS. For example, in some input data the field used for effort is named `Search.time`, whereas MCDS only recognises the field if it is named `SMP_EFFORT`.

This function has two inputs:

- `data_cols`: the original column names of the data frame
- `pot_names`: a vector containing the potential names that are expected for a specific field, for example `c("effort","SMP_EFFORT", "Search.time")` for the `SMP_EFFORT` field.

The function will identify the column that has one of the expected names, and outputs the index of that column:

```
id_fields <- function(data_cols, pot_names) {
  # first, identify which column contains the relevant data
  # we want to check through each of the potential names to see which appears
  # in the column names, so we need to know the number of potential names and
  # keep track of whether it is present
  name_pres <- FALSE
  # convert vector of potential names to lower case for ease of comparison
  pot_names <- tolower(pot_names)
  # to ensure columns that include parts of other column names aren't mistakenly
  # identified
  pot_names <- gsub("^","^",pot_names)
```

```
  pot_names <- gsub("$","$",pot_names)
  # check through each of the potential column names to identify if it is used
  for(i in 1:length(data_cols)) {
    if(TRUE %in% grepl(pot_names[i],tolower(data_cols))) {
      # identify the index of the column that needs renamed
      name_pres <- grep(pot_names[i],tolower(data_cols))
    }
  }
  # return either that the name wasn't found, or the index it was found at
  return(name_pres)
}
```

This was then used in `reformat data()` to reduce the amount of repetition. One segment of `reformat_data()` where this is used is:

```
  # create a vector to store the indices of the recognised fields
  fields_index <- rep("", length(req_fields))
  # for each of the required fields, find the column and add in if not present
  for(i in 1:length(req_fields)) {
    # record which column the ith required field is found in the dataframe
    fields_index[i] <- id_fields(colnames(data),req_fields[[i]][[2]])
    # if the field is not present but is required, add in a dummy variable
    if(fields_index$index[i] == FALSE && req_fields[[i]][[3]] == TRUE) {
      data$new <- rep(1,nrow(data))
      tail(colnames(data),1) <- req_fields[[i]][[1]]
    } else {
      # rename the column to match the name required by MCDS
      colnames(data)[strtoi(fields_index[i])] <- req_fields[[i]][[1]]
    }
  }
```

Combining the new functions

These functions were combined into a new function named `create_command_file_mod()`, which can be found in the repository, alongside all other functions mentioned above:

https://github.com/jonah417/summer_project/tree/main/mcds_input

Conclusion and Next Steps

The creation of the new functions has drastically reduced the amount of repetition in the `create_command_file()` function, therefore making it more efficient and more readable. In addition, it is now possible to test, debug, and alter individual sections of the code independently, therefore making the code more robust.

Unfortunately, I ran out of time within my allotted 15 hours to complete testing of the function, so although the individual functions have been tested and appear to work as intended, the overall function still requires testing.

References

The following resources were sent to me by Dr Miller to give background information on the methods used to improve code.

http://adv-r.had.co.nz/Functional-programming.html https://stat545.com/functions-part1.html