Analysis Template

Simon Bond

2022-04-09

What this Document Does

The package `cctu` is designed to help run the analysis of clinical trials which typically produce an output of a large word document with one page per table or figure following on from a Statistical Analysis Plan, and requires substantial audit trails. Automating the trivialities of getting tables into word from R, and capturing the meta data and audit trials is what the package achieves.

This document should give you an overview of the tools provided in this package, explain the thought behind them, and show how to use them together. We will start by explaining the rationale, and then finish with a worked example. A complimentary set of R scripts and outputs, that aren't easily directly captured within a vignette are also provided with the /doc directory.

Key Concepts

Meta Table

As part of writing the Statistical Analysis Plan (SAP), a structured meta-table with one row per table or figure needs to be produced. This provides titles, subtitles, numbering, populations, etc. See the help page `?meta_table` for more details. The package is very reliant on this being provided by the user. Whilst producing the tables/figures the package will:

- look up which population is to be used for a table/figure and explicitly filter the data sets to the desired population
- whilst saving the output it will use the numbering to determing file names of xml or graphics files
- write the name of the code file that produced the output

When producing the final output it will use the meta-table to index all the outputs, and pull out the meta-data needed.

Given all the interactions needed, which is somewhat contrary to the concept in R of functions just returning output rather than modifying the user environment, the meta_table is stored within a environment that is private to the package. So it needs to be set up using `set_meta_table()` and `get_meta_table()` to extract a copy.

Populations

It is assumed that there is a data.frame that has one row per subject, and a column for each population defined in the SAP that is a logical indicating if a subject is included in each population. The definition of the populations, their names, and how the inclusion is to determined, are all important steps but outside of this package!

We provide a function `create_popn_envir` that takes the data.frame described above, and a list of other data.frames. It then filters each of the listed data.frame, looping over each of the populations, and saves them within a set of environments one per population.

During the analysis code, the function `attach_pop()` is repeatedly called with the number of a table/figure. The corresponding row from meta-table is used to identify which population is associated with the table/figure, and the environment is attached. So the list of dataframes that were filtered in `create_popn_envir` can now be accessed, safe in the knowledge that the right population is used.

Code Architecture

A set of default local files can be set up using `cctu_initialise` where the outputs will be stored using the default arguments.

The code itself is assumed to be modularised into a sequence of code that calls other code using `source`.

- `Main.R` at the top with a minimal amount of initialisation: working directory, PATHs, `library(cctu)`, `run_batch()`, then a sequence of `source()`, final calling of `create_word_xml()` and maybe further calls to `render` for other outputs.
- A configuration file, loading packages, setting graphical defaults, loading bespoke functions,
- Data import and manipulation: where cleaning, merging, creating derived variables maybe, and finally using `create_popn_envir` happens.
- Analysis: where the actual statistical work happens. May well be modularised further with individual scripts for each section of the study, ordered by Form or chronological ordering of the study.

It is important to call `library(cctu)` before the first use of `source`. The package modifies the base function `base::source`, so as to record the name of the file that called source and the name of the file sourced. This is to provide an audit trail allowing the sequence of code used to create a table/figure to be easily found. Like `meta_table` this audit trail lives in a private environment and is not easily directly accessible.

XML

The step of transfering table and figures to word is facilitated by xml. The tables are stored as a copy on a local drive (in /Output/Core by default) in semi-readible format with tags similar to html tables.

During the analysis, a block of code to produce a table or figure is put between an `attach_pop()` and a `write_table()` or `write_gglot()`. An argument to `write_table` will be a data frame; `write_ggplot` defaults to using the last plot object, or can take a ggplot object as an argument. Both then look up the number of the table called by `attach_pop` (temporarily stored in a private environment); the output is stored on the local drive either an XML file, or graphics file named with a suffix "_X.Y.Z" to identfy the output; the name of the code file that created the output is written to meta-table; by default a final call to `clean_up` removes all objects in the global environment (apart from those named in the vector `.reserved`) and detach the population environment.

After the analysis, the function `create_word_xml` glues these files all together along with the meta data into a large xml file. The tables are directly copied, and the figures have links to their file paths. Then xslt technology is used to convert the document into the schema used by MS Word, so it can be opened directly. Note though that the links to the graphics files would need to be turned into hard copies within Word if you want to move the document. If you want to tweak the style of the word document then you would edit the xslt file totally at your own risk!

Pointers

Other function may be worth looking up in their help files, and hopefully their names give a hint!

Data import & Manipulation

- `read_data`
- `data_table_summary`
- `clean_names`
- `remove_blank_rows_cols`

The set of external data files to be read in, along with the names of the corresponding data.frames created should be recorded in one initial data.frame. This can then be summarised in a table in the report using `data_table_summary` and also used to succinctly read in all the data files with `read_data`.

Analysis

- `sumby()` is a work-horse to produce standard summary tables, with additional figures produced as side-effects
- `propercase`
- `rbind_space` : to glue together repeated outputs from `sumby`
- `clean_up` is called by default within each call of `write_*`. They have arguments to not call `clean_up` if the code is such that you need to re-use the same R objects for multiple outputs.
- `.reserved` is a hidden variable in the global environment. `clean_up` ignores any objects named in `.reserved`. So you can edit this as an alternative to using the arguments within `write_*`, but ideally it should be defined once at the end of data manipulation.
- `rm_envir` maybe helpful during code development and interactive use of `attach_pop`.

Audit trails

As described already `source()` captures a trail of which code sources other code. There is a demonstration below of how to convert this into a graphical tree representation using a combination of Rmarkdown and latex. Or you can just get a copy `cctu:::cctu_env$code_tree` and write a local copy to a csv file say.

In a similar fashion we have some examples of Rmarkdown that can be use to read in the outputs from `write_*` and produce a Html or pdf, or ... version of the main report, depending on what the readers of the report prefer as their format.

At the end of your code it is good practice to run `Sys.info(), date(), sessionInfo()` to document the fine details of your session, package versions, etc. Also it may want to be run for the final definitive version of the report on a different server with a validated instance of R, using a `R CMD Batch` which has the nice side-effect of creating a .Rout file with a complete log of all the code and outputs. A wrapper function to avoid having to open a command line window for your operating system is `run_batch()` .

Worked Example

This document is used to illustrate how to set up a standard set of analysis using the library cctu. It assumes that you have copied across a template blank folder structure (`cctu_initialise`), created a library within the main folder with all the extra packages you may need, and set up a git versioning instance and rstudio project. A future project will be to document this step.

In the the top level this is a file called "main.R"

Initial lines

```
rm(list=ls())
#set to the library folder
.libPaths()
#> [1] "C:/Users/alimd/AppData/Local/Temp/RtmpeaTRPD/temp_libpath7748402747f7"
#> [2] "C:/Users/alimd/Documents/R/win-library/4.1"
#> [3] "C:/Program Files/R/R-4.1.0/library"
library(cctu)
#>
#> Attaching package: 'cctu'
#> The following object is masked from 'package:base':
#>
```

```
#>     source
options(verbose=TRUE)
#run_batch("main.R")
DATA <- "PATH_TO_DATA"
cctu_initialise()
rm_output()
```

If you run just these initial lines, the last command will evoke R CMD BATCH to run the entire set of code and produce a log file "main.Rout", which should be the final step, using the validated server. The `run_batch` line is commented out as the vignette will not work with this though..

This vignette now differs from a standard use, in that the `Main.R` file would now be a sequence of `source()` calls. Here we do run the source files, and then quote the R code they contain. There is a copy of all files and outputs from a standar use starting from Main.R

Configuration

It is recommended to set up a config.R file that

- reads in all your libraries
- sets up graphical settings
- maybe reads in some bespoke functions you want to define

```
source("Progs/config.R")
```

```
options(verbose = TRUE)

mylibs <- c("readxl", "dplyr", "ggplot2", "magrittr","tidyr","rmarkdown","knitr","xml2","rvest")

for(package in mylibs){
  library(package, character.only = TRUE)
}

# read in all function files
my_functions <- list.files("Progs\\functions")
for(file in my_functions){
  if( file != "archive"){
  source(paste0("Progs\\functions\\", file))
  }
}

# define theme for figures
default_theme <- theme_get()

graphical_theme <- theme_bw() + theme(
  axis.line.x      = element_line(color = "black"),
  axis.line.y      = element_line(color = "black"),
  panel.grid.major = element_blank() ,
  panel.grid.minor = element_blank(),
  panel.background = element_blank(),
  # panel.border = element_blank(),
  # axis.text = element_text(size = rel(1), angle = 45)
  axis.title.x     = element_text(margin = margin(t = 10)),
  legend.key       = element_rect(colour = "white", fill = NA),
  strip.background = element_rect(colour = "black")
)
```

```
# remove anything no longer required
rm(mylibs, package, my_functions, file)
```

Data Import

Next step is to import the meta-table, study data and apply manipulations, and finally create the population environments.

- Grab data from the "DATA" folder. Always use relative paths, or build absolute paths up from MACRO variables defined once neear the start 'paste0(DATA, "/myfile.csv")'
- convert the variable names into standard 'lower_case' so you have to remember less
- remove blank rows and columns
- give factor variables their levels
- make dates into dates if needed, whilst being careful of partial dates...

Here we grab a ready prepared 'dirty' raw data typical of MACRO DB and apply some of these concepts.

```
source("Progs/data_import.R")
```

```
options(stringsAsFactors = FALSE)

data_table <- data.frame(
  name=c("data","codes"),
  file=c("dirtydata.csv",
         "codes.csv"),
  folder=system.file("extdata", package="cctu"),
  stringsAsFactors = FALSE
)


read_data(data_table)
# if you wanted to read in just one data set, using non-standard options say
read_data( data_table[2,], remove_blank_rows_cols_option=FALSE, clean_names_option=FALSE)


set_meta_table( cctu::meta_table_example)
write_table(data_table_summary(data_table),number = "9", clean_up = FALSE)


# Not strictly needed as the default is to apply these two functions inside of read_data(), but this
# can be turned off with clean_names_option = FALSE , or remove_blank_rows_cols_option=FALSE
data %<>% clean_names() %>% remove_blank_rows_cols()
codes %<>% clean_names


for( x in unique(codes$var)){
  code_df <-  subset(codes, var==x)
  print(code_df)
  data[,x] <- factor( data[,x], levels=code_df$code, labels=code_df$label)
}

data$start_date <- as.POSIXct( data$start_date , format="%d/%m/%Y")
data_name <- names(data)
names(data)[match("subject_id", data_name)] <- "subjid"
```

```
#Create the population table

popn <- data[,"subjid", drop=FALSE]
popn$safety <- TRUE
popn$full <- popn$subjid<5

create_popn_envir("data",popn)

#tidy up
rm(code_df, codes, data_name, x)
.reserved <- ls()
```

Analysis

```
source("Progs/analysis.R")
```

```
attach_pop("1.1")
X <- rbind_space(
  sumby(age, treatment , data=data),
  sumby(gender, treatment, data=data)
)
write_table(X)

attach_pop("1.1.1")
X <- rbind_space(
  sumby(age, treatment, data=data, label="aGe (yAars)", text_clean = NULL),
  sumby(gender, treatment, data=data, text_clean = NULL)
)
write_table(X)


attach_pop("1.10")
X <- sumby(age, treatment, data=data)
write_ggplot(attr(X,"fig"))

attach_pop("2")
write_text("There were no deaths")
```

Creating the Report

Need to create names for the population labels, including the number of subjects. Good to name the report ending with the suffix ".doc".

```
pop_size <- sapply( popn[,names(popn)!="subjid"], sum)
pop_name <- unique(get_meta_table()$population)
index <- match(pop_name, names(pop_size))
popn_labels <- paste0(propercase(pop_name), " (n = ", pop_size[index],")")

write.csv(get_meta_table(), file=file.path("Output","meta_table.csv"), row.names = FALSE)
write.csv(get_code_tree(), file=file.path("Output","codetree.csv"), row.names = FALSE)
create_word_xml(report_title="Vignette Report",
                author="Simon Bond",
```

```
                filename=file.path("Output","Reports","Vignette_Report.doc"),
                popn_labels=popn_labels
            )
#> now dyn.load("C:/Users/alimd/Documents/R/win-library/4.1/xslt/libs/x64/xslt.dll") ...
#> C:\Users\alimd\OneDrive - University of Cambridge\Tools\cctu\vignettes\Output\Reports\Vignette_Repor
#> All figures in the word document are links to local files
#> You must manually include them with word if you want to move the word document.
Sys.info()
#>        sysname        release        version       nodename        machine
#>      "Windows"       "10 x64"  "build 22000"  "ALIM-LAPTOP"       "x86-64"
#>          login           user effective_user
#>        "alimd"        "alimd"        "alimd"
sessionInfo()
#> R version 4.1.0 (2021-05-18)
#> Platform: x86_64-w64-mingw32/x64 (64-bit)
#> Running under: Windows 10 x64 (build 22000)
#>
#> Matrix products: default
#>
#> locale:
#> [1] LC_COLLATE=English_United Kingdom.1252
#> [2] LC_CTYPE=English_United Kingdom.1252
#> [3] LC_MONETARY=English_United Kingdom.1252
#> [4] LC_NUMERIC=C
#> [5] LC_TIME=English_United Kingdom.1252
#> system code page: 936
#>
#> attached base packages:
#> [1] stats     graphics  grDevices utils     datasets  methods   base
#>
#> other attached packages:
#>  [1] rvest_1.0.2    xml2_1.3.3     knitr_1.36     rmarkdown_2.11 tidyr_1.1.4
#>  [6] magrittr_2.0.3 ggplot2_3.3.5  dplyr_1.0.7    readxl_1.3.1   cctu_0.6.3
#>
#> loaded via a namespace (and not attached):
#>  [1] Rcpp_1.0.8.3      prettyunits_1.1.1 ps_1.6.0          assertthat_0.2.1
#>  [5] rprojroot_2.0.3   digest_0.6.29     utf8_1.2.2        R6_2.5.1
#>  [9] cellranger_1.1.0  evaluate_0.14     httr_1.4.2        pillar_1.7.0
#> [13] rlang_1.0.2       rstudioapi_0.13   data.table_1.14.2 callr_3.7.0
#> [17] textshaping_0.3.6 desc_1.4.1        labeling_0.4.2    devtools_2.4.3
#> [21] xslt_1.4.3        stringr_1.4.0     munsell_0.5.0     compiler_4.1.0
#> [25] xfun_0.27         pkgconfig_2.0.3   systemfonts_1.0.3 pkgbuild_1.3.1
#> [29] htmltools_0.5.2   tidyselect_1.1.1  tibble_3.1.6      gridExtra_2.3
#> [33] fansi_1.0.3       crayon_1.5.1      withr_2.5.0       grid_4.1.0
#> [37] gtable_0.3.0      lifecycle_1.0.1   DBI_1.1.1         scales_1.1.1
#> [41] cli_3.2.0         stringi_1.7.5     cachem_1.0.6      farver_2.1.0
#> [45] fs_1.5.0          remotes_2.4.1     testthat_3.1.0    ellipsis_0.3.2
#> [49] ragg_1.2.0        generics_0.1.2    vctrs_0.4.0       tools_4.1.0
#> [53] glue_1.6.2        purrr_0.3.4       processx_3.5.3    pkgload_1.2.3
#> [57] fastmap_1.1.0     yaml_2.2.1        colorspace_2.0-3  sessioninfo_1.2.1
#> [61] memoise_2.0.1     usethis_2.1.5
date()
#> [1] "Sat Apr 09 22:25:29 2022"
```

The output is Output/Reports/Vignette_Report.doc. To permanently save, first go to file > Edit Links to Files; highlight all the figures (shift + scroll), and click "break link". Then File> save as, and esnure it is Saved As Type a "Word Document (*.docx)".

Other Outputs

We can use rmarkdown to create other versions of the main report or a graphical representation of the code architecture. You *may* need to use `Sys.setenv(RSTUDIO_PANDOC="C:/Program Files/RStudio/bin/pandoc")` before calling `render`, but this is not understood by the author at present as to why.

Numerous other outputs can be obtained from rmarkdown: slide show of the figures, reformatting of output as desired...

Code Tree

See the vignette `Code Tree Document` for an example that produces a seperate pdf document.

HTML version of the report

Your readers may prefer to get the report in a HTML document to view on-screen. This provides easier navigation with a floating toolbar. See the vignette `Vignette Report HTML`