Tips and Tricks for R Code Optimisation

➢ Optimising Code

Find biggest bottleneck and eliminate it;

➔ Profile code

```
Rprof("profiling.out")
## CODE ##
Rprof() summaryRprof("profiling.out")
```

When the bottleneck is found, try some of the following solutions;

1. **Look for existing solutions (online etc.)**

2. **Do less work**
   - Make functions do less work by tailoring functions to specific inputs/outputs
   - E.g. if possible avoid using `apply()` on a data frame (`apply()` first turns its input into a matrix)

3. **Vectorise**
   - Avoid loops
   - `rowSums(), colSums(), rowMeans(),` and `colMeans()`

4. **Parallelise**
   - Doesn't reduce computing time, but saves time by using more of a computer's resources

5. **Avoid copies**
   - If using `c(), append(), cbind(), rbind(),` or `paste()` to create a bigger object, R must first allocate space for the new object and then copy the old object to its new home. If repeating this many times, e.g. in a for loop, it can prove quite time-consuming

6. **Byte-code compile**
   - `compiler::cmpfun(function)`
   - Generally increases speed by 5 – 10% (base R slready byte-code compiled)

➢ Function Speed

`microbenchmark(function, function, unit = "eps")` – compares speed of functions, be sure to take note of the units used, unit = "eps" notes the number of evaluations needed to take 1 second.

`system.time()` – less precise than above

Avoid unnecessary arguments – additional arguments slow down the function

Memory

`pryr::object_size()` – accounts for shared elements within an object and includes size of environment

`pryr::mem_used()` – reports total size of all objects in memory

`gc()` – releases objects which are no longer used (R runs garbage collection automatically)

Loops

- Avoid appending to a vector/list with each pass of a loop. Instead, first create an empty vector/list of the correct and then fill its elements.
- Taking statements which check for conditions outside the loop greatly speeds up code.
- Run the loop only for TRUE conditions
- For example,

```
output <- character(nrow(df))
condition <- (df$col1 + df$col2 + df$col3 + df$col4) > 4
for (i in (1:nrow(df))[condition]) {   # run loop only for true conditions
if (condition[i]) {
output[i] <- "greater_than_4"
} else {
output[i] <- "lesser_than_4"
}
}
df$output
```

- Use `ifelse()` when possible
- Use `which()` to select rows
- Use `apply()` rather than for loops e.g. `apply(data, 1, FUN = function)`

Modification in place

Loops are notoriously slow as they modify a copy rather than modifying in place

Increase efficiency by using a list in this case, rather than a data frame. (lists modify in place)

Data Structures

Simpler data structures which only store one data type can be manipulated faster e.g. representing data in a matrix rather than a data frame speeds up code considerably.

data.table

This package speeds up operations on large data frames (particularly when sub-setting and indexing)

Parentheses

Avoid using unnecessary brackets – the more `( )` used the longer code takes to run

`{ }` run quicker than `( )` - `{ }` is treated as a special operator whose arguments aren’t automatically evaluated, unlike `( )` which is an inbuilt operator whose arguments are automatically evaluated

Note*: time saved minimal*

Readable Code

Code which is readable and reproducible can greatly save time

Google's R Style Guide is standard practice when coding in R

Further Reading

How to Speed Up R Code: An Introduction

Burns' R Inferno