Profiling to Reduce Running Time

Rachel

2023-01-08

In this report I will use the `microbenchmark()` function from the package of the same name to show how we can reduce the running time of inefficient code, using the example of computing t-test statistics.

Suppose we design an experiment collecting a result on 100 observations, randomly divided into two groups. We then repeat the experiment 1000 times with the aim of using a t test on the two groups.

Creating the samples

We first create a matrix with the appropriate data, the first 50 observations in each study corresponds to group 1 and the last 50 correspond to group 2

```
trials <- 1000

n <- 100

X <- matrix(rnorm(n*trials, mean = 4, sd = 1), nrow = trials)

groups <- rep(c(1,2), each = n/2)
```

Using the `t.test()` function

In loops

The `t.test()` function can be a standard S3 method, or a method for an object of class `formula`.

We then write two functions to create a vector of t-test statistics:

```
standard_t_test <- function(X, groups){
  trials <-  nrow(X)

  statistics <- vector(length = trials)

  for (i in 1:trials){
    statistics[i] <- t.test(X[i, groups ==1], X[i, groups ==2])$stat
  }

  return(statistics)
}
```

```
formula_t_test <- function(X, groups){
  trials <-  nrow(X)

  statistics <- vector(length = trials)

  for (i in 1:trials){
    statistics[i] <- t.test(X[i,] ~ groups)$stat
  }

  return(statistics)
}
```

We can compare how these perform using the `microbenchmark` package:

```
library(microbenchmark)
microbenchmark(standard_t_test(X, groups),formula_t_test(X, groups))
```

```
## Unit: milliseconds
##                         expr       min        lq      mean    median        uq
##  standard_t_test(X, groups)  50.65331  52.15398  53.88464  53.33563  54.22225
##   formula_t_test(X, groups) 247.10522 249.60911 253.11212 251.74237 253.36050
##       max neval
##   78.2080   100
##  280.2654   100
```

We can see here that the standard method is more efficient by a factor of about 5.

Using the `apply()` function

While the `standard_t_test()` function is more efficient, it still contains a loop which is typically quite slow. Thus we can write a function which replaces the loop with the `apply()` function and compare the running times

```
T.stat <- function(x, groups){
  t.test(x[groups == 1], x[groups == 2])$stat
}

apply_t_test <- function(X, groups){

  statistics <- apply(X,1, T.stat, groups = groups )
}


microbenchmark(standard_t_test(X, groups), apply_t_test(X, groups))
```

```
## Unit: milliseconds
##                         expr      min       lq     mean   median       uq
##  standard_t_test(X, groups) 50.76777 54.4197 57.04766 55.90736 58.12462
##     apply_t_test(X, groups) 52.20781 55.0954 56.91672 56.90788 58.64535
##       max neval
##  83.15682   100
##  62.71301   100
```

Reducing Computation

We can look at the source code for the `t.test()` function to see that as well as computing the test statistic, it produces a p value, a confidence interval and gives the output a format for printing. These computations are then taking up processing time, while being entirely useless for our purpose. We can then create our own function which will only compute and return the t-test statistic:

```
t.test.stat <- function(x, groups){
  t <- function(x){
    mu <- mean(x)
    n <- length(x)
    var <- sum((x - mu) ^ 2) / (n - 1)

    list(mu = mu, n = n, var = var)
  }

  t_1 <- t(x[groups =1])
  t_2 <- t(x[groups =2])

  error <- sqrt(t_1$var/t_1$n + t_2$var/t_2$n)

  return((t_1$mu - t_2$mu)/error)
}
```

We can then use the `apply()` function to compare this to the `t.test()` function

```
apply_t_stat <- function(X, groups){
  apply(X, 1, t.test.stat, groups = groups)
}

microbenchmark(standard_t_test(X, groups), apply_t_stat(X, groups))
```

```
## Unit: milliseconds
##                         expr       min        lq      mean    median        uq
##  standard_t_test(X, groups) 51.006025 52.889703 54.640280 53.875631 54.908085
##     apply_t_stat(X, groups)  5.968238  6.170647  6.968374  6.347256  7.965414
##       max neval
##  83.55544   100
##  10.83272   100
```

We can see a large improvement in the efficiency by only computing the t-test statistic. Finally we consider one other way to improve the execution time of the function.

Vectorisation

Many of the lines of code in `t.test.stat()` written for a vector, have vectorised versions that can be applied to matrices:

```
vectorised.t.test <- function(X, groups){
  t <- function(X) {
    mu <- rowMeans(X)
```

```
    n <- ncol(X)
    var <- rowSums((X - mu) ^ 2) / (n - 1)

    list(mu = mu, n = n, var = var)
  }

  t_1 <- t(X[,groups ==1])
  t_2 <- t(X[,groups ==2])

  error <- sqrt(t_1$var/t_1$n + t_2$var/t_2$n)

  return((t_1$mu - t_2$mu)/error)
}
```

Comparing this to it's non-vectorised counterpart, we see the improvement is significant (approximately by a factor of 10)

```
microbenchmark(apply_t_stat(X, groups), vectorised.t.test(X, groups))
```

```
## Unit: microseconds
##                          expr      min        lq      mean    median        uq
##       apply_t_stat(X, groups) 5900.457 6182.0010 6983.6653 6323.7660 8049.5820
##  vectorised.t.test(X, groups)  595.543  635.2705  758.5262  667.7525  700.1895
##       max neval
##  9916.763   100
##  5436.603   100
```

Scaling with number of trials

We can see the improvements from making the changes described in previous sections, but how do these differences in efficiencies evolve with the number of trials?

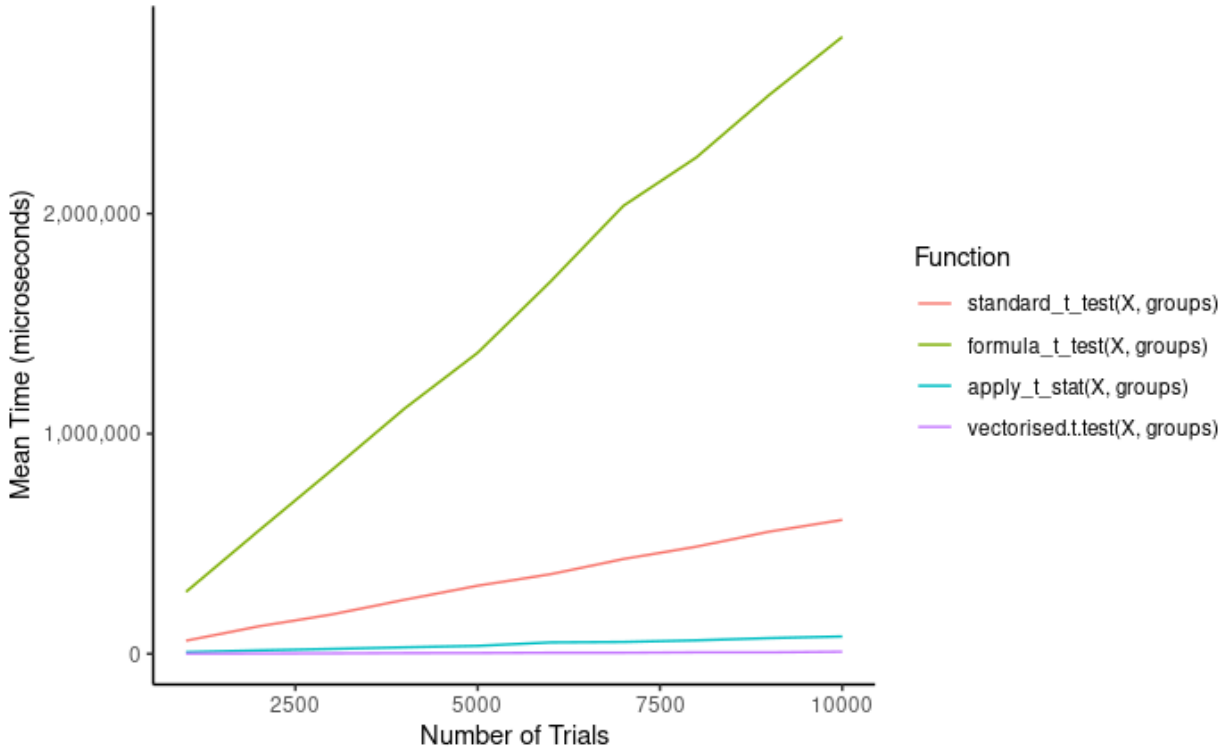
To answer this question, we can run the functions we've written for different number of rows of `X`.

```
library(tidyr)
library(dplyr)
library(ggplot2)



trials <- seq(1000, 10000, 1000)
n <- 100

tbl_colnames <- c("trials", "expr", "mean")
mean_times <- as_tibble(matrix(nrow = 0, ncol = length(tbl_colnames)), .name_repair = ~ tbl_colnames)

for (i in trials){

  X <- matrix(rnorm(n*i, mean = 4, sd = 1), nrow = i)

  groups <- rep(c(1,2), each = n/2)
```

```
  time <- microbenchmark(standard_t_test(X, groups), formula_t_test(X, groups), apply_t_stat(X, groups)
  time <- as_tibble(time)

  time <- time %>% group_by(expr) %>% summarise(mean = mean(time))
  time <- cbind(i, time)

  mean_times <- rbind(mean_times,time)
}

mean_times$mean <- mean_times$mean/1000
```

We can now plot how the mean time changes with the number of trials

```
library(scales)


ggplot(mean_times) +
  geom_line(aes(x = i, y =mean, colour = expr)) +
  labs(x = "Number of Trials", y = "Mean Time (microseconds)", color = "Function") +
  theme_classic() +
  scale_y_continuous(labels = scales::comma)
```



We can see from this plot, we get a minor decrease in the mean run time when vectorising our user-written function, and another decrease when we stop using the `t.test()` function and use our function which only computes the value of the statistic. However, we can see by-far the most drastic change we get is when the `t.test()` method for an object of class `formula` is used.