Common R Functions

Rachel

2022-10-23

This portfolio will detail the different ways we can make code faster and more efficient by eliminating loops as much as possible. We will first define a few functions containing loops, for which we will write more efficient versions. We can then compare the speed of these using the `system.time()` function.

The first function we will test is to sum the elements of a numeric matrix (we do this for emphasis, as the results for this are more dramatic than for a vector):

```
sum_loop <- function(x){
  count <- 0
  for (i in 1:nrow(x)){
    for (j in 1:ncol(x)){
      count <- count + x[i,j]
    }
  }
  return(count)
}
```

```
sin_loop <- function(x){
  sin_mat <- matrix(nrow = nrow(x), ncol = ncol(x))
  for (i in 1:nrow(x)){
    for (j in 1:ncol(x)){
      sin_mat[i,j] <- sin(x[i,j])
    }
  }
  return(sin_mat)
}
```

Vectorisation

Many base R functions are vectorised, meaning they will take a vector as input and will perform the computation on the vector elementwise, which will run much faster than if we were to use a loop. To see this, we can compare the run times for our `sum_loop()` function and the built-in `sum()` function on an arbitrary matrix `X`:

```
x <- matrix(1:1000000, nrow =1000)
system.time(sum_loop(x))
```

```
##    user  system elapsed
##   0.041   0.003   0.045
```

```
system.time(sum(x))
```

```
##    user  system elapsed
##       0       0       0
```

While there is a difference, it might be more obvious if we apply two nested functions, each containing a loop and two nested vectorised functions:

```
system.time(sum_loop(sin_loop(x)))
```

```
##    user  system elapsed
##   0.113   0.004   0.117
```

```
system.time(sum(sin(x)))
```

```
##    user  system elapsed
##    0.01    0.00    0.01
```

Apply Family of Functions

This class of built-in R functions

Parallel Programming