Chapter_6

Henry Bourne

2022-11-28

Performance and Bugs

When writing code we want it to "a) do its job" and "b) be efficient". In order to help achieve these two aims we will discuss in this chapter how to debug, increase performance and profile (check the time spent on pieces of code) in R.

Debugging

We will start by discussing **debugging** which is the process of removing **bugs** which are computations that lead to errors being thrown or unintended results. Often it is not immediately obvious where the bug is in your piece of code or how to fix it. A good process for debugging is as follows:

- Understand precisely the error, note that the symptom of the problem may not be entirely helpful for finding the source of the problem.
- Produce a minimal (least complexity as possible) working example of the unintended behavior.
  - At this stage you could submit a bug report, if this isn't your code.
- Identify the actual issue.
- Fix the bug and add tests so that this bug does not crop up again later.

Of course the best case scenario is no having to do any debugging! and on top of this write code that makes debugging easier! and we can try work towards this by employing good programming practices such as testing and breaking large tasks into smaller ones. However, we can't stop the inevitable, we will get bugs and the following tools can help us in removing them.

One such tool is the **traceback()** function (also graphically accessible in R) which will print out the call stack, the sequence of calls, that lead to the error. Often we find that the bug only crops up in certain scenarios and often we work with randomness which can make it difficult to isolate the problem (as sometimes it will arise and other times it won't). We can fix our pseudo-random number generator by calling **set.seed()** which can help us isolate the computation to a scenario where the bug pops up. Lastly we can use the **browser()** function, we use it by writing it before a piece of code we would like to evaluate in more detail. At run time when it gets to this part of the code then pause and an interactive window will pop up where we can type commands one of the following commands:

- 'n': Evaluates the next statement, stepping over function calls.
- 's': Evaluates the next statement, stepping into function calls.
- 'f': Finish execution of the current loop or function.
- 'where': Prints a stack trace of of all active function calls.
- 'c': Exits the browser and continues execution at the next statement.
- 'Q': Exits the browser and the current evaluation and returns to the top-level prompt.

When using *browser()* one can also type variable names in order to inspect their values.

Profiling

Now we will look at profiling which we use to tell us how much time we spend computing different lines or chunks of code, this information can then be useful when modifying our code to make it more efficient or often even in debugging. To profile in R we can use the **profvis** package, which is a **statistical profiler**, how it works is that it regularly (at fixed intervals) interrupts to check what code is currently being executed. This allows the profiler to build a picture of where most of the computation is taking place and using Monte Carlo methods we can give estimates for the time spent on each part of the code. The tradeoff here being that although it is not deterministic it adds very little overhead to computation. In contrast, an **instrumenting profiler** is deterministic and work by incrementing counters whenever certain events occur (eg. whenever a function is called), however, these incur significant computational overhead.

Performance

Now we will look at methods we can use to increase performance in R.

The first method we will talk about is using **optimized routines**. We note that code written in R will often fail to obtain as good performance as if we were to write the same code in a language like C. The main exception to this is when we delegate a task to optimized library code, the linear algebra operations are a good example of this (note that many of these packages are written in C).

Another thing we must be careful about in R is **memory management**. Note that R uses **pass by value** semantics (as opposed to **pass by reference**, where only a reference to the variable is passed around), this means that whenever you pass a variable around R will create a copy, which can obviously lead to drastic increases in memory. Therefore it is very important to keep this fact in mind when trying to increase the performance of your code. Do note, however, that there are certain hidden optimizations which R will perform in order to improve performance. For example, when you update elements in an array in a for loop R will not create a new copy of the array each time which can save allot of memory.

The last thing we will consider here is the way matrices are stored. R uses **column-major** storage (as opposed to **row-major** storage). What this means is that elements stored in a given column are stored in a contiguous block in memory. It is useful to keep this in mind when writing code, as operations done on a matrix by column will be much faster than by row, as reading and writing memory in contiguous blocks is much faster.

The last thing we will note is that it is also possible to directly write code in C++ and use it in R with the help of the **Rcpp** package.

An Example

Here we will look at an example where we can put into practice some of what we have learned above. Here we will try to build a binary logistic regression classifier. First we load my package stattools.

```
library(stattools)
# Can use "devtools::install_github("h-aze/compass_yr1", subdir = "/labs/stattools")" if
↪  not already downloaded
```

We now define a toy dataset that we will work on.

```
x <- c(1:10); x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
y <- c(c(rep(0,5), rep(1,5))); y
```

```
##  [1] 0 0 0 0 0 1 1 1 1 1
```

Now we will define the negative log likelihood function which we will use to perform our logistic regression.

```
neg_log_likelihood = function(par, D, y){
  D <- model_matrix(D)
  # D <- cbind(rep(1, nrow(D)), D )
  y_hat <-  rowSums(D %*% par)

  p <- sigmoid(y_hat)
  val <- -sum(y * log(p) + (1-y)*log(1-p))
  val
}
```

Now all we have to do is optimize this for our data to obtain our classifier:

```
results <- optim(par = c(0,0), fn = neg_log_likelihood, D=x, y=y); results
```

```
## $par
## [1] -45.155623   8.189242
##
## $value
## [1] 0.03327149
##
## $counts
## function gradient
##      135       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

We now define the function which will return us our prediction:

```
prediction = function(x, par){
  x <- model_matrix(x)
  # y_hat <- cbind(rep(1, nrow(D)), D )
  y_hat <-  rowSums(x %*% par)

  prob = sigmoid(y_hat)
  return(prob)
}
```

So using the parameter we found carrying out logistic regression we can now obtain predictions, eg.

```
prediction(5, results$par)
```

```
## [1] 0.01463764
```

```
prediction(6, results$par)
```

```
## [1] 0.981654
```

We see that it has correctly predicted that 5 belongs to class 0 and 6 to class 1.

Now what we would like to do is run profiling to check if there is any speed ups that we could do before we add these functions to the stattools package. I ran profvis on the below on a larger binary classification dataset and found that what took the most time was the model_matrix function and within that function the *as.matrix* function. Although this slows down performance we will not remove it as having it in matrix form will allow for speedups in other functions in the package as we can use highly optimized linear algebra operations.

```
library("mlbench")
data(Sonar)
x <- Sonar[,1:ncol(Sonar)-1]
y <- ifelse(Sonar[,ncol(Sonar)] == 'R', 0, 1)
optim(par = rep(1,ncol(x)+1), fn = neg_log_likelihood, D=x, y=y)
```

```
## $par
##  [1] 0.7492001 1.0315917 1.0287473 1.0257202 1.0190561 1.0153855 1.0027945
##  [8] 0.9980014 1.0870092 0.9585158 0.9749244 1.1936793 1.0138659 1.0112794
## [15] 1.0086927 1.0061078 1.0012263 0.9959472 0.9946673 0.9176609 0.9829204
## [22] 0.9709072 0.9684218 0.9624448 0.9590680 0.9565402 0.9480923 0.9506697
## [29] 0.9540076 0.9735285 0.9668251 0.9797778 0.9934340 0.9971694 0.9959131
## [36] 0.9922156 0.9910148 0.9984342 1.0025007 1.0048171 1.0075377 1.0073998
## [43] 1.0099860 1.0125727 1.0151587 0.9814371 0.9928721 1.0072749 1.0114650
## [50] 1.0224952 1.0342678 1.0367888 1.0391673 1.0425497 1.0444951 1.0463462
## [57] 1.0479362 1.0437847 0.9698176 1.0474587 1.0497396
##
## $value
## [1] 1616.873
##
## $counts
## function gradient
##      502       NA
##
## $convergence
## [1] 1
##
## $message
## NULL
```