Exploration of Compiler in R

Michael Frasco

2/20/2018

```r
library(compiler)
library(microbenchmark)
```

The esteemed James Lamb pointed me to this book, which has a section on an R package that can compile a function to byte code, providing speed improvements. It was shown that compiling works well for functions with obvious inefficiencies (e.g. looping over a vector of numbers to calculate the mean).

However, I wanted to throw two harder use cases at the compiler to see what else it can do. First, I wanted to test it on an function that is already vectorized and implemented in C. Second, I wanted to test it on a function that uses a for-loop but doesn't have an obvious way to make it faster.

For an instance of the first function, I am using an implementation of the function that calculates the area under the ROC curve using the Mann-Whitney algorithm.

```r
auc <- function(preds, target) {
    r <- rank(preds)
    num_pos <- as.numeric(sum(target == 1))
    num_neg <- length(target) - num_pos
    return((sum(r[target == 1]) - num_pos * (num_pos + 1) / 2) / (num_pos * num_neg))
}

cmp_auc <- cmpfun(auc)
```

Here are the timings

```r
preds <- c(rbeta(1e6, 25, 30), rbeta(1e6, 30, 25))
target <- rep(c(0, 1), each = 1e6)

microbenchmark(regular = auc(preds, target)
               , compiled = cmp_auc(preds, target)
               , times = 10)
```

```
## Unit: seconds
##      expr      min       lq     mean   median       uq      max neval
##   regular 1.858168 1.899046 1.973502 1.971444 2.057331 2.075903    10
##  compiled 1.840615 1.905961 1.957142 1.928976 1.996733 2.158868    10
```

There is no significant difference between these functions.

In the second example, I wrote a very complicated function that doesn't do anything interesting. However, it is a function that cannot be vectorized, due to the fact that the random data generated in one iteration depends on the results of a linear regression from the previous iteration.

```r
crazy_loop <- function(num_iters = 1e3, num_points = 1e3) {
    data_mean <- rnorm(1); data_sd <- rgamma(1, 1)
    results <- rep(0, num_iters)
    for (i in seq_along(results)) {
        random_x <- rnorm(num_points, data_mean, data_sd)
        random_y <- rnorm(num_points, data_mean, data_sd)
        model <- lm(random_y ~ random_x)
        data_mean <- summary(model)$r.squared
        data_sd <- summary(model)$coefficients[2,4]
        results[i] <- sum(coef(model))
    }
    return(max(results))
}

cmp_crazy_loop <- cmpfun(crazy_loop)
```

And here are the timings for this function

```
microbenchmark(regular = crazy_loop()
               , compiled = cmp_crazy_loop()
               , times = 10)
```

```
## Unit: seconds
##      expr      min       lq     mean  median       uq      max neval
##   regular 1.824623 1.872153 1.922704 1.91309 1.938819 2.122003    10
##  compiled 1.810192 1.854254 1.929331 1.89083 1.969473 2.184451    10
```

Once again, there is no significant improvement. I wasn't expecting that the compiler would be able to improve either of these functions. I didn't think that the compiler was some magic black box that would make all R code faster.

However, it is clear that it can improve some code. And it is always exciting to learn about the different tools available within R.