Stat 5014 HW6

Bob Settlage

2017-10-17

Problem 2: Sums of Squares

In this problem, we are to compare traditional for loop style computing of sums of squares (SST) to the same computation using vector operations. The comparison is in timing and code pretty-ness. The data using in this problem is generated and included in the following code.

```
# generate the data
set.seed(12345)
y <- seq(from = 1, to = 100, length.out = 100000000) + rnorm(100000000)

## Part a: for loop
SST <- 0
t1 <- system.time({
    y_bar <- mean(y)
    for (i in 1:100000000) {
        SST <- SST + (y[i] - y_bar)^2
    }
})

## Part b: vector operations
SST <- 0
t2 <- system.time({
    y_bar <- mean(y)
    SST <- t(y - y_bar) %*% (y - y_bar)
})
```

Times for the "for loop" method and the vectorized method are respectively. I had initially included the mean(y) inside the square term in the loop function. This was taking WAY to long, so I precalculated that saving a ton of redundant calculation of $\bar{y}$.

```
## Part b: vector opertations
SST <- 0
t3 <- system.time({
    y_star <- (y - mean(y))
    SST <- crossprod(y_star)
})
```

This one change resulted in a new time for the vectorized method of giving a -60.279617% reduction.

Problem 3: Dual nature for speed

In this problem, we are being asked to code up a gradient descent algorithm, again using for loop and matrix operations comparing the timings and code cleanliness.

First, the for loop version of gradient descent:

```
# generate the data
set.seed(1256)
theta <- as.matrix(c(1, 2), nrow = 2)
```

```r
X <- cbind(1, rep(1:10, 10))
h <- X %*% theta + rnorm(100, 0, 0.2)

theta0_current <- 0
theta0_new <- 1
theta1_current <- 0
theta1_new <- 1
alpha <- 0.0001
tolerance <- 0.000001
m <- length(h)

# could probably do better by: a. do both updates in the
# same loop OR b. use the new theta0 in the theta1 loop
t4 <- system.time({
    while (abs(theta0_new - theta0_current) > tolerance &
        abs(theta1_new - theta1_current) > tolerance) {
        theta0_current <- theta0_new
        theta1_current <- theta1_new
        theta0_grad <- 0
        for (i in 1:m) {
            theta0_grad <- theta0_grad + theta0_current +
                theta1_current * X[i, 2] - h[i]
        }
        theta0_new <- theta0_current - alpha/m * theta0_grad
        theta1_grad <- 0
        for (i in 1:m) {
            theta1_grad <- theta1_grad + theta0_current +
                (theta1_current * X[i, 2] - h[i]) * X[i,
                  2]
        }

        theta1_new <- theta1_current - alpha/m * theta1_grad
    }
})
```

```r
# generate the data
set.seed(1256)
theta <- as.matrix(c(1, 2), nrow = 2)
X <- cbind(1, rep(1:10, 10))
h <- X %*% theta + rnorm(100, 0, 0.2)

theta_current <- as.matrix(c(0, 0), nrow = 2)
theta_new <- as.matrix(c(1, 1), nrow = 2)
alpha <- 0.0001
tolerance <- 0.000001
m <- length(h)

tX <- t(X)
t5 <- system.time({
    while (sum(abs(theta_new - theta_current) > tolerance)) {
        theta_current <- theta_new
        theta_grad <- tX %*% ((X %*% theta_current) - h)
        theta_new <- theta_current - alpha/m * theta_grad
    }
```

```
})
```

In this case, we greatly improved the code readibility. We did not see an improvement in speed and in fact are a bit slower: 0.352 vs 0.159 for matrix and for loops respectively.

Problem 5:

Here the goal is to compute a set of vector/matrix operations quickly. As a reminder, the operation we are to compute is:

$$y = p + AB^{-1}(q - r) \tag{1}$$

Without any improvements, a single iteration (i.e. single randomly populated B matrix) takes 10-20 min. Without going through the optimization strategy, here is the best I came up with:

```
################################### Tian's code to make the matrices
set.seed(12456)  #not currently doing parallel stuff, so probably don't need parRNG
# G: 1600 * 10 matrix, elements can take three values:
# 0, 0.5 and 1 id: vector of length 932 -- 932 random
# indexes in G_c (vectorized G) p: vector of length 932
# r: vector of length 15068

G <- matrix(sample(c(0, 0.5, 1), size = 16000, replace = T),
    ncol = 10)
R <- cor(G)  # R: 10 * 10 correlation matrix of G
C <- kronecker(R, diag(1600))  # kronecker product, C is a 16000 * 16000 block diagonal matrix
# G_c <- as.vector(G) # vectorized G with length of
# 16000 -- concatenate columns of G
id <- sample(1:16000, size = 932, replace = F)
q <- sample(c(0, 0.5, 1), size = 15068, replace = T)  # vector of length 15068
A <- C[id, -id]  # matrix of dimension 932 * 15068
B <- C[-id, -id]  # matrix of dimension 15068 * 15068, still a block diagonal matrix, sparse
p <- runif(932, 0, 1)
r <- runif(15068, 0, 1)
C <- NULL

qr <- q - r  # not really part of the compute as updates are to B
Bsp <- Matrix(B)  #if more than 1/2 entries are zero, cast as sparse = default
Asp <- Matrix(A)  #probably don't need to do this, but it doesn't hurt
t6 <- system.time(z4 <- p + Asp %*% solve(Bsp, tol = 1e-19) %*%
    qr)
```

There was a question on the structure of B. B is a 15068 x 15068 matrix with 1 along the diagonal and 10 x 10 blocks along the diagonal with off diagonal elements as correlation values. Here is a tabulation of the elements in the matrix. There should only be 100 different elements, so we can do this. ;)

```
## quick view of diagonal elements
table(diag(B))
```

```
##
##     1
## 15068
```

```r
## quick view of all elements
table(B)
```

```
## B
##   -0.0593434916659995   -0.0508276241929542   -0.0437017561156568
##                  2838                  2836                  2850
##    -0.038750409534823   -0.0305274493319447   -0.0273326680743183
##                  2846                  2846                  2840
##   -0.0227720797133449   -0.0225580436431903   -0.0197143753612597
##                  2838                  2778                  2786
##   -0.0191550046699806   -0.0172044735203295   -0.0165529818525149
##                  2818                  2864                  2826
##   -0.0112268117618719  -0.00999895750295145  -0.00992443838738708
##                  2834                  2880                  2852
##  -0.00570913833279853  -0.00350732978899381  -0.00157798603174804
##                  2890                  2832                  2840
## -0.000735565416207467 -0.000307894947901916                     0
##                  2830                  2878             226901840
##   0.00315488431358059   0.00440492064485365   0.00519764852786881
##                  2850                  2856                  2862
##    0.0120457779053466    0.0139231038084424    0.0156534798664632
##                  2798                  2860                  2790
##    0.0164999934196776    0.0178918023080869     0.019154353130133
##                  2846                  2814                  2832
##    0.0200073730433835    0.0207074777999135    0.0232835244473396
##                  2816                  2876                  2806
##    0.0234085689443415    0.0274379741101879    0.0276659540361215
##                  2830                  2820                  2876
##    0.0328079464072214    0.0350875434242563    0.0358178330927301
##                  2866                  2850                  2834
##    0.0377559526370284    0.0382181912683319      0.04263912518562
##                  2836                  2824                  2860
##    0.0499500699922705     0.052335886671229    0.0596711135806099
##                  2860                  2826                  2806
##    0.0709580851423905                     1
##                  2820                 15068
```