Homework 6

Ryan Kingery

10/11/2017

Problem 2

In this problem we analyze the speed gain potential from using vectorized operations as opposed to using loops. To do so, let's define a large vector y of size 10e8.

```
set.seed(12345)
y <- seq(from = 1, to = 100, length.out = 1e+08) + rnorm(1e+08)
ybar <- mean(y)
```

The goal is to calculate the sum of squares error (SSE) of y

$$SSE = \sum_i (y_i - \overline{y})^2.$$

We first do this using a loop. The following code calculates the SSE via a for loop while keep track of processing time.

```
system.time({
    sse_loop <- 0
    for (i in 1:length(y)) {
        sse_loop <- sse_loop + (y[i] - ybar)^2
    }
})
```

```
##    user  system elapsed
##   9.210   0.110   9.723
```

Next, we compute the SSE using vectorization. The following code calculates the SSE via this method while keeping track of processing time.

```
system.time({
    sse_vect <- sum((y - ybar)^2)
})
```

```
##    user  system elapsed
##   0.934   1.097   3.337
```

As we can see in comparing the two times, the vectorized method offers a rough speed up of about a factor of 10. Note: a better way to do this would probably involve doing this over many iterations and averaging the processing times, but my computer is too slow to do this.

Last, it's worth noting that the SSE for each of the methods do indeed equal, at least to machine precision:

```
sse_loop
```

```
## [1] 81774703375
```

```
sse_vect
```

```
## [1] 81774703375
```

Problem 3

The goal now is to analyze performance differences in two different implementations of gradient descent for a simple linear regression problem. The following code initializes the parameters in the problem.

```
set.seed(1256)
tol = 1e-05
alpha = 0.01
theta_prev <- as.matrix(rnorm(2, 0, 1), nrow = 2)
theta <- as.matrix(c(1, 2), nrow = 2)
X <- cbind(1, rep(1:10, 10))
y <- X %*% theta + rnorm(100, 0, 0.2)
m <- length(y)
```

The first implementation uses a loop that terminates when the iterates stop changing substantially according to some pre-defined tolerance, in our problem we take a tolerance tol=10e-6. For the initial parameter vector θ^0 we sample from a standard Gaussian. For the learning rate we take $alpha = 0.01$. As before, we keep track of processing time.

```
system.time({
    while ((abs(theta[1, 1] - theta_prev[1, 1]) & abs(theta[2, 1] - theta_prev[2,
        1])) > tol) {
        theta_prev <- theta
        theta[1, 1] <- theta[1, 1] - alpha * 1/m * sum((theta[1, 1] + theta[2,
            1] * X[, 2] - y))
        theta[2, 1] <- theta[2, 1] - alpha * 1/m * sum((theta[1, 1] + theta[2,
            1] * X[, 2] - y) * X[, 2])
    }
})
```

```
##    user  system elapsed
##   0.181   0.045   0.237
```

```
theta
```

```
##           [,1]
## [1,] 0.9630442
## [2,] 2.0027616
```

Next, we implement the same algorithm using the lm function in R, again keeping track of processing time.

```
system.time({
    lm(y ~ 0 + X)
})
```

```
##    user  system elapsed
##   0.006   0.001   0.009
```

First, observe that the two values for the vector θ do indeed match to machine precision. Next, observe that the run time for the lm function is significantly faster than for gradient descent. I assume the lm function is implemented using some kind of optimized quasi-Newton scheme that outperforms gradient descent for linear models with a squared error function like this.

Problem 4

Consider the least squares solution $\hat{\beta} = (X^T X)^{-1} X^T y$. While this equation is mathematically true it would be egregious to solve for $\hat{\beta}$ this way on a computer. This is due to both computation time and the fact that matrix inversion is an often unstable operation (especially on ill-conditioned matrices). Moreover, inversion doesn't preserve the sacrosanct property of sparcity. Thus, the better way to solve for $\hat{\beta}$ is to first uninvert the equation to get $X^T X\hat{\beta} = X^T y$. This is a simple matrix solve that can be done stably and quickly using any standard solve function.

Problem 5

The following code initializes the parameters of the problem.

```
set.seed(12456)
G <- matrix(sample(c(0, 0.5, 1), size = 16000, replace = T), ncol = 10)
R <- cor(G)
C <- kronecker(R, diag(1600))
id <- sample(1:16000, size = 932, replace = F)
q <- sample(c(0, 0.5, 1), size = 15068, replace = T)
A <- C[id, -id]
B <- C[-id, -id]
p <- runif(932, 0, 1)
r <- runif(15068, 0, 1)
C <- NULL
```

From below, we see calculating the size of A yields about 100 MB, and the size of B yields about 1.8 GB.

```
object.size(A)
```

```
## 112347208 bytes
```

```
object.size(B)
```

```
## 1816357192 bytes
```

Last, calculating $y = p + AB^{-1}(q - r)$ the idiotic way evidently takes what might as well be an infinite amount of time on my computer, so I won't post that attempt. Instead, I define a vector $v = B^{-1}(q - r)$, solve $Bv = q - r$ for v, and then finally get y as $y = p + Av$.

```
# system.time({ v <- solve(B,q-r) y <- p+A%*%v })
```

I would attempt to report on the run time, but my R session keeps freezing when trying to process this code, even when trying different implementations. Since every such implementation is causing me the same problems, I can't really say much more about what will optimize this matrix solve.