Applied Statistical Programming - Error Handling

Berta Diaz, Zion Little, Alma Velazquez

2/21/2022

Write the R code to answer the following questions. Write the code, and then show what the computer returns when that code is run. Thoroughly comment your solutions.

You have until the beginning of class 2/23 at 10:00am to complete the assignment below. You may use R, but not any online R documentation. Submit the Rmarkdown and the knitted PDF to Canvas. Have one group member submit the activity with all group members listed at the top.

Dang it, Bobby

Your summer intern Bobby was tasked with writing a function. Bobby's internship ended, and he couldn't get the code working in time. Unfortunately, Bobby barely commented his code, and you need to fix the problem before Wednesday's code review at 10:00am.

Your task is to trace errors and generate error handling statements to get the function working as intended. You will not be evaluated based on matching output between your fixed code and other existing implementations. Instead, you will be evaluated based on how well you can convince your absentee project manager that you genuinely fixed Bobby's code.

A solution to this in class activity will include the following modifications to Bobby's code:

1. print statements at different stages of the algorithm to verify correct output,
2. error handling that prevents a user from passing the wrong type of matrix to the function, and
3. commenting for each step of the algorithm.

QR Factorization

Bobby used the Gram-Schmidt process to perform a QR factorization of a matrix A.[1] A QR factorization of a matrix A satisfies $A = QR$ where Q is an orthogonal matrix and R is upper triangular.

> Definition: A matrix Q is orthogonal if and only if $Q^\intercal Q = QQ^\intercal = I$.
>
> Definition: A matrix R is upper triangular if and only if all entries below the main diagonal of R are all zero.

Knowing a matrix has a QR factorization is useful for two reasons. First, the matrix Q being orthogonal provides computational savings because $Q^\intercal = Q^{-1}$. Transposing a matrix is considerably cheaper than inverting one. We are often not interested in $A = QR$ by itself, but instead another product $PAX = PQRX$ where P reduces nicely with Q and R reduces nicely with X.

The second reason that a QR factorization is useful is because it reduces the number of calculations needed to perform matrix operations. To see how, first consider naively multiplying two $n \times n$ matrices together. Each entry in a matrix $A = BC$ is the inner product of a row of length n from B and a column of length

[1] That boy ain't right. Never, **ever** use textbook Gram-Schmidt to invert or factor a matrix. It is both inefficient and unstable without modifications. However, it is still an excellent tool for teaching.

n from C. There are thus n^2 inner products, and each inner product requires n multiplications. So naive multiplication of two $n \times n$ matrices takes n^3 operations.

Suppose instead it is known that R in $A = QR$ is upper triangular. Then the inner product of the ith row of Q and the jth column of R only requires $n - (j-1)$ multiplications. This is because $j-1$ of entries from the jth column of R are zero. Not requesting the computer to make a calculation that is known to be zero adds up to tremendous computational savings, especially for large matrices.

Bobby's function `gramschmidt` is provided below. You can verify the code is broken by generating a square matrix A and comparing the output of `qr(A)` with `gramschmidt(A)`. Use the following QR factorization algorithm for an $n \times n$ matrix $A = [a_1|a_2|\dots|a_n]$ to help you identify where the function isn't working.[2]

- $u_1 = a_1$, $e_1 = u_1/\|u_1\|$
- $u_2 = a_2 - (a_2^\intercal e_1) * e_1$, $e_2 = u_2/\|u_2\|$
- $u_3 = a_3 - (a_3^\intercal e_1) * e_1 - (a_3^\intercal e_2) * e_2$, $e_3 = u_3/\|u_3\|$
- $\dots$
- $u_n = a_n - \sum_{j=1}^{n-1} (a_n^\intercal e_j) * e_j$, $e_n = u_n/\|u_n\|$

The matrices Q and R are built in the following way.

$$Q = [e_1|e_2|\dots|e_n]$$

$$R = \begin{bmatrix} a_1^\intercal e_1 & a_1^\intercal e_2 & a_1^\intercal e_3 & \dots a_1^\intercal e_n \\ 0 & a_2^\intercal e_2 & \dots & a_2^\intercal e_n \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & a_n^\intercal e_n \end{bmatrix}$$

```
# Remove eval=FALSE from this code block to have it run.
gramschmidt <- function(x) {

    # Add some input checks
    criteria <- is.matrix(x) & det(x) != 0

    # Error if inputs not as expected
    if (!criteria) {
        stop("Error: x must be a nonsingular matrix")
    }

    # Get the number of rows and columns of the matrix
    n <- ncol(x)
    m <- nrow(x)

    # Initialize matrices Q and R
    Q <- matrix(0, m, n)
    R <- matrix(0, n, n)

    # Gram-Schmidt process
    for (j in 1:n) {
        v = x[, j]
        for (i in 1:j) {
            R[i, j] <- t(Q[, i]) %*% x[, j]
            v <- v - (R[i, j] * Q[, i])
```

[2] See https://www.math.ucla.edu/~yanovsky/Teaching/Math151B/handouts/GramSchmidt.pdf for a numerical example.

```
        }

        # Q[,j] <- v / sqrt(sum(v^2)) R[j,j] <- t(v) * Q[,j]

        R[j, j] <- -1 * sqrt(sum(v^2))
        Q[, j] <- v/R[j, j]

        # print(R[j,j]) print(t(x[,j] )%*% Q[,j])

    }

    # Return matrices Q and R in a list
    QRdecomp <- list(Q = Q, R = R)
    return(QRdecomp)
}
```

Changes Made

- The outer loop indexes each column from 1 to n; previously indexing was messed up, from 1:n-1
- Removed an unnecessary if statement
- To match the output of qr(), multiplied entries in R by -1
- Added input criteria, namely that the matrix X be of type 'matrix' and composed of linearly independent columns

Testing Bobby's Function

```
A <- matrix(sample(10,9), 3,3)
A
```

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

```
gramschmidt(A)
```

```
## $Q
##            [,1]       [,2]       [,3]
## [1,] -0.3980149  0.9102834  0.1138784
## [2,] -0.6965260 -0.2190724 -0.6832707
## [3,] -0.5970223 -0.3512712  0.7212301
##
## $R
##           [,1]       [,2]      [,3]
## [1,] -10.04988 -13.532506 -7.164268
## [2,]   0.00000  -2.621314 -1.544838
## [3,]   0.00000   0.000000 -3.909827
```

Functions below provide R's QR factorization of matrix A

```
qr.Q(qr(A))
```

```
##            [,1]       [,2]       [,3]
## [1,] -0.3980149  0.9102834  0.1138784
## [2,] -0.6965260 -0.2190724 -0.6832707
```

```
## [3,] -0.5970223 -0.3512712  0.7212301
```

```
qr.R(qr(A))
```

```
##           [,1]       [,2]      [,3]
## [1,] -10.04988 -13.532506 -7.164268
## [2,]   0.00000  -2.621314 -1.544838
## [3,]   0.00000   0.000000 -3.909827
```

Below should give us the matrix R

```
solve(gramschmidt(A)$Q) %*% A
```

```
##               [,1]          [,2]      [,3]
## [1,] -1.004988e+01 -1.353251e+01 -7.164268
## [2,] -4.440892e-16 -2.621314e+00 -1.544838
## [3,]  1.776357e-15  1.776357e-15 -3.909827
```

The product of Q and R should give back A

```
gramschmidt(A)$Q %*% gramschmidt(A)$R
```

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

```
A
```

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

Test a singular matrix

```
test_mat <- matrix(c(1,2,2,4),2,2)
gramschmidt(test_mat)
```

Gives error as desired