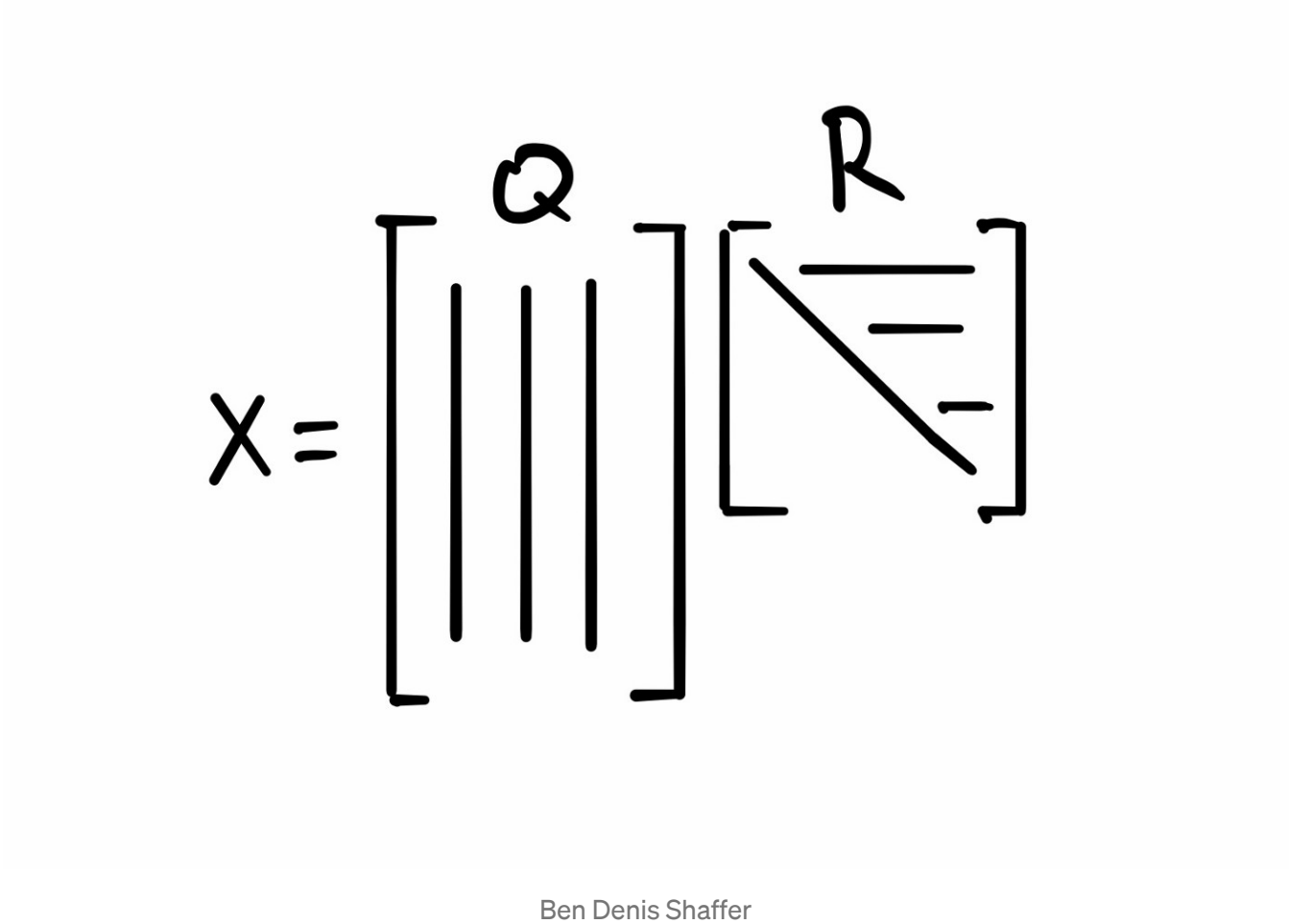
QR Matrix Factorization

Least Squares and Computation (with R and C++)

Ben Denis Shaffer Feb 27, 2020 · 9 min read ★



Ben Denis Shaffer

Data Science and Matrix Factorizations

methods for actually computing and estimating results for the models and algorithms we use. In some cases, a particular form of factorization is the algorithm (ex. PCA and SVD). In all cases, matrix factorizations help develop intuition and the ability to be analytical.

The ***QR*** factorization is one of these matrix factorizations that is very useful and has very important applications in Data Science, Statistics, and Data Analysis. One of these applications is the computation of the solution to the Least Squares (LS) Problem.

Agenda

- Recap the Least Squares Problem
- Introduce the QR matrix factorization
- Solve the LS using QR
- Implement the QR computation with R and C++ and compare.

. . .

The LS Problem

The ***QR*** matrix decomposition allows us to *compute* the solution to the Least Squares problem. I emphasize *compute* because OLS gives us the closed from solution in the form of the normal equations. That is great, but when you want to find the actual numerical solution they aren't really useful.

Here is a recap of the Least Squares problem. We want to solve the equation below

$$X\beta = Y$$

The problem is that we can't solve for $\boldsymbol{\beta}$ because usually if we have more observations than variables $\boldsymbol{X}$ doesn't have an inverse and the following can't be done:

Instead, we try to find some $\hat{\boldsymbol{\beta}}$ that solves this equation, not perfectly, but with as little error as possible. One way to do that is to minimize the following objective function, which is a function of $\hat{\boldsymbol{\beta}}$.

$$\sum(Y - \hat{Y})^2 = \sum(Y - X\hat{\beta})^2$$

Minimizing this sum of *squared* deviations is why the problem is called the *Least Squares* problem. Taking derivatives with respect to $\hat{\boldsymbol{\beta}}$ and setting to zero will lead you to the normal equations and provide you with a closed-form solution.

That is one way to do it. But we could also just use Linear Algebra. This is where the ***QR*** matrix decomposition comes in and saves the day.

. . .

QR Factorization

First, let's just go ahead and describe what this decomposition is. The ***QR*** matrix decomposition allows one to express a matrix as a product of two separate matrices, ***Q***, and ***R***.

$$X = QR$$

This means that

$$Q^T Q = QQ^T = I \implies Q^T = Q^{-1}$$

And since $\boldsymbol{R}$ is square, as long as the diagonal entries don't have a zero, it is also invertible. If columns of $\boldsymbol{X}$ are linearly independent then this will always be the case. Though if there is collinearity in the data then problems can still arise. That aside though, what this $\boldsymbol{QR}$ factorization implies is that a rectangular and non-invertible $\boldsymbol{X}$ can be expressed as two invertible matrices! This is bound to be useful.

. . .

Solving the LS problem Using the QR factorization.

Now that we know about the $\boldsymbol{QR}$ factorization, once we can actually find it, we will be able to solve the LS problem in the following way:

$$X\beta = QR\beta = y$$

so

$$\hat{\beta} = (QR)^{-1}y = R^{-1}Q^T y$$

This means that all we need to do is find an inverse of $\boldsymbol{R}$, transpose $\boldsymbol{Q}$, and take the product. That will produce the OLS coefficients. We don't even need to compute the

Performing the QR factorization.

The way to find the ***QR*** factors of a matrix is to use the Gram-Schmidt process to first find ***Q***. Then to find ***R*** we just multiply the original matrix by the transpose of ***Q***. Let's go ahead and do the ***QR*** using functions implemented in `R` and `C++`. Later we can look inside these functions to get a better picture of what is going on.

Compute the Coefficients

I am loading in two functions. `myQRR` and `myQRCpp` that use the Gram-Schmidt process to do the ***QR*** factorization. One function is written in `R` and the other in `C++` and loaded into the `R` environment via `Rcpp`. Later I will compare their performance.

```
library(Rcpp)
library(tidyverse)
library(microbenchmark)

sourceCpp("../source-code/myQRC.cpp")
source("../source-code/myQRR.R")
```

Let's begin with a small example where we simulate **y** and **X** and then solve it using the ***QR*** decomposition. We can also double check that the ***QR*** decomposition actually works and gives back the ***X*** we simulated.

Here is our simulated response variable.

```
y = rnorm(6)
y

## [1] 0.6914727 2.4810138 0.4049580 0.3117301 0.6084374 1.4778950
```

Here is the data that we will use to solve for the LS coefficients. We have 3 variables at our disposal.

```
X = matrix(c(3, 2, 3, 2, -1, 4,
             5, 1, -5, 4, 2, 1,
```

```
##      [,1] [,2] [,3]
## [1,]    3    5    9
## [2,]    2    1   -3
## [3,]    3   -5    2
## [4,]    2    4   -1
## [5,]   -1    2    8
## [6,]    4    1    1
```

Now I will use the `myQRCpp` to find the `Q` and the `R`.

1. You can see that ***R*** is indeed upper triangular.

```
Q = myQRCpp(X)$Q
R = t(Q) %*% X %>% round(14)

R

##          [,1]     [,2]      [,3]
## [1,] 6.557439 1.829983  3.202470
## [2,] 0.000000 8.285600  4.723802
## [3,] 0.000000 0.000000 11.288484

Q

##            [,1]        [,2]        [,3]
## [1,]  0.4574957  0.50241272  0.45724344
## [2,]  0.3049971  0.05332872 -0.37459932
## [3,]  0.4574957 -0.70450052  0.34218986
## [4,]  0.3049971  0.41540270 -0.34894183
## [5,] -0.1524986  0.27506395  0.63684585
## [6,]  0.6099943 -0.01403387 -0.07859294
```

2. Here we can verify that **Q** is in fact Orthogonal.

```
t(Q)%*%Q %>% round(14)

##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

3. And that ***QR*** really does give back the original ***X*** matrix.

```
##      [,1] [,2] [,3]
## [1,]    3    5    9
## [2,]    2    1   -3
## [3,]    3   -5    2
## [4,]    2    4   -1
## [5,]   -1    2    8
## [6,]    4    1    1
```

Now, let's compute the actual coefficients.

```
beta_qr = solve(R) %*% t(Q) %*% y

beta_qr

##             [,1]
## [1,]  0.32297414
## [2,]  0.07255123
## [3,] -0.02764668
```

To check that this is the correct solution we can compare the computed $\hat{\boldsymbol{\beta}}$ to what the `lm` function gives us.

```
coef(lm(y ~ -1 + ., data = data.frame(cbind(y,X))))

##          V2          V3          V4
##  0.32297414  0.07255123 -0.02764668
```

Clearly we get the exact same solution for the estimated coefficients.

. . .

Implementing the QR

The Gram–Schmidt process

The *Gram–Schmidt process* is a method for computing an orthogonal matrix $\boldsymbol{Q}$ that is made up of orthogonal/independent unit vectors and spans the same space as the original matrix $\boldsymbol{X}$.

- Then we find a vector orthogonal to ***u1*** by projecting the next column of ***X***, say **x2** onto it and, subtracting the projection from it ***u2 = x2 − proj u1x2***. Now we have a set of two orthogonal vectors. In a previous post, I covered the details of why this works.
- The next step is to proceed in the same way but subtract the sum of projections onto each vector in the set of orthogonal vectors ***uk***.

We can express this as follows:

$$u_1 = x_1$$

$$u_k = x_k - \sum_{j=1}^{k-1} proj_{u_j} x_k$$

ref. Once we have the full set of orthogonal vectors we simply divide each by its norm and put them in a matrix:

$$Q = [\frac{u_1}{||u_1||}, \frac{u_1}{||u_1||}, \ldots, \frac{u_m}{||u_m||}]$$

Once we have ***Q*** we can solve for ***R*** easily

$$R = Q^{-1}X = Q^{T}X$$

Implementing in R and C++

Of course, there is a built-in function in `R` that will do the ***QR*** matrix decomposition for you. Because the GS algorithm above is iterative in nature I decided to implement it in `C++` which is a good tool for something like this, and compare it to an equivalent `R` function. Here is what my `R` version looks like:

```
myQR = function(A){
  dimU = dim(A)
  U = matrix(nrow = dimU[1], ncol = dimU[2])
  U[,1] = A[,1]
  for(k in 2:dimU[2]){
    subt = 0
    j = 1
    while(j < k){
      subt = subt + proj(U[,j], A[,k])
      j = j + 1
    }
    U[,k] = A[,k] - subt
  }
  Q = apply(U, 2, function(x) x/sqrt(sum(x^2)))
  R = round(t(Q) %*% A, 10)
  return(list(Q = Q, R = R, U = U))
}
```

It is very literal. There is a while loop inside a for loop, and the projection function being called is also a function written in `R`.

This is what my `C++` version looks like. The logic is essentially the same except there is another for loop to normalize the orthogonal columns.

```
// [[Rcpp::export]]
List myQRCpp(NumericMatrix A) {
  int a = A.rows();
```

```
  NumericMatrix R(a,b);
  NumericMatrix::Column Ucol1 = U(_ , 0);
  NumericMatrix::Column Acol1 = A(_ , 0);

  Ucol1 = Acol1;

  for(int i = 1; i < b; i++){
    NumericMatrix::Column Ucol = U(_ , i);
    NumericMatrix::Column Acol = A(_ , i);
    NumericVector subt(a);
    int j = 0;
    while(j < i){
      NumericVector uj = U(_ , j);
      NumericVector ai = A(_ , i);
      subt = subt + projC(uj, ai);
      j++;
    }
    Ucol = Acol - subt;
  }
  for(int i = 0; i < b; i++){
    NumericMatrix::Column ui = U(_ , i);
    NumericMatrix::Column qi = Q(_ , i);

    double sum2_ui = 0;
    for(int j = 0; j < a; j++){
      sum2_ui = sum2_ui + ui[j]*ui[j];
    }
    qi = ui/sqrt(sum2_ui);
  }

  List L = List::create(Named("Q") = Q , _["U"] = U);
  return L;
}
```

Comparing the R vs C++ implementation

In addition to the two functions above, I have a third function that is identical to the `R` one except that it calls `projC` instead of `proj`. I name this function `myQRC`. (`projC` is written in `C++` while `proj` is written in `R`). Otherwise, we have a purely `C++` function `myQRCpp` and a purely `R` function `myQR`.

To compare how quickly these three functions perform the ***QR*** factorization I put them in a function `QR_comp` that calls and times each with the same matrix argument.

```
QR_comp = function(A){
  t0 = Sys.time()
  myQR(A)
```

```
  tQRC = Sys.time() - t0
  t0 = Sys.time()
  myQRCpp(A)
  tQRCpp = Sys.time() - t0

  return(data.frame(tQR = as.numeric(tQR),
                    tQRC = as.numeric(tQRC),
                    tQRCpp = as.numeric(tQRCpp)))
}
```

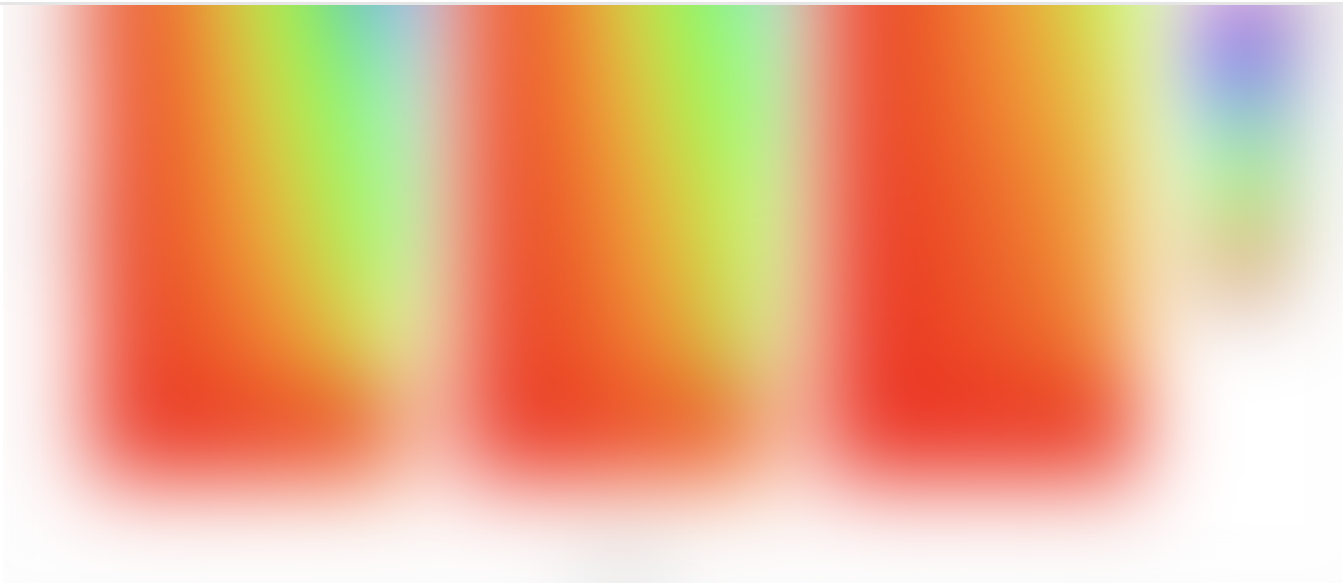
We can compare their performance over a grid of `n` by `m` random matrices. These matrices are generated when calling the `QR_comp` function.

```
grid = expand.grid(n = seq(10, 3010, 500),
                   m = seq(50, 600, 50))

tvec = map2(grid$n,
            grid$m,
            ~QR_comp(matrix(runif(.x*.y), ncol = .y)))
```

Finally, we can visually asses how these vary.

```
plotly::ggplotly(
bind_rows(tvec) %>%
  gather("func","time") %>%
  mutate(n = rep(grid$n, 3),
         m = rep(grid$m, 3)) %>%
  ggplot(aes(m, n, fill = time)) +
  geom_tile() +
  facet_grid(.~func) +
  scale_fill_gradientn(colours = rainbow(9)) +
  theme(panel.background = element_blank(),
        axis.ticks.y = element_blank(),
        axis.text.y = element_text(angle = 35, size = 5),
        axis.text.x = element_text(angle = 30, size = 5)), width =
550, heigh = 400)
```



Clearly the more `C++` involved the faster the ***QR*** factorization can be computed. The all `C++` function solves in under a minute for matrices with up to `250` columns and `3000` rows or `600` columns and `500` rows. The `R` function is 2-3 times slower.

Conclusion

QR is just one matrix factorization and LS is just one application of the QR. Hopefully, the above discussion demonstrates how important and useful Linea Algebra is for data science. In the future, I'll cover another application of the QR factorization and move onto some other important factorizations like the Eigenvalue and the SVD decompositions.

Additionally, you can tell that I am using `R` and `C++` to implement these methods computationally. I hope that this is useful and will inspire other `R` users like myself to learn `C++` and `Rcpp` and have that in their toolkit to make their `R` work even more powerful.

Thanks for reading!

Data Science Linear Algebra Linear Regression Mathematics R Programming