Cholesky Post

Will Edwards

1/30/2020

The Cholesky Decomposition is a useful method for solving linear systems of equations. The scenario I will discuss is solving a system of equations for the parameters in a linear regression; equations of the form $Ax = b$.

When we want to build a regression model, we will usually have a matrix X and a vector Y which constitutes our data. The vector Y is our response; this is the dependent variable. The matrix X is the design matrix; information about the independent variables is contained here.

Building a regression model involves finding the vector β that minimizes the errors. This problem is commonly written as:

$$argmin||Y - X\beta||^2$$

Basically, we are looking for "the argument that minimizes the distance between Y and its approximation."

Some derivation and algebra leads to a resulting system of equations that can be solved to find the vector β that minimizes this distance:

$$X^T X\beta = X^T Y$$

These equations are known to statisticians and data scientists as the normal equations. Notice that this is of the form of $AX = b$, where $A = X^T X$ and $b = X^T Y$.

Solving these equations for β usually involves finding the inverse of $X^T X$ and then multiplying both sides by the inverse and finding that β is:

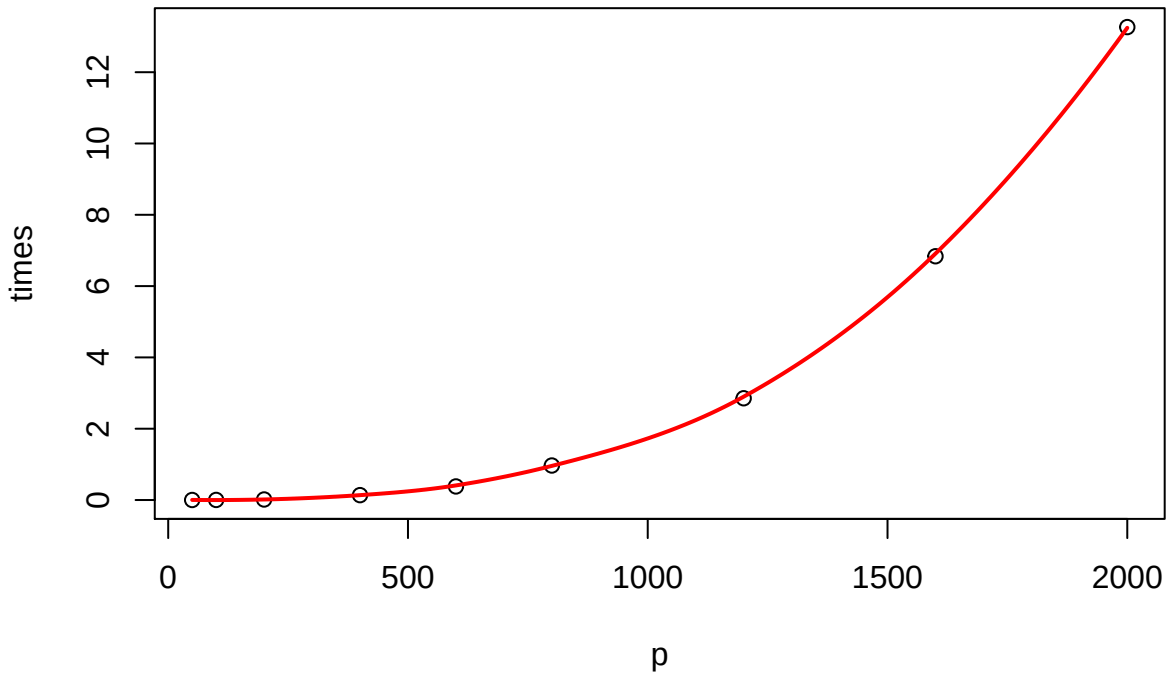
$$\beta = (X^T X)^{-1} X^T Y$$

Computational Complexity

Computational complexity is a way to quantify the resources needed for an algorithm to run. When building a regression model the most expensive operation is finding the inverse of $X^T X$. Naively computing the inverse of $X^T X$ is $O(n^3)$ complexity. This notation, known as "big-O notation", illustrates how run-time increases as n increases. In this situation, n refers to the dimension of the matrix $X^T X$. When dealing with high-dimensional data, data with a lot of parameters, this means that as the number of variables increases, run-time increses in a cubic manner. The following is an illustration of how runtime increases.

```
library(knitr)
set.seed(3452)

p <- c(50,100,200,400,600,800,1200,1600,2000)

times <- c() #hold times


for (i in 1:9) {
```

```
      X <- matrix(rnorm(2000*p[i]), ncol = p[i])
      A <- t(X)%*%X
      times[i] <- system.time(solve(A))[1] #naive brute force solve by finding inverse
}

lo <- loess(times~p)
plot(p,times)
xl <- seq(min(p),max(p), (max(p) - min(p))/1000)
lines(xl, predict(lo,xl), col='red', lwd=2)
```



P	Time (s)
50	0.001
100	0.002
200	0.015
400	0.137
600	0.381
800	0.968
1200	2.854
1600	6.837
2000	13.265

Note that $X^T X$ is a matrix with dimension PxP. We can see from the plot that as the dimensions of the matrix increase, the computation time can really start to increase. Notice in the table that doubling the number of paramters increases computation about 8 times; this is the cubic complexity of the naive inverse at work. Doubling from 400 to 800 increases the time about 6 or 7 times longer. Doubling again from 800 to 1600 increases the time even further, about 7 times larger.

It's clear that as our $X^T X$ matrix grows, the run-time can really start to ramp up fast! This is where the Cholesky Decomposition comes in.

Cholesky

If we have a symmetric and positive-definite matrix, call it A, the Cholesky Decomposition produces a unique factorization such that $A = R^T R$ where the matrix R is upper triangular, meaning that the upper half of the matrix is filled with 0s. Example:

```
X <- matrix(c(4,7,6,2,9,9,7,5,4), ncol = 3)
A <- t(X)%*%X
R <- chol(A)
R
```

```
##          [,1]      [,2]       [,3]
## [1,] 10.04988 12.437965  8.6568236
## [2,]  0.00000  3.361105 -3.7705652
## [3,]  0.00000  0.000000  0.9177383
```

Note that once we have R, we could get A back by multiplying $R^T R$:

```
t(R)%*%R
```

```
##      [,1] [,2] [,3]
## [1,]  101  125   87
## [2,]  125  166   95
## [3,]   87   95   90
```

The trick is that after factorizing A, the system of linear equations is much easier to solve. Remember that we started out with the normal equations, and we will let $X^T X = A$. Then:

$$A\beta = X^T Y$$

$$\Rightarrow Chol(A)\beta = X^T Y$$

$$R^T R\beta = X^T Y$$

Now we have a new system of equations. Now we let $R\beta$ be some unknown z, and we have:

$$R^T z = X^T Y$$

Now we have a system of equations that is much easier to solve; because R is upper triangular, R^T will be lower triangular and can be solved much faster. So first we solve this sytem for z. Then we simply recall that

$$R\beta = z$$

Realize that this system of equations is *also* triangular; it will be easy to just solve for β!

The `forwardsolve()` and the `backsolve()` functions available in base R will be useful to us because they solve lower and upper triangular systems of equations.

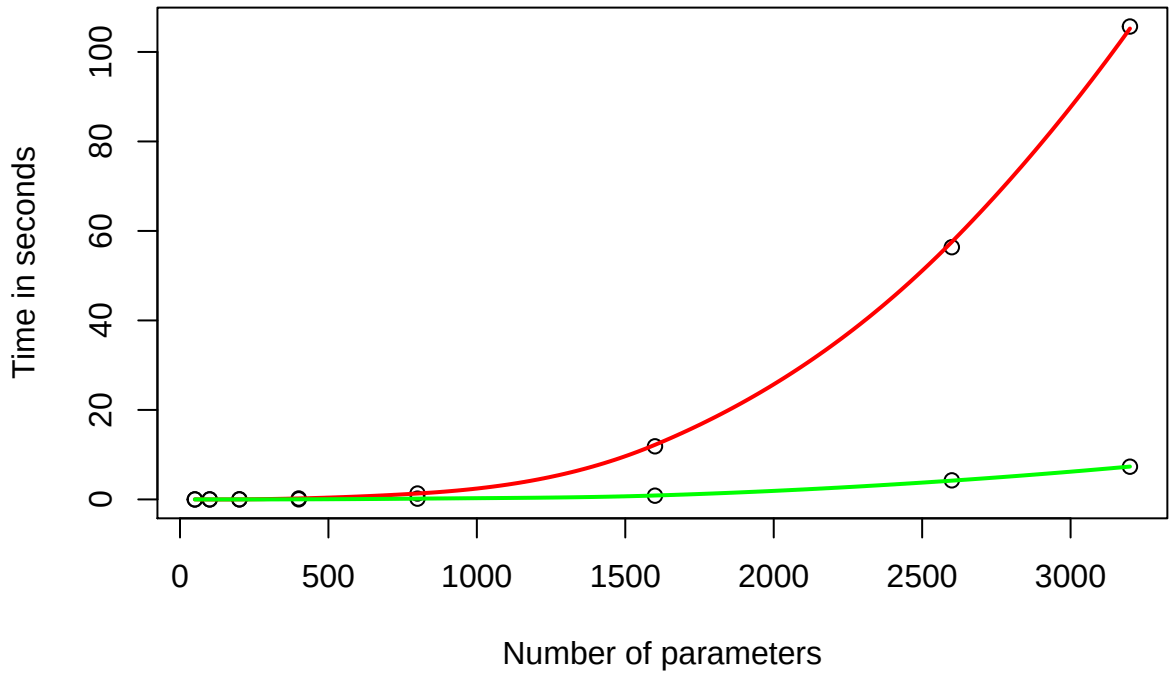
The good part

Forget all the linear algebra and let's get to the fun part: a demonstration of how much faster we can compute the β solution to the normal equations!

```
set.seed(1234)
n <- c(50,100,200,400,800,1600,3200,4000)
p <- c(50,100,200,400,800,1600,2600,3200)
library(MASS)
library(stats)
```

```
times <- c() #hold times for naive
beta <- matrix( 0, ncol = 3,nrow=8) #matrix to hold values in case need to check betas match
times_faster <-c() #hold times for faster
beta1 <- matrix( 0, ncol = 3,nrow=8)

for (i in 1:8) {

      Y <- rnorm(n[i]) #create matrix of random response data
      X <- matrix(rnorm(n[i]*p[i]), ncol = p[i]) #random matrix of data

      A <- t(X)%*%X #create X^TX matrix and call it A

      beta[i,] <- (solve(t(X)%*%X,tol=.0000000001)%*%t(X)%*%Y)[1:3] #grabs 4th beta value to compare wi

      times[i] <- system.time(solve(A)%*%t(X)%*%Y)[1] #get time for naive brute force solve by finding

      time_decompose <- system.time(R <- chol(A) )[1]#time to decompose

    t1 <- system.time(z <- forwardsolve(t(R), t(X)%*%Y))[1] # time solve for R^T*beta
    t2 <- system.time(beta1[i,] <- backsolve(R,z)[1:3])[1] # solves for beta, also grabs the 4th beta v
    times_faster[i] <- time_decompose+t1+t2 # add the times from several operations
}
```

```
plot(p, times, xlab = "Number of parameters", ylab = "Time in seconds")
points(p,times_faster)
lo <- loess(times~p)
lo2 <- loess(times_faster~p)
xl <- seq(min(p),max(p), (max(p) - min(p))/1000)
lines(xl, predict(lo,xl), col='red', lwd=2)
lines(xl, predict(lo2,xl), col='green',lwd=2)
```



Also included is the table of values:

P	Time Naive (s)	Cholesky Time (s)
50	0.001	0.000
100	0.002	0.000
200	0.024	0.003
400	0.207	0.015
800	1.318	0.174
1600	11.865	0.832
2600	56.367	4.263
3200	105.685	7.312

Additionally, to ensure that the Cholesky method finds the same solution, during each iteration I grabbed the first 3 beta values computed by each method. Then I compared to see if all the values computed with the naive method matched those created by the Cholesky method.

```
all.equal(beta,beta1)
```

```
## [1] TRUE
```

Although a situation where this is relevant may be rare for most people, being aware of some linear algebra tricks can be invaluable for a data scientist or statistician dealing with large data.