Least Squares Solutions for Overdetermined Systems

Joel S Steele

Overdetermined systems

When we want to solve systems of linear equations, $\hat{y} = X\beta$, we need as many equations as unknowns. We also hope that each equation give unique information, or that it is independent of the other equations. When the number of unknowns is equal to the number of equations there may exist a single unique solution. In the case where there are more unknowns than equations we say that the system is *underdetermined* since we have no way to uniquely solve for every unknown.

A more common case is when there are more equations than unknowns, in this case the system is *overdetermined* and we have many possible solutions, but no single unique one. The problem then becomes, how to identify satisfactory solutions. This leads to the development of the concept of *error* and *cost*. We typically define error as the difference between **expected** ($\hat{y}$) and **observed** (y) values,

$$\epsilon = X\beta - y.$$

In the majority of situations we define a cost function as the sum of all of squared errors that a given candidate solution produces. We then hope to minimize this cost function.

$$min_{\beta}||X\beta - y||$$

Example data

Below are some example data from the **High-School & Beyond** dataset that we will be using for these examples.

```r
mls = data.frame(
  read = c(63, 55, 60, 73, 37, 68, 76, 66, 63, 60,
           52, 50, 36, 57, 50, 42, 73, 47, 68, 50),
  write = c(57, 39, 62, 67, 44, 60, 63, 67, 57, 46,
            49, 41, 57, 52, 49, 49, 62, 62, 65, 52),
  science = c(58, 53, 50, 58, 39, 69, 67, 61, 58, 53,
              44, 44, 50, 61, 47, 50, 69, 53, 55, 39),
  math = c(55, 57, 67, 62, 45, 64, 60, 67, 54, 51, 49,
           45, 42, 40, 56, 43, 73, 53, 62, 53),
  socst = c(41, 46, 56, 66, 46, 66, 66, 66, 51, 61,
            61, 56, 41, 56, 46, 56, 66, 61, 61, 56))
```

The Linear Model

For these data we will be using the following linear model

$$Sci_i = \beta_0 + \beta_1 Read_i + \beta_2 Math_i + \epsilon_i$$

Data preparation and linear modeling in R

To begin, we prepare our data by selecting out our response *science* into the vector y, and combining our predictors *read* and *math* with a column of 1*s* as our design matrix.

```r
# prepare the data
y = mls$science
x = cbind('(Intercept)'=1,mls[,c('read','math')])
x = as.matrix(x)
```

For comparison we will estimate the model using R's built in function *lm()*.

```r
# from R
coef(lm(science~read+math,mls))->lma
lma
```

```
(Intercept)        read        math 
21.88956117  0.64655253 -0.09174902 
```

Direct Matrix Inversion

The analytic solution to this system using matrix algebra is expressed as

$$[X^TX]^{-1}X^Ty = \beta$$

which can written in R as the following.

```r
# the solve() function performs the matrix inversion
solve( t(x) %*% x ) %*% t(x) %*% y -> ols
ols
```

```
                   [,1]
(Intercept) 21.88956117
read         0.64655253
math        -0.09174902
```

This produces the same estimates which is nice to see. However, inverting a matrix can be difficult, time consuming, and overall computationally intensive. Below are other methods that break down the solution into more manageable parts. Specifically, instead of inverting the full matrix X^TX, it can be broken down or **factorized** in some way to make the inversion process much less computationally intensive.

QR method

The QR decomposition, or factorization, takes a matrix A and produces two additional matrices Q and R that represent an *orthogonal* matrix and a *triangular* matrix respectively. The form of the decomposition is

$$A = QR$$

Where

$$Q^TQ = I$$

because it's *orthogonal* and R is *upper triangular*. The real benefit to this decomposition is that inverting a triangular matrix is **much** easier than inverting a full matrix.

```r
# QR decomposition to solve
qr.Q(qr(x)) -> Q
qr.R(qr(x)) -> R
# Make sure Q is orthogonal
kable(as.data.frame( t(Q) %*% Q ))
```

V1	V2	V3
1	0	0
0	1	0
0	0	1

```r
# What is in R
R
```

```
     (Intercept)       read       math
[1,]   -4.472136 -256.25339 -245.52026
[2,]    0.000000   51.24646   30.70651
[3,]    0.000000    0.00000  -26.01750
```

Using the QR decomposition, the solution to our least squares problem from before becomes

$$R^{-1}Q^Ty = \beta$$

```r
# Using QR (Spoiler Alert: this is what the lm() function does on the backend)
solve(R) %*% t(Q) %*% y ->qrs
qrs
```

```
                   [,1]
(Intercept) 21.88956117
read         0.64655253
math        -0.09174902
```

Again, the same results.

Cholesky method

Cholesky (*Coal-Ess-Key*) decomposition, takes a similar approach, but here the matrix A is factored into two *triangular* matrices. In R the factorization produces **upper** triangular matrices U, such that $U^TU = A$. For our purposes we will therefore create a sum-of-squares and cross-products matrix $A = X^TX$ and use the Cholesky decomposition on A. Since we want the inverse of A this becomes $A^{-1} = U^{-1}U^{-1T}$

Now, we can then solve our least squares problem as

$$U^{-1}U^{-1T}X^Ty = \beta$$

```r
# Cholesky
cv = t(x) %*%x
u = chol(cv)
uinv = solve(u)
uinv %*% t(uinv) %*% t(x) %*% y ->chs
chs
```

```
                    [,1]
(Intercept) 21.88956117
read         0.64655253
math        -0.09174902
```

Same results as above.

Singular Value Decomposition

The final matrix based method that I will present is the Singular Value Decomposition or SVD for short. This decomposition breaks down our matrix A into three new matrices U, D, and V, such that $A = UDV^T$. In this case both U and V are *orthogonal* and D is a diagonal matrix containing the singular values such that $D = U^T AV$.

Inverting diagonal matrices

The fact that D is diagonal makes this a much easier problem since the inverse of a diagonal matrix is equal to a matrix with each diagonal element inverted (only for non-zero elements of course!)

As a quick example

```r
A = diag(c(2,4,5))
A
```

```
     [,1] [,2] [,3]
[1,]    2    0    0
[2,]    0    4    0
[3,]    0    0    5
```

```r
solve(A)
```

```
     [,1] [,2] [,3]
[1,]  0.5 0.00  0.0
[2,]  0.0 0.25  0.0
[3,]  0.0 0.00  0.2
```

Now, most of the computational time for this method is not spent with matrix inversion, but rather with the SVD itself, which can be very time consuming.

Nevertheless, using this decomposition the solution to our least squares problem becomes

$$VD^{-1}U^T y = \beta$$

```r
# Singular Value Decomposition
dcomp = svd(x)
V = dcomp$v       # orthogonal eigenvectors
D = diag(dcomp$d) # diagonal singular values
U = dcomp$u       # orthogonal
V %*% solve(D) %*% t(U) %*% y ->svs
svs
```

```
            [,1]
[1,] 21.88956117
[2,]  0.64655253
[3,] -0.09174902
```

Again, the same results.

How do they compare?

```r
results = data.frame('R'=lma, 'OLS'=ols, 'QR'=qrs, 'Cholesky'=chs, 'SVD'=svs)
kable(results, digits=5)
```

	R	OLS	QR	Cholesky	SVD
(Intercept)	21.88956	21.88956	21.88956	21.88956	21.88956
read	0.64655	0.64655	0.64655	0.64655	0.64655
math	-0.09175	-0.09175	-0.09175	-0.09175	-0.09175

As expected, the estimates are exact. Now, keep in mind that this is not a new method for solving the system of equations, but rather for dealing with the messy business of matrix inversion. We are still using the solution

$$[X^TX]^{-1}X^Ty = \beta$$

but we are using **factorization** tricks on the $[X^TX]^{-1}X^T$ part.

Interaction effect estimates?

Here we include an interaction effect between *math* and *read*, thus our model becomes,

$$Sci_i = \beta_0 + \beta_1 Read_i + \beta_2 Math_i + \beta_3(Read_i \times Math_i) + \epsilon_i$$

```r
# interactions?
x = cbind(x,'intrxn'=mls$read*mls$math)
x = as.matrix(x)
# from R
lm0 = lm(science~read*math,mls)
lm0$coef ->beta1
# Traditional Least squares
solve( t(x) %*% x ) %*% t(x) %*% y -> beta2
# QR decomposition to solve
qr.Q(qr(x)) -> Q
qr.R(qr(x)) -> R
solve(R) %*% t(Q) %*% y -> beta3
# Cholesky
cv = t(x) %*%x
u = chol(cv)
uinv = solve(u)
uinv %*% t(uinv) %*% t(x) %*% y ->beta4
# Singular Value Decomposition
```

```
dcomp = svd(x)
V = dcomp$v       # orthogonal eigenvectors
D = diag(dcomp$d) # diagonal singular values
U = dcomp$u       # orthogonal
V %*% solve(D) %*% t(U) %*% y ->beta5
```

How do they compare?

```
res = data.frame('R'=beta1, 'OLS'=beta2, 'QR'=beta3, 'Cholesky'=beta4, 'SVD'=beta5)
kable(res, digits=5)
```

	R	OLS	QR	Cholesky	SVD
(Intercept)	107.29380	107.29380	107.29380	107.29380	107.29380
read	-0.78323	-0.78323	-0.78323	-0.78323	-0.78323
math	-1.78291	-1.78291	-1.78291	-1.78291	-1.78291
read:math	0.02772	0.02772	0.02772	0.02772	0.02772

Perfect, as expected!

Standard errors of estimates

Just as a quick aside, we can also use aspects of these matrix methods for model assessment as well. In particular we may be interested in computing *standard errors* of the parameters as well as of the estimates.

Looking for standard errors of estimates for QR decomposition it will be helpful to know that errors are defined as

$$\epsilon = (I - QQ^T)y$$

Also, remember that

$$MSE = \frac{1}{n-p}\epsilon^T\epsilon$$

and

$$SE_\beta = \sqrt{\frac{MSE}{SS_x}}$$

which in the case of QR factorization becomes,

$$SE_\beta = \sqrt{\frac{MSE}{R^TR}}$$

```
n = nrow(x)
p = ncol(x)
# e = (I - QQ')y
( diag(20) - Q %*% t(Q) ) %*% y -> err3
# MSE = 1/n-p (e'e)
MSE = 1/(n-p) * t(err3) %*% err3
# SEb = (MSE/SSx)^.5
SEb = sqrt( MSE * diag( solve( t(R) %*% R ) ) )
kable(data.frame('Est' = beta3,'Std.Err'=SEb, 't val'=beta3/SEb), digits=c(5,5,3))
```

	Est	Std.Err	t.val
(Intercept)	107.29380	44.13585	2.431
read	-0.78323	0.74634	-1.049
math	-1.78291	0.88619	-2.012
intrxn	0.02772	0.01411	1.964

with a residual standard error of,

```
sqrt(MSE)
```

```
         [,1]
[1,] 5.455545
```

And by comparison

```
summary(lm0)
```

```

Call:
lm(formula = science ~ read * math, data = mls)

Residuals:
   Min     1Q Median     3Q    Max 
-8.084 -5.258  1.223  2.603  8.454 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)  
(Intercept) 107.29380   44.13585   2.431   0.0272 *
read         -0.78323    0.74634  -1.049   0.3096  
math         -1.78291    0.88619  -2.012   0.0614 .
read:math     0.02772    0.01411   1.964   0.0671 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 5.456 on 16 degrees of freedom
Multiple R-squared:  0.6858,	Adjusted R-squared:  0.6269 
F-statistic: 11.64 on 3 and 16 DF,  p-value: 0.0002689
```