Lecture 13
Solving Least Squares

28 October 2015

Taylor B. Arnold
Yale Statistics
STAT 312/612

Yale

Notes

- Problem Set #4 - Due today

Goals for today

- How to solve the normal equations in a stable way
- An alternative QR method
- Simulation to a highly correlated dataset
- Relationship of statistical noise to numerical error

Solving least squares

When started looking at multivariate regression, I wrote down the normal equations:

$$(X^t X)\widehat{\beta} = X^t y$$

Recall that these are called equations (plural) because we can think of this as a set of p simultaneous equations.

For the last month we have been assuming that we solve this by just taking the matrix inverse of $X^t X$ to yield the following:

$$\widehat{\beta} = (X^t X)^{-1} X^t y$$

For the last month we have been assuming that we solve this by just taking the matrix inverse of $X^t X$ to yield the following:

$$\widehat{\beta} = (X^t X)^{-1} X^t y$$

Why might this be a problem? Well, consider the simple case where we have $n = p = 2$ with the following:

$$X = \begin{pmatrix} 10^9 & -1 \\ -1 & 10^{-5} \end{pmatrix}$$

$$\beta = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

For simplicity, we'll even assume that there is no noise vector. Then we have:

$$\begin{aligned} y &= \begin{pmatrix} 10^9 & -1 \\ -1 & 10^{-5} \end{pmatrix} * \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 10^9 - 1 \\ -0.99999 \end{pmatrix} \end{aligned}$$

Let's try running this in R:

```
> X  <- matrix(c(10^9, -1, -1, 10^(-5)), 2, 2)
> X
       [,1]   [,2]
[1,]  1e+09 -1e+00
[2,] -1e+00  1e-05
> beta <- c(1,1)
> y <- X %*% beta
> y
            [,1]
[1,]  1.0000e+09
[2,] -9.9999e-01
```

So far, so good. Now, because we have a square design matrix we can invert it directly and solve $\beta = X^{-1}y$. This is not a problem here:

```
> Xinv <- solve(X)
> Xinv %*% y
     [,1]
[1,]    1
[2,]    1
```

The correct β is returned.

However, what if we try to calculate this with the normal equations? Here we need to invert the matrix $X^t X$.

```
> XtXinv <- solve(t(X) %*% X)
Error in solve.default(t(X) %*% X) :
  system is computationally singular: reciprocal
  condition number = 8.09999e-23
```

R knows that this is not going to be good, and refuses to calculate the inverse by default.

Suppose that we turn off this warning (by setting the tolerance to zero); what happens?

```
> XtX <- t(X) %*% X
> Xty <- t(X) %*% y
> XtXinv <- solve(XtX, tol=0)
> XtXinv
           [,1]       [,2]
[1,] 1.0002e-08 1.0002e+01
[2,] 1.0002e+01 1.0002e+10
> betaHat <- XtXinv %*% Xty
> betaHat
             [,1]
[1,]    0.9999995
[2,] 1080.4998042
```

So, here we see the results of the numerical instability. The returned $\widehat{\beta}$ is not equal to the correct β!

So, other than inverting the matrix what are our options?

We'll start by looking at three approachs, each of which involves a matrix decomposition.

Consider the qr-decomposition of the matrix X^tX:

$$X^tX = QR$$

Where Q is a square orthonormal matrix ($QQ^t = \mathbb{I}_p$) and R is an upper triangular matrix.

Because Q is orthonormal, we can calculate its inverse in an easy and stable way, yielding the following:

$$\begin{aligned}
X^t X\widehat{\beta} &= X^t y \\
QR\widehat{\beta} &= X^t y \\
R\widehat{\beta} &= Q^t X^t y \\
R\widehat{\beta} &= v
\end{aligned}$$

Where v is just a compact way of writing $Q^t X^t y$.

To solve the equation:

$$Rb = v$$

We use a technique called *backsolving*.

The best way to describe the backsolve algorithm is with a simple example. Consider the 3 by 3 matrix case:

$$\begin{pmatrix} R_{1,1} & R_{1,2} & R_{1,3} \\ 0 & R_{2,2} & R_{2,3} \\ 0 & 0 & R_{3,3} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} v1 \\ v2 \\ v3 \end{pmatrix}$$

What is an obvious place to start?

Well, we can see right away that:

$$b_3 = \frac{v_3}{R_{3,3}}$$

Now, if we know b_3 what can we do as a next step?

$$\begin{pmatrix} R_{1,1} & R_{1,2} & R_{1,3} \\ 0 & R_{2,2} & R_{2,3} \\ 0 & 0 & R_{3,3} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} v1 \\ v2 \\ v3 \end{pmatrix}$$

Well, now we can calculate

$$b_2 = \frac{v_2}{R_{2,2}} + \frac{R_{2,3} \cdot b_3}{R_{2,2}}$$

So, now for the final line of

$$\begin{pmatrix} R_{1,1} & R_{1,2} & R_{1,3} \\ 0 & R_{2,2} & R_{2,3} \\ 0 & 0 & R_{3,3} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} v1 \\ v2 \\ v3 \end{pmatrix}$$

By calculating the same thing:

$$b_1 = \frac{v_1}{R_{1,1}} + \frac{R_{1,2} \cdot b_2}{R_{1,1}} + \frac{R_{1,3} \cdot b_3}{R_{1,1}}$$

You can see that this algorithm generalizes to solving any linear system $Ab = x$ with an upper diagonal matrix A. In only requires basic arithmetic and therefore has minimal numerical issues.

Notice that we never actually produce the inverse matrix itself. We only apply and algorithm which produces the inverse at any given point.

Now, back to our example:

$$\begin{aligned} X^t X \widehat{\beta} &= X^t y \\ QR\widehat{\beta} &= X^t y \\ R\widehat{\beta} &= Q^t X^t y \\ R\widehat{\beta} &= v \end{aligned}$$

We can then complete the final line by backsolving.

We can calculate the qr decomposition in R:

```
> QR <- qr(XtX)
> QR
$qr
        [,1]         [,2]
[1,] -1e+18 1.000000e+09
[2,] -1e-09 9.998002e-11

$rank
[1] 1

$qraux
[1] 2.000000e+00 9.998002e-11

$pivot
[1] 1 2
```

To get the actual matricies, we need to run two functions to calcualte them from the R object:

```
> Q <- qr.Q(QR)
> R <- qr.R(QR)
```

And then we can solve the normal equations:

```
> v <- t(Q) %*% Xty
> backsolve(R, v)
     [,1]
[1,]    1
[2,]    0
```

Okay, so still not the correct β (which should have both components equal to 1), but still a lot better than the completely unstable solution we had previously.

Note that we can do this all in one step automatically inside of R. However, by default we get a warning this this is a bad idea:

```
> betaHat <- qr.solve(XtX, Xty)
Error in qr.solve(XtX, Xty) : singular matrix 'a' in solve
```

Turning this off yields the result that we calculated ourselves:

```
> betaHat <- qr.solve(XtX, Xty, tol=0)
> betaHat
     [,1]
[1,]    1
[2,]    0
```

Another decompostion that is useful for solving this problem is the Cholseky decomposition, which writes a matrix as $U^t U$ for some upper diagonal matrix U. If we use this the solution can be calculated by forward solving (the same as backsolving, but for a lower diagonal matrix) and then backsolving.

In R, we can do this as follows:

```
> U <- chol(XtX)
> U
      [,1]          [,2]
[1,] 1e+09 -1.000000e+00
[2,] 0e+00  9.999001e-06
```

And solving yields:

```
> betaHat <- backsolve(U, forwardsolve(t(U), Xty))
> betaHat
     [,1]
[1,]    1
[2,]    0
```

So this gives the same solution as the QR decomposition.

A final technique is the eigenvalue decomposition, which writes a matrix as QDQ^t for Q orthogonal and D a diagonal matrix.

To solve this we simply multiply by Q^t on the left, divide by the diagonal of D, and multiply by Q.

In R, the eigen value decomposition is given by the function `eigen`:

```
> EIGEN <- eigen(XtX)
> EIGEN
$values
[1] 1.000000e+18 9.998002e-11

$vectors
       [,1]   [,2]
[1,] -1e+00 -1e-09
[2,]  1e-09 -1e+00
```

And to solve the linear system we see that:

```
> Q <- EIGEN$vectors
> D <- diag(EIGEN$values)
> step01 <- t(Q) %*% Xty
> step02 <- step01 / diag(D)
> betaHat <- Q %*% step02
> betaHat
        [,1]
[1,]  1e+00
[2,] -1e-09
```

Which is very close to what the other methods yielded.

So we have three methods: the Cholesky decomposition which involves two diagonal solves (one back and one forward), the QR which involves one diagonal solve and one matrix multiplication, and the eigenvalue decompositon which involves two matrix multiplications.

Each of these has its own uses and pros and cons. We will likely discuss some of these over the next few weeks. For now though, we are mostly concerned with the fact that none of them return the correct $\widehat{\beta}$ that we expect.

So far we have only taken decompositions of square matricies. The QR decomposition actually yields a solution even for rectangular matricies. We can write the decomposition of the n-by-p matrix A as:

$$A = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \cdot \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$$

With R_1 being a p-by-p matrix, Q_1 a n-by-p matrix and Q_2 a n-by-$(n-p)$ matrix. This is sometimes known as the *thin* QR decomposition. Notice that Q_2 is not unique, nor is it even needed to reconstruct A.

Now, let's take a step back from the normal equations and consider the sum of squares directly:

$$||y - X\beta||_2^2 = (y - X\beta)^t(y - X\beta)$$

Notice that if we have an orthonormal matrix Q we can insert this into the middle of the sum of squares without changing the value:

$$\begin{aligned}
||Q(y - X\beta)||_2^2 &= (y - X\beta)^t Q^t Q(y - X\beta) \\
&= (y - X\beta)^t Q^t Q(y - X\beta) \\
&= (y - X\beta)^t(y - X\beta) \\
&= ||y - X\beta||_2^2
\end{aligned}$$

On its own, this is an important (though hopefully unsurprising) result. We can rotate the problem by any orthonormal matrix and return the same sum of squares; this will be very useful in several ways.

On of those ways comes from taking the QR decomposition of the matrix X and rotating the space by the transpose of the resulting Q.

Then, we can then write the squared residuals as

$$\begin{aligned} Q^t(y - X\beta) &= \begin{bmatrix} Q_1^t \\ Q_2^t \end{bmatrix} y - \begin{bmatrix} Q_1^t \\ Q_2^t \end{bmatrix} \times \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \cdot \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \beta \\ &= \begin{bmatrix} Q_1^t \\ Q_2^t \end{bmatrix} y - \begin{bmatrix} R_1 \\ 0 \end{bmatrix} \beta \end{aligned}$$

Notice that the final $n - p$ rows of the resulting vector do not involve β as they are canceled out by the 0.

So we now have the following equation:

$$\begin{aligned} ||y - X\beta||_2^2 &= ||Q^t(y - X\beta)||_2^2 \\ &= ||Q_1^t y - R_1\beta||_2^2 + ||Q_2^t y||_2^2 \end{aligned}$$

This is actually very useful, because we can minimize over β by just minimizing the first term, which can actually be set exactly equal to zero:

$$\begin{aligned} Q_1^t y &= R_1\beta \\ R_1^{-1} Q_1^t y &= \beta \end{aligned}$$

Though, as you now know, we would actually backsolve rather than take the actual inverse of R_1.

So now, let's try this on our problem:

```
> E <- qr(X)
> Q <- qr.Q(E)
> R <- qr.R(E)
> backsolve(R, t(Q) %*% y)
     [,1]
[1,]    1
[2,]    1
```

Thankfully, this yields the expected result!

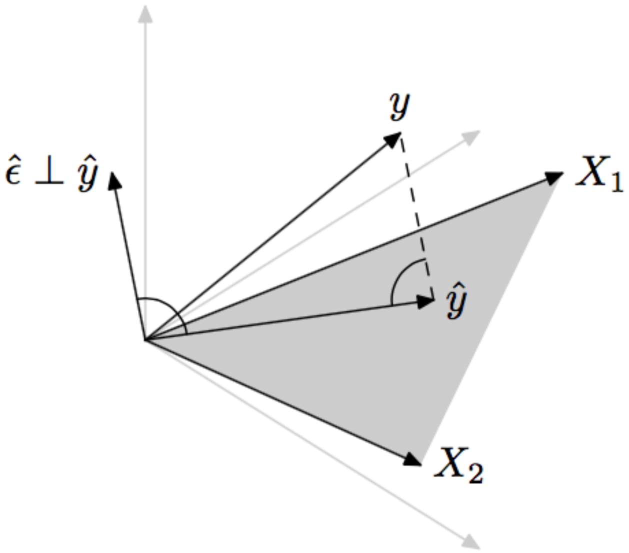
Carefully note though, that there is some numerical error that is hidden by the default precision of R:

```
> options(digits=22)
> backsolve(R, t(Q) %*% y)
                            [,1]
[1,] 1.000000000000000000000
[2,] 0.999999999999827693386
```

Carefully note though, that there is some numerical error that is hidden by the default precision of R:

```
> options(digits=22)
> backsolve(R, t(Q) %*% y)
                          [,1]
[1,] 1.000000000000000000000
[2,] 0.999999999999827693386 58
```

This method has a nice geometric interpretation in relationship to the geometry of least squares:



What is really going on here?

```
> options(digits=22)
> betaHat <- qr.solve(XtX, Xty, tol=0)
                            [,1]
[1,] 0.9999999990000000282819
[2,] 0.0000000000000000000000
> XtX %*% betaHat - Xty
     [,1]
[1,]    0
[2,]    0
```

Directly solving the normal equations produces solutions that correctly give $X^tX\beta = X^ty$.

Now, let us try this on a non-square matrix to illustrate what happens in a more typically setting.

We construct a dataset with 25 highly correlated columns:

```
> options(digits=22)
> n <- 1000
> p <- 25
> alpha <- 1e-5
> X <- matrix(runif(n*p), ncol=p)
> X <- alpha * X + matrix(rnorm(n), nrow=n, ncol=p)
>
> beta <- runif(p)
> y <- X %*% beta + rnorm(n,sd=0.5)
> cor(X[,1], X[,2])
[1] 0.9999999999916668880218
```

Now let us try our two methods for solving the ordinary least squares problem.

Normal equations (Cholesky)

```
> XtX <- t(X) %*% X
> Xty <- t(X) %*% y
>
> U <- chol(XtX)
> betaChol <- backsolve(U, forwardsolve(t(U),Xty))
```

QR-decomposition

```
> QR <- qr(X)
> Q <- qr.Q(QR)
> R <- qr.R(QR)
> dim(Q)
[1] 1000   25
> dim(R)
[1] 25 25
>
> betaQR <- backsolve(R, t(Q) %*% y)
```

Look at the quantiles of the results

```
> options(digits=8)
> quantile(beta)
        0%         25%          50%          75%         100%
0.0042836969 0.2510972756 0.5518817164 0.8211262035 0.982795
> quantile(betaQR)
        0%         25%          50%          75%         100%
-15406.39499 -1497.12478   134.70958  3361.91288  7500.6566
> quantile(betaChol)
        0%         25%          50%          75%         100%
-15405.61168 -1499.20582   134.37818  3361.92816  7499.4664
```

The predicted values are significantly larger in magnitude than the true beta values!

Comparison of regression vectors:

```
> sqrt(sum(abs(betaQR - betaChol)^2)) / sqrt(sum(abs(betaQR)
[1] 0.00047420293
```

And the predicted values:

```
> sqrt(sum(abs(X %*% betaQR - X%*% betaChol)^2)) /
+  sqrt(sum(abs(X %*% betaQR)^2))
[1] 2.7345493e-06
```

These represent numerical imprecision; the difference between the two methods of calculating the regression vector

Now, compare the predicted value (QR) to the actual β and y values:

```
> sqrt(sum(abs(betaQR - beta)^2)) /
+   sqrt(sum(abs(betaQR)^2))
[1] 0.99999031
```

And the predicted values:

```
> sqrt(sum(abs(X %*% betaQR - X %*% beta)^2)) /
+  sqrt(sum(abs(X %*% betaQR)^2))
[1] 0.0055357352
```

These represent the statistical error. They are much worse, particularly for the regression vector.

Notice that the normal equations are solved very well, even though the regression vector is not:

```
> max(abs(t(X) %*% X %*% betaQR - t(X) %*% y))
[1] 5.7070793e-08
> max(abs(t(X) %*% X %*% betaChol - t(X) %*% y))
[1] 9.7188604e-09
```

What is going on here? We have seen this before when we had a model that was generated by the following:

$$Y = X_1 + X_2 + \text{noise}$$

If X_1 and X_2 are highly correlated it will be very difficult to distinguish the true model from any of the following:

$$\begin{aligned} Y &= 2 \times X_1 + \text{noise} \\ Y &= 3 \times X_1 - X_2 + \text{noise} \\ Y &= 200 \times X_1 - 199 \times X_2 + \text{noise} \end{aligned}$$

Notice how the coefficents can easily be orders of magnitude larger than the true model.

The big picture

Why do we (i.e., STAT 612) care about numerical precision?

It is very difficult to construct non-trival datasets that exhibit numerically unstable results (at least, without doing something that makes no sense in practice). The main reason we are still interested is because methods for addressing the statistical error mimic (and are motivated by) the methods for fixing numerical errors.

Bad statistical noise is **very** common even with only moderately correlated covariates!