Comparing Least Squares Calculations

Douglas Bates
R Development Core Team
Douglas.Bates@R-project.org

November 26, 2019

Abstract

Many statistics methods require one or more least squares problems to be solved. There are several ways to perform this calculation, using objects from the base R system and using objects in the classes defined in the `Matrix` package.

We compare the speed of some of these methods on a very small example and on a example for which the model matrix is large and sparse.

1 Linear least squares calculations

Many statistical techniques require least squares solutions

$$\hat{\beta} = \arg\min_{\beta} \|y - X\beta\|^2 \tag{1}$$

where X is an $n \times p$ model matrix ($p \leq n$), y is n-dimensional and β is p dimensional. Most statistics texts state that the solution to (1) is

$$\hat{\beta} = \left(X^\top X\right)^{-1} X^\top y \tag{2}$$

when X has full column rank (i.e. the columns of X are linearly independent) and all too frequently it is calculated in exactly this way.

1.1 A small example

As an example, let's create a model matrix, `mm`, and corresponding response vector, `y`, for a simple linear regression model using the `Formaldehyde` data.

```
> data(Formaldehyde)
> str(Formaldehyde)
```

```
'data.frame':        6 obs. of  2 variables:
 $ carb  : num  0.1 0.3 0.5 0.6 0.7 0.9
 $ optden: num  0.086 0.269 0.446 0.538 0.626 0.782
```

1.2 A large example

For a large, ill-conditioned least squares problem, such as that described in Koenker and Ng (2003), the literal translation of (2) does not perform well.

```
> library(Matrix)
> data(KNex, package = "Matrix")
>  y <- KNex$y
> mm <- as(KNex$mm, "matrix") # full traditional matrix
> dim(mm)

[1] 1850  712

> system.time(naive.sol <- solve(t(mm) %*% mm) %*% t(mm) %*% y)

   user  system elapsed
  1.413   0.010   1.426
```

Because the calculation of a "cross-product" matrix, such as $\mathbf{X}^\top\mathbf{X}$ or $\mathbf{X}^\top\mathbf{y}$, is a common operation in statistics, the `crossprod` function has been provided to do this efficiently. In the single argument form `crossprod(mm)` calculates $\mathbf{X}^\top\mathbf{X}$, taking advantage of the symmetry of the product. That is, instead of calculating the $712^2 = 506944$ elements of $\mathbf{X}^\top\mathbf{X}$ separately, it only calculates the $(712 \cdot 713)/2 = 253828$ elements in the upper triangle and replicates them in the lower triangle. Furthermore, there is no need to calculate the inverse of a matrix explicitly when solving a linear system of equations. When the two argument form of the `solve` function is used the linear system

$$\left(\mathbf{X}^\top\mathbf{X}\right)\boldsymbol{\beta} = \mathbf{X}^\top\mathbf{y} \tag{3}$$

is solved directly.

Combining these optimizations we obtain

```
> system.time(cpod.sol <- solve(crossprod(mm), crossprod(mm,y)))

   user  system elapsed
  0.604   0.002   0.608

> all.equal(naive.sol, cpod.sol)

[1] TRUE
```

On this computer (2.0 GHz Pentium-4, 1 GB Memory, Goto's BLAS, in Spring 2004) the crossprod form of the calculation is about four times as fast as the naive calculation. In fact, the entire crossprod solution is faster than simply calculating $\mathbf{X}^\top\mathbf{X}$ the naive way.

```
> system.time(t(mm) %*% mm)

   user  system elapsed
  0.581   0.003   0.585
```

Note that in newer versions of R and the BLAS library (as of summer 2007), R's %*% is able to detect the many zeros in mm and shortcut many operations, and is hence much faster for such a sparse matrix than crossprod which currently does not make use of such optimizations. This is not the case when R is linked against an optimized BLAS library such as GOTO or ATLAS. Also, for fully dense matrices, crossprod() indeed remains faster (by a factor of two, typically) independently of the BLAS library:

```
> fm <- mm
> set.seed(11)
> fm[] <- rnorm(length(fm))
> system.time(c1 <- t(fm) %*% fm)

   user  system elapsed
  0.593   0.000   0.595

> system.time(c2 <- crossprod(fm))

   user  system elapsed
  0.525   0.000   0.526

> stopifnot(all.equal(c1, c2, tol = 1e-12))
```

1.3 Least squares calculations with Matrix classes

The crossprod function applied to a single matrix takes advantage of symmetry when calculating the product but does not retain the information that the product is symmetric (and positive semidefinite). As a result the solution of (3) is performed using general linear system solver based on an LU decomposition when it would be faster, and more stable numerically, to use a Cholesky decomposition. The Cholesky decomposition could be used but it is rather awkward

```
> system.time(ch <- chol(crossprod(mm)))

   user  system elapsed
  0.585   0.000   0.587

> system.time(chol.sol <-
+             backsolve(ch, forwardsolve(ch, crossprod(mm, y),
+                                        upper = TRUE, trans = TRUE)))

   user  system elapsed
  0.003   0.000   0.003

> stopifnot(all.equal(chol.sol, naive.sol))
```

The Matrix package uses the S4 class system (Chambers, 1998) to retain information on the structure of matrices from the intermediate calculations. A general matrix in dense storage, created by the Matrix function, has class "dgeMatrix" but its cross-product has class "dpoMatrix". The solve methods for the "dpoMatrix" class use the Cholesky decomposition.