Comparing Least Squares Calculations

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

April 20, 2004

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

$$\widehat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \left\| \boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta} \right\|^2 \tag{1}$$

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

$$\widehat{\boldsymbol{\beta}} = \left(\boldsymbol{X}^{\mathsf{T}}\boldsymbol{X}\right)^{-1} \boldsymbol{X}^{\mathsf{T}}\boldsymbol{y} \tag{2}$$

when $\boldsymbol{X}$ has full column rank (i.e. the columns of $\boldsymbol{X}$ 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
```

```
> print(mm <- cbind(1, Formaldehyde$carb))

     [,1] [,2]
[1,]    1  0.1
[2,]    1  0.3
[3,]    1  0.5
[4,]    1  0.6
[5,]    1  0.7
[6,]    1  0.9

> print(y <- Formaldehyde$optden)

[1] 0.086 0.269 0.446 0.538 0.626 0.782
```

Using `t` to evaluate the transpose, `solve` to take an inverse, and the `%*%` operator for matrix multiplication, we can translate 2 into the S language as

```
> solve(t(mm) %*% mm) %*% t(mm) %*% y

            [,1]
[1,] 0.005085714
[2,] 0.876285714
```

On modern computers this calculation is performed so quickly that it cannot be timed accurately in R

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

[1] 0 0 0 0 0
```

and it provides essentially the same results as the standard `lm.fit` function that is called by `lm`.

```
> dput(c(solve(t(mm) %*% mm) %*% t(mm) %*% y))

c(0.00508571428571444, 0.876285714285715)

> dput(lm.fit(mm, y)$coefficients)

structure(c(0.00508571428571435, 0.876285714285714), .Names = c("x1",
"x2"))
```

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(mm, package = "Matrix")
> data(y, package = "Matrix")
> mm = as(mm, "matrix")
> dim(mm)
```

```
[1] 1850  712

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

[1] 2.64 0.11 2.75 0.00 0.00

> system.time(solve(t(mm) %*% mm) %*% t(mm) %*% y)

[1] 2.31 0.18 2.53 0.00 0.00

> system.time(solve(t(mm) %*% mm) %*% t(mm) %*% y)

[1] 2.35 0.16 2.56 0.00 0.00
```

(Here and in what follows we will repeat timings four times to obtain timings that are close to the steady-state values.)

Because the calculation of a "cross-product" matrix, such as $\boldsymbol{X}^\mathsf{T}\boldsymbol{X}$ or $\boldsymbol{X}^\mathsf{T}\boldsymbol{y}$, is a common operation in statistics, the `crossprod` function has been provided to do this efficiently. In the single argument form `crossprod(mm)` calculates $\boldsymbol{X}^\mathsf{T}\boldsymbol{X}$, taking advantage of the symmetry of the product. That is, instead of calculating the $712^2 = 506944$ elements of $\boldsymbol{X}^\mathsf{T}\boldsymbol{X}$ 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

$$\left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X}\right)\widehat{\boldsymbol{\beta}} = \boldsymbol{X}^\mathsf{T}\boldsymbol{y} \tag{3}$$

is solved directly.

Combining these optimizations we obtain

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

[1] 0.68 0.04 0.74 0.00 0.00

> system.time(solve(crossprod(mm), crossprod(mm, y)))

[1] 0.52 0.06 0.61 0.00 0.00

> system.time(solve(crossprod(mm), crossprod(mm, y)))

[1] 0.67 0.06 1.30 0.00 0.00

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

[1] TRUE
```

On this computer (2.0 GHz Pentium-4, 1 GB Memory, Goto's BLAS) 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 $\boldsymbol{X}^\mathsf{T}\boldsymbol{X}$ the naive way.

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

[1] 0.86 0.01 0.88 0.00 0.00

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

[1] 0.73 0.02 0.75 0.00 0.00

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

[1] 0.74 0.02 0.76 0.00 0.00

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

[1] 0.73 0.02 0.75 0.00 0.00
```

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

```
> ch = chol(crossprod(mm))
> system.time(chol(crossprod(mm)))

[1] 0.58 0.01 0.59 0.00 0.00

> system.time(chol(crossprod(mm)))

[1] 0.44 0.02 0.46 0.00 0.00

> system.time(chol(crossprod(mm)))

[1] 0.44 0.02 0.46 0.00 0.00

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

[1] 0.22 0.06 0.28 0.00 0.00
```

```
> system.time(backsolve(ch, forwardsolve(ch, crossprod(mm,
+     y), upper = TRUE, trans = TRUE)))

[1] 0.36 0.05 0.41 0.00 0.00

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

[1] 0.19 0.10 0.29 0.00 0.00

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

[1] TRUE
```

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 `"geMatrix"` but its cross-product has class `"poMatrix"`. The `solve` methods for the `"poMatrix"` class use the Cholesky decomposition.

```
> data(mm, package = "Matrix")
> mm = as(mm, "geMatrix")
> class(crossprod(mm))

[1] "poMatrix"
attr(,"package")
[1] "Matrix"

> Mat.sol = solve(crossprod(mm), crossprod(mm, y))
> system.time(solve(crossprod(mm), crossprod(mm, y)))

[1] 0.46 0.02 0.48 0.00 0.00

> system.time(solve(crossprod(mm), crossprod(mm, y)))

[1] 0.59 0.01 0.60 0.00 0.00

> system.time(solve(crossprod(mm), crossprod(mm, y)))

[1] 0.45 0.02 0.47 0.00 0.00

> all.equal(naive.sol, as(Mat.sol, "matrix"))

[1] TRUE
```

Furthermore, any method that calculates a decomposition or factorization stores the resulting factorization with the original object so that it can be reused without recalculation.

```
> xpx = crossprod(mm)
> xpy = crossprod(mm, y)
> system.time(solve(xpx, xpy))

[1] 0.08 0.00 0.08 0.00 0.00

> system.time(solve(xpx, xpy))

[1] 0.01 0.00 0.01 0.00 0.00

> system.time(solve(xpx, xpy))

[1] 0 0 0 0 0
```

The model matrix `mm` is sparse; that is, most of the elements of `mm` are zero. The `Matrix` package incorporates special methods for sparse matrices, which produce the fastest results of all.

```
> data(mm, package = "Matrix")
> class(mm)

[1] "cscMatrix"
attr(,"package")
[1] "Matrix"

> sparse.sol = solve(crossprod(mm), crossprod(mm, y))
> system.time(solve(crossprod(mm), crossprod(mm, y)))

[1] 0.06 0.00 0.06 0.00 0.00

> system.time(solve(crossprod(mm), crossprod(mm, y)))

[1] 0.07 0.00 0.07 0.00 0.00

> system.time(solve(crossprod(mm), crossprod(mm, y)))

[1] 0.06 0.00 0.06 0.00 0.00

> all.equal(naive.sol, as(sparse.sol, "matrix"))

[1] TRUE
```

As with other classes in the `Matrix` package, the `sscMatrix` retains any factorization that has been calculated although, in this case, the decomposition is so fast that it is difficult to determine the difference in the solution times.

```
> xpx = crossprod(mm)
> xpy = crossprod(mm, y)
> system.time(solve(xpx, xpy))

[1] 0.02 0.00 0.02 0.00 0.00

> system.time(solve(xpx, xpy))

[1] 0 0 0 0 0

> system.time(solve(xpx, xpy))

[1] 0 0 0 0 0
```

References

John M. Chambers. *Programming with Data*. Springer, New York, 1998. ISBN 0-387-98503-4. 5

Roger Koenker and Pin Ng. SparseM: A sparse matrix package for R. *J. of Statistical Software*, 8(6), 2003. 2