Comparing Least Squares Calculations

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

March 13, 2014

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

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

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
```

```
> (m <- 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

> (yo <- 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(m) %*% m) %*% t(m) %*% yo

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

On modern computers this calculation is performed so quickly that it cannot be timed accurately in R [1]

```
> system.time(solve(t(m) %*% m) %*% t(m) %*% yo)

   user  system elapsed
      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(m) %*% m) %*% t(m) %*% yo))

c(0.00508571428571428, 0.876285714285715)

> dput(unname(lm.fit(m, yo)$coefficients))

c(0.00508571428571408, 0.876285714285715)
```

[1] From R version 2.2.0, `system.time()` has default argument `gcFirst = TRUE` which is assumed and relevant for all subsequent timings

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
  2.112   0.007   2.124
```

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

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

is solved directly.

Combining these optimizations we obtain

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

   user  system elapsed
  0.584   0.004   0.589

> 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 $\boldsymbol{X}^\top\boldsymbol{X}$ the naive way.

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

```
   user  system elapsed
  0.724   0.003   0.729
```

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
  0.794   0.002   0.798

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

   user  system elapsed
  0.503   0.000   0.505

> 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
  0.566   0.002   0.570

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

   user  system elapsed
  0.006   0.000   0.006

> 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.

```
> mm <- as(KNex$mm, "dgeMatrix")
> class(crossprod(mm))

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

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

   user  system elapsed
  0.567   0.000   0.569

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

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))

   user  system elapsed
  0.062   0.000   0.061

> system.time(solve(xpx, xpy)) # reusing factorization

   user  system elapsed
  0.000   0.000   0.001
```

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.

```
> mm <- KNex$mm
> class(mm)

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

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

```
   user  system elapsed
  0.004   0.000   0.004
```

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

As with other classes in the `Matrix` package, the `dsCMatrix` 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))

   user  system elapsed
  0.002   0.000   0.001

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

   user  system elapsed
      0       0       0
```

Session Info

```
> toLatex(sessionInfo())
```

- R version 2.15.3 (2013-03-01), `x86_64-unknown-linux-gnu`
- Locale: `LC_CTYPE=de_CH.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_US.UTF-8`, `LC_COLLATE=C`, `LC_MONETARY=en_US.UTF-8`, `LC_MESSAGES=de_CH.UTF-8`, `LC_PAPER=C`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=de_CH.UTF-8`, `LC_IDENTIFICATION=C`
- Base packages: base, datasets, grDevices, graphics, methods, stats, tools, utils
- Other packages: Matrix 1.1-3
- Loaded via a namespace (and not attached): grid 2.15.3, lattice 0.20-27

```
> if(identical(1L, grep("linux", R.version[["os"]]))) { ## Linux - only ---
+  Scpu <- sfsmisc::Sys.procinfo("/proc/cpuinfo")
+  Smem <- sfsmisc::Sys.procinfo("/proc/meminfo")
+  print(Scpu[c("model name", "cpu MHz", "cache size", "bogomips")])
+  print(Smem[c("MemTotal", "SwapTotal")])
+ }

           _
model name Intel(R) Core(TM) i7-4765T CPU @ 2.00GHz
cpu MHz    2684.609
```

```
cache size 8192 KB
bogomips   3991.06

MemTotal  16319592 kB
SwapTotal 16777212 kB
```

References

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

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