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 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

(1)

where is an $ntimesp$ model matrix (pln), by is n-dimensional and beta$ispdimensional.Moststatisticstextsstatet$

1.1 A small example

As an example, let's create a model matrix, and corresponding response vector, for a simple linear regression model using the 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 to evaluate the transpose, to take an inverse, and the operator for matrix multiplication, we can translate **??** into the 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 From R version 2.2.0, system.time() has default argument gcFirst TRUE which is assumed and relevant for all subsequent timings

```
> 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)
```

1.2 A large example

For a large, ill-conditioned least squares problem, such as that described in koen:ng:2003, the literal translation of (**??**) does not perform well.

Because the calculation of a "cross-product" matrix, such as 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, taking advantage of the symmetry of the product. That is, instead of calculating the $712^2 = 506944$ elements of separately, it only calculates the $(712713)/2 = 253828$ 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 is solved directly.

Combining these optimization we obtain system.time(cpod.sol <- solve(crossprod(mm), crossprod(mm,y))) all.equal(naive.sol, cpod.sol)

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 the naive way. system.time(t(mm)

Note that in newer versions of and the BLAS library (as of summer 2007), is able to detect the many zeros in and shortcut many operations, and is hence much faster for such a sparse matrix than which currently does not make use of such optimization. This is not the case when is linked against an optimized BLAS library such as GOTO or ATLAS. Also, for fully dense matrices, 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) system.time(c2 <- crossprod(fm)) stopifnot(all.equal(c1, c2, tol = 1e-12))

1.3 Least squares calculations with Matrix classes

The 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 (**??**) 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))) system.time(chol.sol <- backsolve(ch, forwardsolve(ch, crossprod(mm, y), upper = TRUE, trans = TRUE))) stopifnot(all.equal(chol.sol, naive.sol))

The Matrix package uses the S4 class system R: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$mm, "denseMatrix")class(crossprod(mm)$
$-solve(crossprod(mm), crossprod(mm, y)))stopifnot(all.equal(naive.sol, unname(as(Mat.sol, "matrix"))))$

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. "'r xpx <- crossprod(mm) xpy <- crossprod(mm, y) system.time(solve(xpx, xpy)) "' 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. "'r mm <- KNex$mmclass(mm)system.time(sparse.sol < -solve(crossprod(mm), crossprod(mm, y)))stopifnot(all.equa$
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. "'r

xpx <- crossprod(mm) xpy <- crossprod(mm, y) system.time(solve(xpx, xpy)) system.time(solve(xpx, xpy)) "'

Session Info

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

- R version 4.1.2 (2021-11-01), `x86_64-pc-linux-gnu`
- Locale: `LC_CTYPE=en_US.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_US.UTF-8`, `LC_COLLATE=en_US.UTF-8`, `LC_MONETARY=en_US.UTF-8`, `LC_MESSAGES=en_US.UTF-8`, `LC_PAPER=en_US.UTF-8`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_US.UTF-8`, `LC_IDENTIFICATION=C`
- Running under: `Pop!_OS 22.04 LTS`
- Matrix products: default
- BLAS: `/usr/lib/x86_64-linux-gnu/blas/libblas.so.3.10.0`

- LAPACK:
  `/usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.10.0`
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Loaded via a namespace (and not attached): compiler 4.1.2, tools 4.1.2

```
> 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) i5-7200U CPU @ 2.50GHz
cpu MHz    3100.063
cache size 3072 KB
bogomips   5399.81
          _
MemTotal  3782692 kB
SwapTotal 7976436 kB
```