Efficient computing with high-dimensional matrices in R, with applications to penalized regression

Mark A. van de Wiel*

Dept of Epidemiology and Data Science, Amsterdam Public Health research institute, Amsterdam University Medical Centers, Amsterdam, The Netherlands; mark.vdwiel@amsterdamumc.nl

1 Introduction

Modern statistical methods often require operations on high-dimensional matrices: matrices with many more columns (variables; p) than rows (samples; n). Below, we present several tricks to efficiently execute these operations in R. We always present the naive solution first and then the more clever one. These tricks work for general matrices, hence also for dense ones. Note that for sparse matrices (many zeros), many additional tricks are available, also in terms of storage. We start by presenting some general tricks for operations on large matrices, after which we focus on more advanced tricks that are particularly useful in penalized regression settings. Throughout, we illustrate calculations using the following matrices $L_{p\times n}$ and $R_{n\times p}$, plus vector $\mathbf{v_n}$ and its matrix representation $V_{n\times 1}$:

```
p <- 5000; n<- 200
set.seed(46234) #for reproducibility
L <- matrix(rnorm(n*p),nrow=p)
R <- matrix(rnorm(n*p),nrow=n)
v <- rnorm(p) #vector
V <- matrix(v,nrow=p)
diagV <- diag(v)
```

```
dim(L)
[1] 5000 200

dim(R)
[1] 200 5000

length(v)
[1] 5000

dim(V)
[1] 5000 1
```

2 General tricks

When multiplying large matrices, the order of multiplication does not impact the results, but does matter a lot for computational efficiency. Basically, one should try to avoid generating large matrices whenever possible.

Order matters

Below, we show two ways to compute the product LRV:

```
system.time(A1 <- L %*% R %*% V)
user system elapsed
4.032 0.070 5.120

system.time(A2 <- L %*% (R %*% V))
user system elapsed
0.002 0.000 0.002

sum(A1-A2)
[1] 3.066676e-10
```

Here, the second is much faster as it avoids computing the $p \times p$ matrix $C = LR$.

Computing the diagonal or trace

Often, we only need the diagonal of a matrix product, e.g. when computing variances or a trace. Again, we should avoid computing $C = LR$, and instead compute the diagonal directly by element-wise multiplication:

```
system.time(D1 <- diag(L %*% R))
user system elapsed
3.860 0.096 4.957

system.time(D2 <- rowSums(t(L) * R))
user system elapsed
0.008 0.004 0.012

#check trace
sum(D1)
[1] 620.5024

sum(D2)
[1] 620.5024
```

Combining change of order and element-wise multiplication

Now suppose we need to compute $tr(CC^T)$. Then, we combine the above two tricks (change of order and element-wise multiplication) to render an enormous efficiency gain:

```
#very inefficient
system.time(C <- L %*% R)  + system.time(TC1 <- sum(diag(C %*% t(C))))
user  system elapsed
76.77    0.15   77.23

#faster
system.time(C <- L %*% R)
+ system.time(TC2 <- sum(rowSums(t(C) * t(C))))
user  system elapsed
2.46    0.01    2.48

#fastest
system.time(LRRT <- (L %*% (R %*% t(R))))
+ system.time(TC3 <- sum(rowSums(t(LRRT) * t(L))))
user  system elapsed
0.20    0.02    0.23

TC1
[1] 4999298328

TC2
[1] 4999298328

TC3
[1] 4999298328
```

Here, we used $CC^T = LRR^TL^T = L(RR^T)L^T$.

Matrix times diagonal matrix

Many statistical algorithms, such as iterative weighted least squares, require to 'weigh' the entries of a matrix, which is effectuated by multiplying it with a diagonal weight matrix. When multiplying a large matrix with a diagonal one, it is much faster to make use of element-wise multiplication with the vector containing the diagonal elements:

```
#Matrix times diagonal matrix
system.time(RV1 <- R %*% diagV)
user  system elapsed
2.00    0.00    1.98
```

```
system.time(RV2 <- t(t(R) * v))
user  system elapsed
0.02    0.00    0.00

RV1[1:3,1:3]
            [,1]        [,2]      [,3]
[1,] -1.40074426  0.59430401 -0.590674
[2,]  0.08570781 -0.06474478 -1.151878
[3,] -0.01083104  0.42512389  1.827297

RV2[1:3,1:3]
            [,1]        [,2]      [,3]
[1,] -1.40074426  0.59430401 -0.590674
[2,]  0.08570781 -0.06474478 -1.151878
[3,] -0.01083104  0.42512389  1.827297

sum(RV1-RV2)
[1] 0

#Likewise, diagonal matrix times matrix
system.time(VL1 <- diagV %*% L)
user  system elapsed
   3       0       3

system.time(VL2 <- v * L)
user  system elapsed
0.01    0.00    0.00

VL1[1:3,1:3]
           [,1]       [,2]       [,3]
[1,] -0.1871432 -0.7300480 -0.6626868
[2,] -0.3461059  1.2572818 -2.3818841
[3,] -0.8808107  0.7403945  0.4311718

VL2[1:3,1:3]
           [,1]       [,2]       [,3]
[1,] -0.1871432 -0.7300480 -0.6626868
[2,] -0.3461059  1.2572818 -2.3818841
[3,] -0.8808107  0.7403945  0.4311718

sum(VL1-VL2)
[1] 0
```

Solve instead of inverse

In regression, we often have to compute $x = A^{-1}v$, with A a square matrix. As x is the solution of $Ax = v$, it is faster to solve this linear system than to explicitly compute A^{-1}:

```
p <- 5000; n <- 200
set.seed(46234) #for reproducibility
A <- matrix(rnorm(p*p),nrow=p)
v <- rnorm(p)

system.time(AinvV1 <- solve(A) %*% v)
user  system elapsed
113.51  0.41  114.49

system.time(AinvV2 <- solve(A,v))
user  system elapsed
28.86    0.23   30.11

AinvV1[1:3]
[1] 0.45278463  0.78038634 -0.09664263

AinvV2[1:3]
[1]  0.45278463  0.78038634 -0.09664263

sum(AinvV1-AinvV2)
[1] -2.039108e-11
```

3 Using Woodbury's identity and more

In (penalized) regression we often need to compute:

$$B = (X^T X + D)^{-1} X^T Y,$$

where X is a $n \times p$ design matrix, D is a $p \times p$ diagonal penalty matrix and Y is a response vector of length n. Algorithms like (generalized) ridge regression, but also many implementations of Bayesian regression require repeated calculations of B and related quantities, such as hat matrix $H = XB$. The main problem is that in high-dimensional settings $(X^T X + D)$ is a big $p \times p$ matrix.

3.1 Woodbury's identity

Woodbury's identity avoids having to generate (and invert) that large matrix. It states:

$$(X^T X + D)^{-1} = D^{-1} - D^{-1} X^T (I_{n \times n} + X D^{-1} X^T)^{-1} X D^{-1},$$

implying

$$B = (X^T X + D)^{-1} X^T Y = D^{-1} X^T Y - D^{-1} X^T (I_{n \times n} + X_D)^{-1} X_D Y,$$

with $X_D = X D^{-1} X^T$. To compute B with the above equation, many of the presented tricks are of use. We first generate X, Y and D:

```
p <- 5000; n <- 200
set.seed(46234) #for reproducibility
X <- matrix(rnorm(n*p),nrow=n)
d <- rnorm(p)
Y <- matrix(rnorm(n),ncol=1)
di <- 1/d
D <- diag(d)
```

Now, let's compare the naive solution with a much more efficient one:

```
#slow
system.time(B1 <- solve(t(X) %*% X + D) %*% t(X) %*% Y)
user  system elapsed
109.48    0.12  109.87

#much faster with Woodbury
pmt<-proc.time()
#Diagonal times matrix
DiXt <- di*t(X) #D^(-1)X^T
XD <- X %*% DiXt
IsumXD <- diag(n) + XD #I_(nxn) + XD
XDY <- XD %*% Y
#Inverse times vector using solve
InvXDY <- solve(IsumXD,XDY) #(I_(nxn) + XD)^(-1) XD Y
B2 <- DiXt %*% (Y - InvXDY) #B
proc.time()-pmt
user  system elapsed
0.14    0.05    0.18

#check
B1[1:3,]
[1]  0.02796713 -0.41685524 -0.63964106

B2[1:3,]
[1]  0.02796713 -0.41685524 -0.63964106

sum(B1-B2)
[1] -6.248562e-09
```

3.2 (Piecewise) constant diagonal matrix D

For large p the matrix multiplication $XD^{-1}X^T$ takes most of the time:

```
p <- 10000; n <- 200
set.seed(46234) #for reproducibility
X <- matrix(rnorm(n*p),nrow=n)
d <- rnorm(p)
Y <- matrix(rnorm(n),ncol=1)
di <- 1/d
D <- diag(d)

pmt<-proc.time()
DiXt <- di*t(X)
system.time(XD <- X %*% DiXt) #matrix multiplication time
user  system elapsed
0.23    0.00    0.24

IsumXD <- diag(n) + XD
XDY <- XD %*% Y
InvXDY <- solve(IsumXD,XDY)
B2 <- DiXt %*% (Y - InvXDY)
proc.time()-pmt #total computing time
user  system elapsed
 0.32    0.00    0.31
```

Many algorithms require evaluations of $\Gamma_D = XD^{-1}X^T$ for multiple values of D. For example, when D is a penalty matrix, and tuning of the penalty parameter (e.g. by cross-validation) is desired. Or when MCMC updates of D are required in a Bayesian regression setting. Then, a substantial computational gain is feasible when the elements of D are (blockwise) constant. Denote the value of D that corresponds to the bth block by d_b, its corresponding indices by i_b and the matrix with corresponding columns of X by $X_b = X[, i_b]$. Then, we have:

$$\Gamma_D = XD^{-1}X^T = \sum_{b=1}^{B} d_b^{-1}\Sigma_b, \quad \Sigma_b = X_b X_b^T. \tag{1}$$

This implies that storing the $n \times n$ matrices Σ_b can render a potentially large computational gain when Γ_D needs to be computed for many values of D. Let us first verify (1):

```
p <-  10000
set.seed(46234) #for reproducibility
X <- matrix(rnorm(n*p),nrow=n)

# Two blocks
```

```
di1 <- 1/rep(abs(rnorm(1)),p/2)
di2 <- 1/rep(abs(rnorm(1)),p/2)
di <- c(di1,di2)
X1 <- X[,1:(p/2)]
X2 <- X[,-(1:p/2)]

# naive
system.time(Gd1 <- X %*% (di*t(X)))
user  system elapsed
0.21    0.01    0.23

#alternative
Sigma1 <- X1 %*% t(X1)
Sigma2 <- X2 %*% t(X2)
system.time(Gd2 <- di1*Sigma1 + di2*Sigma2)
user  system elapsed
   0       0       0

Gd1[1:3,1:3]
           [,1]        [,2]        [,3]
[1,] 11983.7552  -196.89345  -109.40583
[2,]  -196.8934 11931.42654   -46.45444
[3,]  -109.4058   -46.45444 11913.23992

Gd2[1:3,1:3]
           [,1]        [,2]        [,3]
[1,] 11983.7552  -196.89345  -109.40583
[2,]  -196.8934 11931.42654   -46.45444
[3,]  -109.4058   -46.45444 11913.23992

max(abs(Gd1-Gd2))
[1] 1.000444e-10
```

Then, the computational gain is evident when we calculate the computing time for 100 different values of d. Below we approximate this by simply multiplying the computing time for a single d by 100, as the latter does not depend on the particular value of d.

```
#naive
100*system.time(Gd1 <- X %*% (di*t(X)))
user  system elapsed
  22       0      20

#much faster
system.time(Sigma1 <- X1 %*% t(X1)) +
system.time(Sigma2 <- X2 %*% t(X2)) +
```

```
100*system.time(Gd2 <- di1*Sigma1 + di2*Sigma2)
user  system elapsed
0.15    0.00    0.18
```

Of note, as Σ_b in (1) is an $n \times n$ cross-sample matrix it does not need to be recomputed when performing cross-validation. Its cross-validation counterpart is obtained by simply selecting the rows (and the same columns) of Σ_b corresponding to the samples in the fold. This provides another substantial computational gain.

4 Discussion

All computations were performed in `R 4.3.1` on a laptop with a `Windows 11` OS, a `12th Gen Intel(R) i5-1235U (1.30 GHz)` core and `16,0 GB` RAM. Our computations were performed in base `R`. Additional speed-ups are feasible when using specialized packages such as `Rfast` [1], or by invoking languages like `C++` using `Rcpp` [2]. The tricks shown here were used in [3, 4, 5, 6, 7]. Extensions for non-diagonal penalty matrices are provided in [8, 9]. Finally, this is a 'living' document. If you have corrections or suggestions, please notify the corresponding author.

References

[1] Michail Tsagris and Manos Papadakis. Taking R to its limits: 70+ tips. *PeerJ Preprints*, 6:e26605v1, 2018.

[2] Dirk Eddelbuettel and James Joseph Balamuta. Extending R with C++: a brief introduction to Rcpp. *The American Statistician*, 72(1):28–36, 2018.

[3] M. A. van de Wiel, T. G. Lien, W. Verlaat, W. N. van Wieringen, and S. M. Wilting. Better prediction by use of co-data: adaptive group-regularized ridge regression. *Statist Med*, 35(3):368–381, 2016.

[4] M. M. Münch, C. F. W. Peeters, A. W. van der Vaart, and M. A. van de Wiel. Adaptive group-regularized logistic elastic net regression. *Biostatistics*, 22(4):723–737, 2021.

[5] Mark A van de Wiel, Mirrelijn M van Nee, and Armin Rauschenberger. Fast cross-validation for multi-penalty high-dimensional ridge regression. *Journal of Computational and Graphical Statistics*, pages 1–13, 2021.

[6] M. M. van Nee, L. F. A. Wessels, and M. A. van de Wiel. Flexible co-data learning for high-dimensional prediction. *Statist Med*, 40(26):5910–5925, 2021.

[7] Claudio Busatto and Mark A van de Wiel. Informative co-data learning for high-dimensional horseshoe regression. *Biometrical Journal*, 68(1):e70105, 2026.

[8] Annelinde Lettink, Mai Chinapaw, and Wessel N van Wieringen. Two-dimensional fused targeted ridge regression for health indicator prediction from accelerometer data. *Journal of the Royal Statistical Society Series C: Applied Statistics*, 72(4):1064–1078, 2023.

[9] Jeroen M Goedhart, Mark A van de Wiel, Wessel N van Wieringen, and Thomas Klausch. Fusion of tree-induced regressions for clinico-genomic data. *arXiv preprint arXiv:2411.02396*, 2024.