Dense and Sparse Matrices

Dom Owens

12/11/2019

Dense Matrices

Matrices are 2-dimensional data structures. In R, entries are specified by column by default. Names can be reassigned without changing values.

```
x <- matrix(1:6, 2, 3) #2 by 3 matrix
x1 <- matrix(1:6, 3, 2, dimnames = list(c("X","Y","Z"), c("A","B"))) #give names to rows and columns
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
x1
```

```
##   A B
## X 1 4
## Y 2 5
## Z 3 6
```

```
rownames(x1) <- c("R1","R2","R3") #add row names
x1
```

```
##    A B
## R1 1 4
## R2 2 5
## R3 3 6
```

We can preserve matrix dimensions when selecting rows with `drop=F`. Note that names are maintained.

```
x1[1,,drop=F] #select row 1
```

```
##    A B
## R1 1 4
```

Arrays are higher-dimensional data structures.

```
y <- array(1:8, c(2,2,2)) #3D data object
y #prints as transects
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
```

Linear Systems

Matrix operations are useful for solving linear systems of the form $Ax = b$. Multiple approaches are available to do this.

We can try this on **Hilbert matrices**; these are close to singular, so are hard to invert.

`solve(A) %*% b` inverts A then multiplies by b.

Alternatively, we use `solve(A,b)`.

```
set.seed(123)
library(Matrix)
n <- 9
A <- as.matrix(Hilbert(n)) #generate Hilbert matrix
x <- matrix(runif(n), n, 1) #randomise x
b <- A%*%x # compute b
x1 <- solve(A) %*% b #method 1
x2 <- solve(A,b) #method 2
norm(x-x1, type = "1") # find numerical errors
```

```
## [1] 0.0001934015
```

```
norm(x - x2 ,type = "1")
```

```
## [1] 3.206041e-05
```

Clearly, the latter outperforms the former in terms of accuracy.

Numerical Stability and Precision

Why has this happened? We should examine the computation underlying these divergences.

In `R`, floating point numbers are stored as *double precision* numbers. 64 bits store a representation of the number: the **sign** in 1 bit, the **exponent** (i.e. magnitude) in 11 bits, and **precision** in 52 bits. Hence have the largest number available:

```
2^1023 + 2^1022.9999999999999 #greatest number representable by R
```

```
## [1] 1.797693e+308
```

```
2^1024 #not representable
```

```
## [1] Inf
```

With small values, we can expect approximation errors.

```
1 + 1e-15 #not equal to 1
```

```
## [1] 1
```

```
print(1 + 1e-15, digits=22)
```

```
## [1] 1.000000000000001110223
```

The operator `==`, which checks for equality, struggles for floating point digits.

```
0.1 + 0.2 -0.3 == 0 #Should return TRUE
```

```
## [1] FALSE
```

From the above, we conclude that we should use `solve(A,b)` to minimise errors.

Eigenvalues

`eigen` returns the **eigenvalues** and **eigenvectors** of the input matrix. We should specify the argument `symmetric = TRUE` if the input matrix is symmetric to reduce error in computation.

We demonstrate how the calculations differ below.

```
n <- 10
A1 <- matrix(rnorm(n*n), nrow=n, ncol=n) #n by n matrix of standard normal samples
A2 <- A1 + t(A1)
eA <- eigen(A2, symmetric=TRUE) #find eigenvalues
summary(abs(eA$values))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.1392  1.2962  2.9755  3.5310  5.7445  8.2920
```

```
c(norm(eA$vectors %*% diag(eA$values) %*% t(eA$vectors) - A2, type='1'),
  norm(eA$vectors %*% t(eA$vectors) - diag(rep(1, n)), type='1')) #calculate divergence
```

```
## [1] 5.739853e-14 5.224826e-15
```

And with `symmetric = TRUE`

```
n <- 100
A1 <- matrix(rnorm(n*n), nrow=n, ncol=n); A2 <- A1 + t(A1)
eA <- eigen(A2, symmetric=TRUE) #find eigenvalues
summary(abs(eA$values))
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
##  0.008361  5.536487 11.357499 11.788913 17.657582 26.677891
```

```
c(norm(eA$vectors %*% diag(eA$values) %*% t(eA$vectors) - A2, type='1'),
  norm(eA$vectors %*% t(eA$vectors) - diag(rep(1, n)), type='1')) #calculate divergences
```

```
## [1] 2.705893e-12 1.954209e-13
```

These are larger by orders of magnitude. We see that specifying that the matrix is symmetric reduces precision error.

Sparse Matrices

Sparse matrices, which consist mostly of zeros, are more of a challenge for a language to handle. It is not possible to store them in the same format as dense matrices due to memory constraints. `Matrix` stores dense matrices as `dgeMatrix` objects; `rankMatrix` will return the rank, and `rcond` returns the condition number; this quantifies the divergence between min and max eigenvalues.

```
rankMatrix(A) #A is full rank
```

```
## [1] 9
## attr(,"method")
## [1] "tolNorm2"
## attr(,"useGrad")
## [1] FALSE
## attr(,"tol")
## [1] 1.998401e-15
```

```r
rcond(A) #condition number
```

```
## [1] 9.093786e-13
```

Sparse matrices are stored by default as `dgCMatrix` objects.

We construct two sparse matrices and observe the memory difference:

```r
nrows <- 1000
ncols <- 1000
vals <- sample(x=c(0, 1, 2), prob=c(0.98, 0.01, 0.01), size=nrows*ncols, replace=TRUE) #sample 1000*100
m1 <- matrix(vals, nrow=nrows, ncol=ncols) # dense matrix representation
m2 <- Matrix(vals, nrow=nrows, ncol=ncols, sparse = TRUE) #sparse matrix representation
m1[1:2, 1:10]
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    0    0    0    0    0    0    0    0    0     0
## [2,]    0    0    0    0    0    0    0    0    0     0
```

```r
c(object.size(m1), object.size(m2)) #compare memory useage
```

```
## [1] 8000200  243448
```

The dense matrix `m1` uses 3 times as much space.

It is possible to coerce a `dgC` into a `dgT`, but not a `dgC` into a `dgR`

```r
object.size(as(m2, 'dgeMatrix')) # `dgC` into a `dgT` possible
```

```
## 8001112 bytes
```

```r
object.size(as(m1, 'dgRMatrix')) # `dgC` into a `dgR` not possible
```

```
## 243448 bytes
```

Operations

Matrix operations on these objects may change or conserve the type.

Addition converts a `dcG` into a `dge`

```r
B <- as(matrix(c(1,0,0,0,2,0), nrow=3, ncol=2), 'dgCMatrix')
B + 10
```

```
## 3 x 2 Matrix of class "dgeMatrix"
##      [,1] [,2]
## [1,]   11   10
## [2,]   10   12
## [3,]   10   10
```

Mutliplication by a dense vector outputs a `dge` matrix

```r
B %*% c(1,1)
```

```
## 3 x 1 Matrix of class "dgeMatrix"
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    0
```

Multiplication by a sparse matrix, or taking a transpose, preserves sparsity.

```
B %*% Matrix(c(1, 1), nrow=2, ncol=1, sparse=TRUE)
```

```
## 3 x 1 sparse Matrix of class "dgCMatrix"
##
## [1,] 1
## [2,] 2
## [3,] .
```

```
t(B)
```

```
## 2 x 3 sparse Matrix of class "dgCMatrix"
##
## [1,] 1 . .
## [2,] . 2 .
```

Solving large linear systems

Inverting sparse matrices is esepcially difficult, given the inverse of the matrix is not guaranteed to be sparse. We see this with a tridiagonal matrix

```
n <- 100
A1 <- bandSparse(n, k=c(0,1), diag=list(rep(1,n), rep(-0.2, n)), symm=T) #Tridiagonal
A1inv <- solve(A1)
c(object.size(A1), object.size(A1inv))
```

```
## [1]  11488 121824
```

```
sum(abs(A1inv) < 1e-100) #no entries near to 0
```

```
## [1] 0
```