STAT-6494 Advanced Statistical Computing with R

Homework 3

Wenjie Wang

16 January 2017

1 Exercise: Linear Model with Sparse Model Matrix

Write **R** functions for linear regression with two inputs, model matrix `x` and response vector `y`. One function `lmQr` uses the QR decomposition of x and other `lmChol` uses the Cholesky decomposition of $x'x$. Each function returns a list with three components: `coefficients` for estimated regression coefficients; `stderr` for standard error of the estimates; and `df.residuals` for the degrees of freedom of the residuals. Compare the performance with `lm.fit` from **R** for both dense matrix and sparse matrix of `x` with various dimensions and sparsity.

1.1 Function `lmQr` and `lmChol`

The function `lmQr` is designed using QR decomposition on design matrix `X` to fit linear regression model. It takes response vector `Y` and design matrix `X`, which can be sparse matrices object in `dgCMatrix` class produced by pacakge **Matrix**. And it returns `coefficients` for estimated coefficients, `stderr` for standard error of the estimates, and `df.residuals` for the degrees of freedom of the residuals. The function `lmChol` is similarly designed but using Cholesky decomposition.

The straightforward implementation is done completely in R. Note that for `lmQr` and `lmChol`, the design matrix `X` input can be either dense matrix class object or sparse matrix class object, `dgCMatrix` class object. The corresponding method will be called depending on the object class.

```
library(Matrix)
lmQr <- function(X, y) {
    resQR <- qr(X)
    betaEst <- qr.coef(resQR, y)
    df <- nrow(X) - (Xp <- ncol(X))
    if (is.matrix(X)) {
        matR <- qr.R(resQR)
        xtxInv <- backsolve(matR, backsolve(matR, diag(Xp),
                                            transpose = TRUE))
    } else {
        matR <- qrR(resQR)
        xtxInv <- solve(crossprod(X))
    }
    sigma2Est <- crossprod(y - X %*% betaEst) / df
    stderr <- sqrt(as.numeric(sigma2Est) * diag(xtxInv))
    list(coefficients = betaEst, stderr = stderr,
         df.residuals = df)
}
lmChol <- function(X, y) {
    resChol <- chol(crossprod(X))
    RHS <- crossprod(X, y)
    xtxInv <- chol2inv(resChol)
    betaEst <- xtxInv %*% RHS
```

```
    df <- nrow(X) - ncol(X)
    sigma2Est <- crossprod(y - X %*% betaEst) / df
    stderr <- sqrt(as.numeric(sigma2Est) * diag(xtxInv))
    list(coefficients = betaEst, stderr = stderr,
         df.residuals = df)
}
```

1.2 Performance Comparison on Simulated Datasets

We generate simulated datasets from linear regression model with various dimension and sparsity. Then we compare the performance of `lm.fit`, `lmQr`, and `lmChol` on those datasets, respectively.

1.2.1 Dense matrix of small dimension

The response `Y` is set as a vector with length 500. The design matrix is a 500 by 10 matrix. We compare the computing performance of `lmQr`, `lmChol` with `lm.fit` from package **stats** with the help of package **microbenchmark** over the simulated `Y` and `X` as follows:

```
set.seed(1216)
n <- 500
p <- 10
X <- matrix(rnorm(n * p), n)
y <- rowSums(X) + rnorm(n)
library(microbenchmark)
(ans <- microbenchmark(lm.fit(X, y), lmQr(X, y),
                       lmChol(X, y), times = 1000))
```

```
Unit: microseconds
         expr min  lq mean median  uq  max neval cld
 lm.fit(X, y) 153 158  175    161 164 1487  1000  b
   lmQr(X, y) 223 231  270    235 239 2870  1000   c
 lmChol(X, y) 109 114  121    117 120 1615  1000 a
```

```
ggplot2::autoplot(ans)
```

From the comparison shown in Figure 1, we find that `lmChol` is the fastest one. `lm.fit` is a little bit slower than `lmChol` but faster than `lmQr`.

1.2.2 Sparse matrix of small dimension

The sparsity is set to be 0.9.

```
r <- 0.9
X[sample(n * p, floor(n * p * r))] <- 0
Xsp <- methods::as(X, "dgCMatrix")
(ans <- microbenchmark(lm.fit(X, y), lmQr(X, y), lmQr(Xsp, y),
                       lmChol(X, y), lmChol(Xsp, y), times = 100))
```

```
Unit: microseconds
           expr  min   lq mean median   uq   max neval cld
   lm.fit(X, y)  145  157  180    162  172  1658   100 a
     lmQr(X, y)  223  239  300    250  283  1678   100 a
   lmQr(Xsp, y) 1509 1556 1727   1601 1677 11526   100   c
```

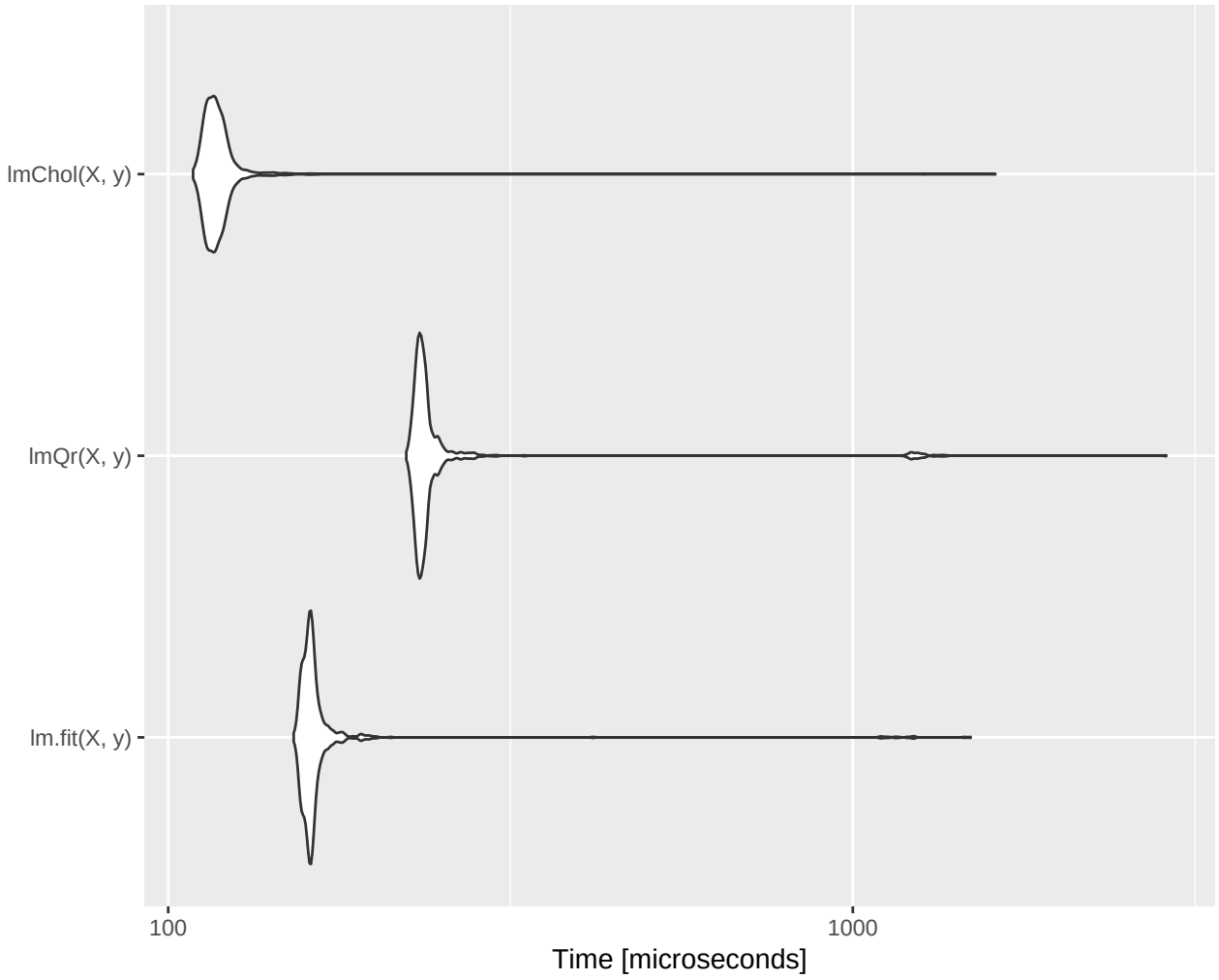

Figure 1: Performance comparison when design matrix is a dense matrix of small dimension

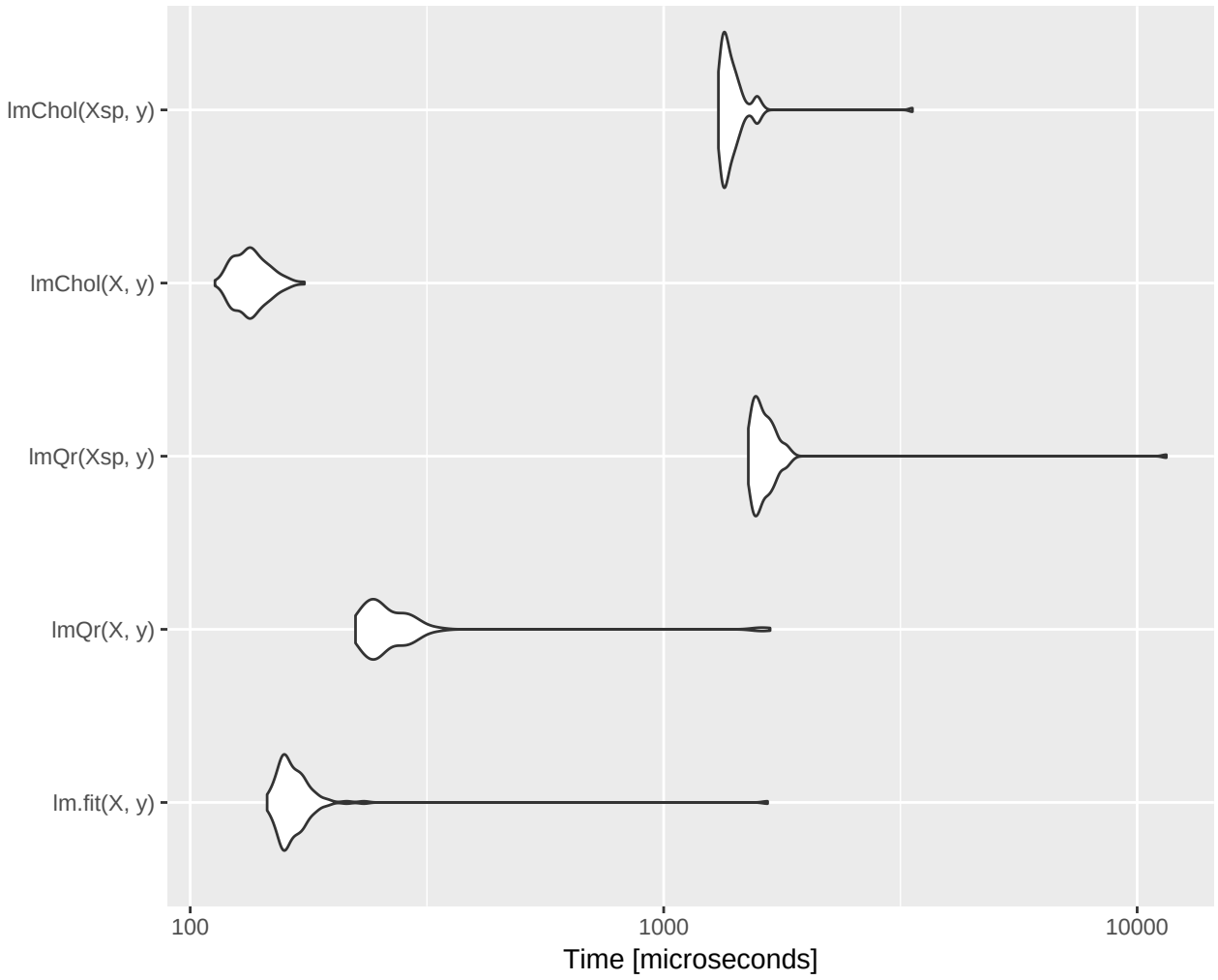

Figure 2: Performance comparison when design matrix is a sparse matrix of small dimension

```
   lmChol(X, y)  113  126  135    134  142   174   100 a
 lmChol(Xsp, y) 1305 1337 1410   1367 1422  3353   100  b
```

```
ggplot2::autoplot(ans)
```

From the comparison shown in Figure 2, we find that `lmChol` on dense matrix objects is still the fastest one. `lm.fit` is a little bit slower than `lmChol` but faster than `lmQr` on dense matrix objects. Both `lmChol` and `lmQr` run much slower when the object input is a sparse matrix class object instead of a dense matrix object. It indicates that sparse matrix objects and their corresponding methods cannot bring much help when the dimension of the sparse matrix is relatively small.

1.2.3 Dense matrix of large dimension

The response `Y` is set as a vector with length 500. The design matrix is a 5,000 by 200 matrix. Similarly, the performance comparison is shown in Figure 3

```
set.seed(1216)
n <- 5e3
p <- 2e2
```

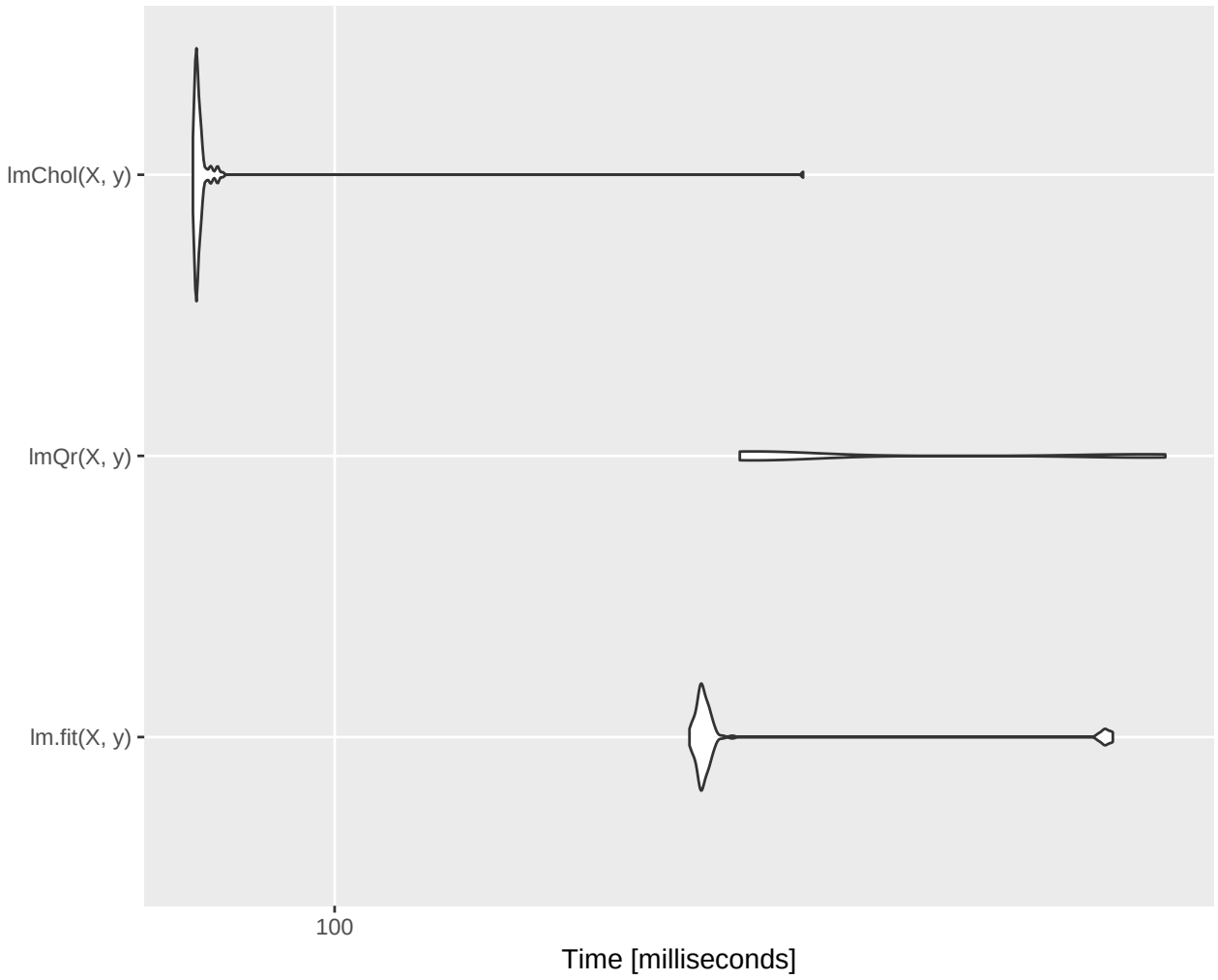

Figure 3: Performance comparison when design matrix is a dense matrix of large dimension

```
X <- matrix(rnorm(n * p), n)
y <- rowSums(X) + rnorm(n)
(ans <- microbenchmark(lm.fit(X, y), lmQr(X, y), lmChol(X, y), times = 100))
```

```
Unit: milliseconds
         expr   min    lq  mean median    uq max neval cld
 lm.fit(X, y) 141.1 142.4 151.1  142.9 143.8 213   100  b
   lmQr(X, y) 148.1 149.6 168.8  150.5 216.9 224   100   c
 lmChol(X, y)  87.2  87.4  88.4   87.5  87.8 157   100 a
```

```
ggplot2::autoplot(ans)
```

When a dense design matrix of large dimension is given, `lmChol` is still the fastest one. `lm.fit` is a little bit slower than `lmChol` but faster than `lmQr`.

1.2.4 Sparse matrix of large dimension

The sparsity is set to be 0.9.

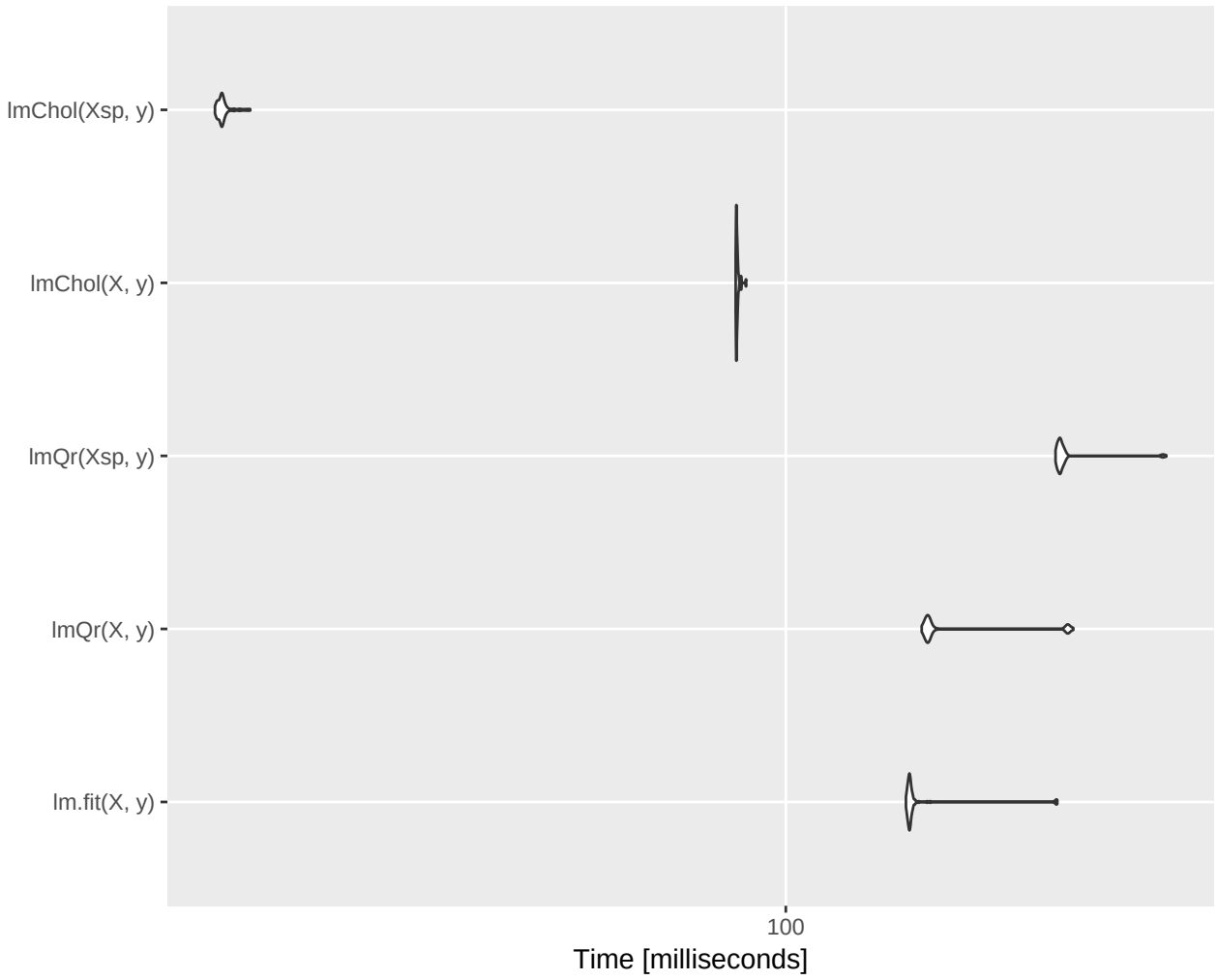

Figure 4: Performance comparison when design matrix is a sparse matrix of large dimension

```
r <- 0.9
X[sample(n * p, floor(n * p * r))] <- 0
Xsp <- methods::as(X, "dgCMatrix")
(ans <- microbenchmark(lm.fit(X, y), lmQr(X, y), lmQr(Xsp, y),
                       lmChol(X, y), lmChol(Xsp, y), times = 100))
```

```
Unit: milliseconds
           expr   min    lq  mean median    uq   max neval   cld
   lm.fit(X, y) 138.9 139.8 143.9  140.3 140.8 210.0   100   c
     lmQr(X, y) 145.0 146.8 161.9  147.7 149.5 219.4   100    d
   lmQr(Xsp, y) 209.0 210.6 215.9  211.8 213.2 282.9   100     e
   lmChol(X, y)  87.2  87.3  87.6   87.4  87.6  89.7   100  b
 lmChol(Xsp, y)  21.0  21.3  21.5   21.4  21.6  23.1   100 a
```

```
ggplot2::autoplot(ans)
```

From the comparison shown in Figure 4, we find that `lmChol` on sparse matrix objects becomes the fastest one. `lmChol` becomes much slower when a dense matrix object is specified, which means that the sparse matrix class object and its methods take advantage of the sparsity of the matrix and the advantage grows

larger when the dimension grows. What's more, `lm.fit` is slower than `lmChol` but faster than `lmQr`. On the contrary, `lmQr` runs slower when a sparse matrix class object is input, which means those existing methods for `dgCMatrix` fail to take advantage of the sparsity after QR decomposition. (Currently, there is no `backsolve` or similar function in package **Matrix**. Package **SparseM** provides method `backsolve` for different class objects.)