Assignment 4

1. The Longley data, available as the variable `longley` in the package `datasets`, provides a well-known example for a highly collinear regression, in particular in the regression of the `Employed` variable on the other six variables plus an intercept. Using this data set as an illustration, this problem explores the accuracy of the QR and Cholesky factorization approaches for fitting a regression.

   (a) Write an R function that uses the package `gmp` to compute the exact coefficients, rounded to the nearest double precision numbers, of a least squares fit of a vector y to the columns of a matrix X. You may find the functions `as.bigq`, `solve`, and `as.double` useful. Your function should allow the arguments to be either floating point or arbitrary precision rational numbers.

   (b) Write an R function that uses the Cholesky factorization of the cross product matrix to compute the coefficients of a least squares fit of a vector y to the columns of a matrix X. Your function should take an optional argument `center` with default `FALSE`; if `center` is `TRUE` then X should contain only non-constant columns and you should mean center the columns before forming the Cholesky factorization (the functions `sweep` and `apply` or `colMeans` may be useful). When centering is used the model includes an intercept, which should be estimated and included in the result.

   (c) Use these functions and the function `lm.fit`, which uses QR factorization, to compare the accuracy of coefficient estimates obtained by the QR and Cholesky approaches for the Longley data. Does it help to apply the Cholesky factorization to mean-centered data? For obtaining the exact coefficients, keep in mind that the `longley` data set is a floating point approximation to the actual decimal data. Looking at the printed representation should show you how to convert this approximation to the exact values as rationals.

2. A random vector Y has a multivariate normal distribution with mean zero and covariance matrix

$$C = \begin{bmatrix} 1 & a & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ a & 1 & a & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & a & 1 & a & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & a & 1 & a & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \dots & a & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \dots & 1 & a & 0 \\ 0 & 0 & 0 & 0 & 0 & \dots & a & 1 & a \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & a & 1 \end{bmatrix}$$

The log-likelihood for an observed vector y, dropping additive constants, is

$$-\frac{1}{2}\log\det C - \frac{1}{2}y^T C^{-1} y$$

(a) Write an R function to compute the log likelihood for a vector `y` and a scalar `a` using dense matrix methods. The functions `chol` and `backsolve` may be useful. You may also find it useful to use *matrix indexing*; for example for a square matrix `M` the expression

```
M[cbind(2 : nrow(M), 1 : (nrow(M) - 1))]
```

extracts the sub-diagonal of `M`.

(b) Rewrite your function to use sparse matrix methods provided by the `Matrix` package to take advantage of the sparseness of the covariance matrix. The functions `bandSparse`, `solve`, and `chol` may be useful.

(c) Compare the performance of your two functions on data vectors of different lengths. The `system.time` function may be useful. You should see significantly better performance of the sparse matrix approach for a large matrix.

Your submission should include a pdf file with your writeup and text files with `.R` extensions with your code for each problem.

You should submit your assignment electronically using Icon. Submit your work as a single compressed tar file. If your work is in a directory `mywork` then you can create a compressed tar file with the command

```
tar czf mywork.tar.gz mywork
```

Solutions and Comments

1. (a) Here is a simple function for computing the exact regression coefficients that assumes X is a matrix and y a vector:

```
exact.fit <- function(X, y) {
    bY <- as.bigq(y)
    bX <- as.bigq(X)
    bXt <- t(bX)
    as.double(solve(bXt %*% bX, bXt %*% bY))
}
```

(b) A function to compute the regression coefficients by Cholesky factorization, with optional mean centering, is

```
chol.fit <- function(X, y, center = FALSE) {
    if (center) {
        cm <- colMeans(X)
        Xm <- sweep(X, 2, cm)
        beta <- chol.fit(Xm, y - mean(y))
        c(mean(y) - crossprod(cm, beta), beta)
    }
    else {
        R <- chol(t(X) %*% X)
        z <- t(X) %*% as.vector(y)
        as.vector(backsolve(R, forwardsolve(t(R), z)))
    }
}
```

- This uses `forwardsolve` and `backsolve` to solve the equations. Using `solve` is less efficient and less accurate.

(c)

- Looking at the printed version of the `longley` data frame suggests that the original data were specified in decimal form with at most three digits after the decimal point.
- Many decimal fractions are not exactly representable as binary fractions, so the floating point values in the data frame `longley` are *not* exactly equal to the values from the original paper.
- The expressions

```
> X <- cbind(1, as.matrix(longley[, -7]))
> y <- longley[,7]
> XE <- as.bigq(round(1000 * X)) / as.bigq(1000)
> yE <- as.bigq(round(1000 * y)) / as.bigq(1000)
```

produce in the variables XE and yE exact rational representations of the exact decimal data.

- The relative differences between the coefficients for the exact fit to the binary values in the `longley` frame and the fit to these data reflect the effect of decimal to binary rounding in the binary data:

  ```
  > (exact.fit(XE, yE) - exact.fit(X, y)) / exact.fit(XE, yE)
  [1] -7.835386e-16 -2.833260e-14 -8.911123e-15 -8.586763e-16  4.029450e-15
  [6]  6.354484e-14 -6.069607e-16
  ```

- *Printed* versions of the coefficients, using the default settings of the number of digits to print, are almost the same:

  ```
  > exact.fit(XE,yE)
  [1] -3.482259e+03  1.506187e-02 -3.581918e-02 -2.020230e-02 -1.033227e-02
  [6] -5.110411e-02  1.829151e+00
  > chol.fit(X,y)
  [1] -3.482259e+03  1.506187e-02 -3.581918e-02 -2.020230e-02 -1.033227e-02
  [6] -5.110411e-02  1.829151e+00
  ```

  This is not surprising since computations are done in double precision. To see the differences you need to compute errors or relative errors:

  ```
  > exact <- exact.fit(XE, yE)
  > (chol.fit(X,y) - exact) / exact
  [1] -1.022182e-08 -2.124171e-08 -2.732136e-08 -7.397719e-09 -4.537949e-09
  [6]  4.874051e-08 -9.988415e-09
  > (as.vector(lm.fit(X,y)$coef) - exact) / exact
  [1] -1.697667e-15 -3.984991e-14 -2.712081e-15 -8.586763e-16 -3.357875e-15
  [6] -2.783481e-14 -1.699490e-15
  ```

  The errors for `lm.fit` are very close in magnitude to the machine unit. The errors for the Cholesky approach are substantially larger.
  Mean centering improves the Cholesky result but they remain worse than the results obtained using the QR approach.

  ```
  > (chol.fit(X[, -1], y, TRUE) - exact) / exact
  [1] -1.120460e-13 -9.147053e-13 -3.843406e-13 -1.009803e-13 -5.490126e-14
  [6]  1.166482e-12 -1.086460e-13
  ```

- The approaches using Cholesky factorization are less accurate because of the loss in accuracy in forming the cross product matrix. Mean centering improves this, but some accuracy is still lost.

2. A function to create the dense covariance matrix is

   ```
   mkC0 <- function(n, a) {
       m <- diag(rep(1, n))
       m[cbind(2 : n, 1 : (n - 1))] <- a
       m[cbind(1 : (n - 1), 2 : n)] <- a
       m
   }
   ```

   A simple implementation of the log likelihood function that closely follows the mathematical definition is

```
llik0 <- function(y, a) {
    C <- mkC0(length(y), a)
    -0.5 * log(det(C)) - 0.5 * t(y) %*% solve(C) %*% y
}
```

This has several issues:

- It is almost always a bad idea to compute an inverse; it is more work than needed if you only want to solve one system of equations, and it will decrease numerical accuracy. It is usually better to compute and use an appropriate decomposition.
- It is almost always a bad idea to compute a determinant and then take its logarithm.
- If a matrix decomposition is already available then this can be used to compute the log determinant efficiently and accurately.
- This function returns a 1 × 1 matrix rather than a simple vector of length 1.

The most natural decomposition to use for covariance matrices is the Cholesky decomposition. R functions for computing the Cholesky decomposition of a matrix C produce an upper triangular matrix R such that $C = R^T R$. In terms of this R the quadratic form in the log likelihood is

$$y^T C^{-1} y = y^T (R^T R)^{-1} y = y^T R^{-1} R^{-T} y = z^T z$$

where $z = R^{-T} y$, or z solves $R^T z = y$. Furthermore, $\frac{1}{2}\log(\det(C))$ is equal to the sum of the logarithms of the diagonal elements of R. This leads to the definition

```
llikD <- function(y, a) {
    C <- mkC0(length(y), a)
    R <- chol(C)
    z <- forwardsolve(t(R), y)
    - sum(log(diag(R))) -0.5 * sum(z ^ 2)
}
```

A small improvement is to use the optional arguments to `forwardsolve` to avoid the transpose:

```
llikD <- function(y, a) {
    C <- mkC0(length(y), a)
    R <- chol(C)
    z <- forwardsolve(R, y, upper.tri = TRUE, transpose = TRUE)
    - sum(log(diag(R))) -0.5 * sum(z ^ 2)
}
```

Using the Matrix package a sparse representation of the covariance matrix can created by

```
library(Matrix)

mkC <- function(n, a) {
    d0 <- rep(1, n)
    d1 <- rep(a, n - 1)
    bandSparse(n, k = 0 : 1, diag = list(d0, d1), symm = TRUE)
}
```

The Cholesky factorization of the sparse covariance matrix is also sparse, and the sparse matrix method for the `solve` generic function will take advantage of this and compute the solution z to $R^T z = y$ efficiently. This leads to the definition

```
llikS <- function(y, a) {
    C <- mkC(length(y), a)
    R <- chol(C)
    z <- solve(t(R), y)
    - sum(log(diag(R))) -0.5 * sum(z ^ 2)
}
```

I do not see an obvious way to avoid the transpose.

The overhead associated with the sparse matrix support in the Matrix package is significant; as a result the dense matrix approach is faster for data vectors with less than about 150 observations. But the computational complexity of the dense matrix approach is $O(n^3)$ whereas the sparse matrix approach is $O(n)$; so the sparse matrix approach is much faster for larger vectors, and also uses much less memory.

Some notes:

- `sum(x ^ 2)` is simpler than `t(x) %*% x` and produces a scalar rather than a 1×1 matrix.
- It is best to separate installing packages from scripts that might be run many times.
- You should properly present the timing results with supporting tables and graphs and explanatory text.
- `system.time` is not very accurate for computations taking less than a second.
  - You can replicate the computations in a loop to get more accurate results.

  - The `microbenchmark` package provides a more sophisticated structure for timing experiments.

- Make sure your functions take the arguments requested in the problem.
- Make sure you provide your code in a text file with a `.R` extension.
- Don't create a dense matrix and the convert to a sparse one — that takes $O(n^2)$ time and space. Use `bandSparse` to create the sparse matrix directly, which is $O(n)$ is time and space.
- Once you have the Cholesky factorization you do not need the inverse.
- Once you have the Cholesky factorization you can compute the log determinant you need as the sum of the logarithms of the diagonals. Do *not* compute the product of the diagonals and then the logarithm!
- You do *not* want to compute the inverse of the sparse matrix or its Cholesky factor: they are not sparse!
- Using `forwardsolve` with a sparse matrix silently converts the sparse matrix to a dense matrix.
- Speed of likelihood calculations is important since they are needed many times in iterative computations of maximum likelihood estimates.
- Space efficiency is also important if you want to be able to handle large data sets.
- No matter how fast or space-efficient your function is, it is of no use if it gets the wrong answer!