Stat 243
Problem set 7

A. Strand, ID: 540441, GitHub: AndreasStrand

November 12, 2015

Problem 1

1. Chen, Li and Fu want to find a procedure of doing hypothesis testing on the order of a normal mixture model. They invent a new expectation-maximazation (EM) for testing the null hypothesis. The power of the EM tests are used as a measure to assess the method.

2. For the simulation study of the EM test the set B (Beta), number of Iterations K (Kappa) and the penalty functions have to be specified. The significance level, the sample sizes and the number of repetitions are likely to affect the power of the test.

3. It is of course possible to test other choices of m, and not only 2 and 3. This is however something they considered for sure. I would be interesting to see some other choices of B though. In this exampe they chose B=0.1,0.3,0.5.

4. First we would make clear what parameters that has to be decided for an individual experiment and what the statistic is. Here the statistic is the EM, and there is a whole bunch of parameters as mentioned above. The next thing to do is specifying test levels of each parameter. Then we would write a function taking a level of every parameter as input and returning a value of the test statistic. For each input combination we want to replicate the experiment m times. This may be done by parallell processing for big data sets. Furthermore, we gather the data of each process into a data structure and present clearly in either a table or a graphical figure. It is impotant to share code, levels, the seed and other data needed in order to reproduce the data. Other things to consider is reporting uncertainty, write efficient code and have good documentation.

5. For the plot of type 1 errors they did a clever thing by showing the four combinations of n in 200, 400 and significance level in 0.01, 0.05 in a 2x2 grid. That way it easy to compare levels seperately or sample size separetely by staring at the figure. However, I think they could have presented the parameter specifications better. It would be nice to have the specifications of the null models and of the alternative models next to each other. A way of doing that may be listing the parameter sets in one table and use horisontal lines to seperate. Still, it could be messy because of the EM power entries.

6. It makes sense how the power varies with the DGM. The power is without exception equal or increasing as the sample size or repetitions increases. The slope of the change in power varies by model.

7. I looked into replication code at http://sas.uwaterloo.ca/ p4li/software/emnorm.R. I did not find sufficient information about the PRNG in the code there, but I may have overlooked a note somewhere about where to find it. Also the JASA requirements include information about computers and major software components, but I cannot find that anywhere in the publication. It seems like satisfy all the other requirements. That includes information about the general procedure, the numerical agoritms, programming languages, estimated accuracy and clarity in presenting the result.

Problem 2

(a) To derive the operation count I have used information about the Cholesky decomposition at `https://en.wikipedia.org/wiki/Cholesky_decomposition`. Let a matrix A be $n \times n$. The CD can be written as

$$\begin{aligned} A &= \begin{pmatrix} L_{11} & 0 & \cdots & 0 \\ L_{21} & L_{22} & & 0 \\ \vdots & & \ddots & \\ L_{n1} & L_{n2} & & L_{nn} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & \cdots & L_{n1} \\ 0 & L_{22} & & L_{n2} \\ \vdots & & \ddots & \\ 0 & 0 & & L_{nn} \end{pmatrix} \\ &= \begin{pmatrix} L_{11}^2 & L_{21}L_{11} & \cdots & L_{n1}L_{11} \\ L_{21}L_{11} & L_{21}^2 + L_{22}^2 & & L_{n1}L_{21} + L_{n2}L_{22} \\ \vdots & & \ddots & \\ L_{n1}L_{11} & L_{n1}L_{21} + L_{n2}L_{22} & \cdots & L_{n1}^2 + L_{n2}^2 + \cdots + L_{nn}^2 \end{pmatrix} . \end{aligned}$$

When starting at the top left we can move through the matrix and compute the entries by

$$\begin{aligned} L_{jj} &= \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk} L_{jk}^*} \quad \text{and} \\ L_{ij} &= \frac{1}{L_{jj}} \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk}^* \right), \quad i > j \, . \end{aligned} \tag{1}$$

Now let us calculate the total number of multiplications and divisions computations the decomposition will result in for a $n \times n$ matrix. We only have to compute the entries for the lower triangular factor since the other factor is just the transpose, so we get it for free. From 1 we have that the diagonal entries involves $j-1$ multiplications, where j is the column index. The off-diagonal entries requires $j-1$ multiplications and one division. Let $md(x)$ be the number of multiplications and divisions for a method x. Then

$$md(CD) = \sum_{j=1}^{n} j - 1 + \sum_{i=1}^{n} \sum_{j=1}^{i-1} j = \frac{(n-1)n}{2} + \frac{(n-1)n(n+1)}{6} = \frac{n^3}{6} + \frac{n^2}{2} - \frac{2n}{3} \, .$$

(b) You can do the computation of the decomposition in-place by applying for example the Cholesky-Crout algorithm. A good feature about Cholesky is that since $U = L^T$, we do not have to store two matrices separately, we can have them point to the same numeric values, just in different orders.

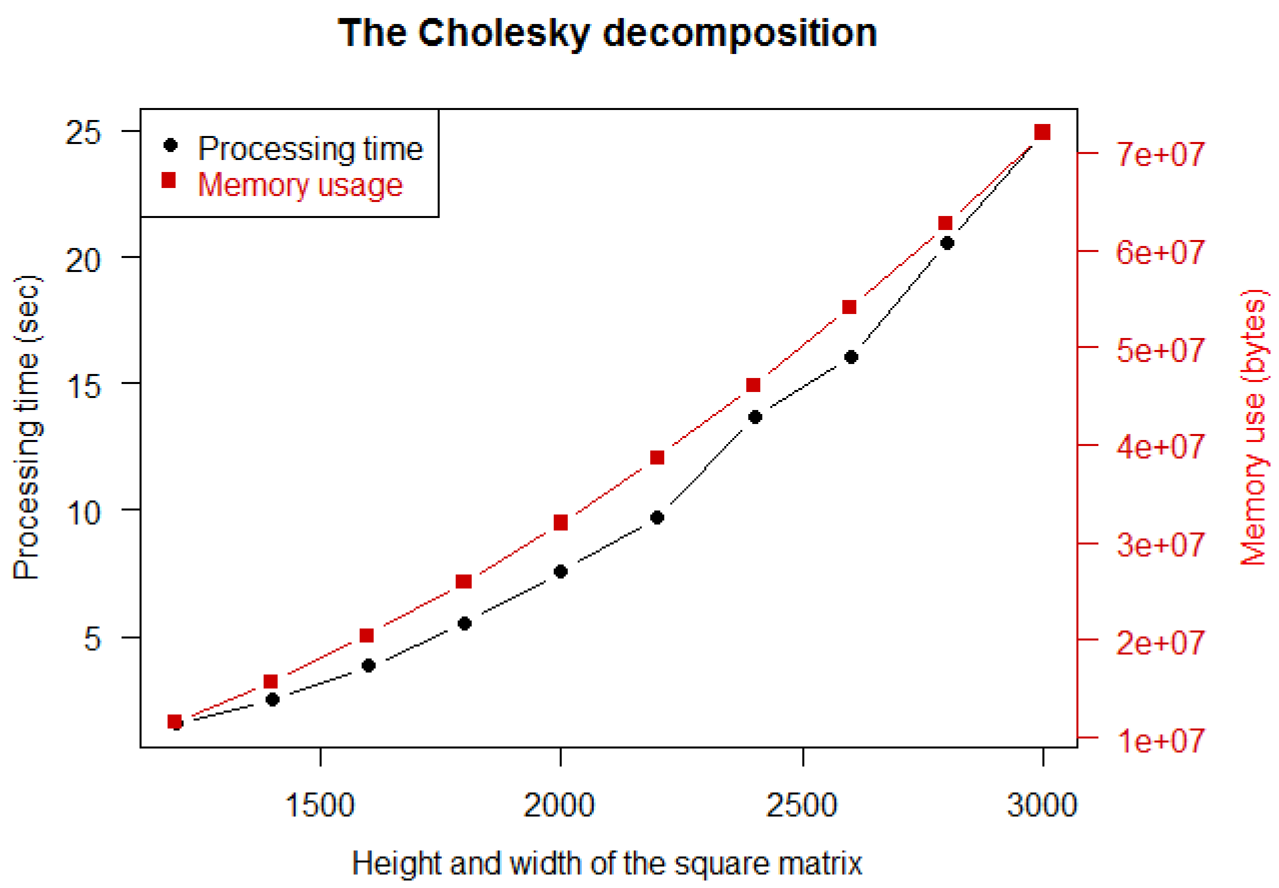

Figure 1: Memory usage of the Cholesky decomposition of matrices as a function n, as well as corresponding processing times.

(c) The code for this part is included as code 1. To check how much space the CD of a matrix takes up in R, I decomposed a 2000×2000 matrix X. The matrix X was of size 32 Mb and CD(X) was also 32 Mb. This is as expected from (b).

The next step was creating random matrices of different sizes to see how memory usage of the CD and processing time relates to the size of the matrix. For every size n of the square matrix 10 replications were done. I extracted the maximum value across the replications for each n. The results are plotted in figure 1. Not surprisingly, it turned out that for a given n the CD is the same across all the replications. The processing time and the memory usage are both increasing with n, and the slope seems to be slightly steeper than linear.

Code 1: Memory usage and processing time of the Cholesky decomposition

```
setwd("C:/Users/Andreas/Documents/stat243")
library(pryr)
set.seed(1337)

## Randomly generating a symmetric positive deminite matrix
spd = function(n=100,maxmin = 100){
  # Create nxn matrix with numeric values in {-maxmin, maxmin}
  X = matrix(runif(n^2, -maxmin, maxmin), ncol=n)
  # Make it symmetric positive definite
  return(X%*%t(X))
}
## Looking at memory usage in the Cholesky decomposition
cholMem = function(msize){
  X = spd(msize)
  memBefore = mem_used()
  C = chol(X)
  memAfter = mem_used()-memBefore
  return(memAfter)
}
cholMem(2000) # (32 Mb)

## Memory usage and processing time by n
nvals = seq(1200,3000,200)
tandm=matrix(rep(0,2*length(nvals)), nrow=2)

# unname(system.time()[3]) returned elapsed time as numeric
tandm = function(){
  return(unlist(sapply(X=nvals, function(X){
  time = unname(system.time(mem <- cholMem(X))[3])
  return(c(time,mem))
  })))
}
r = 10 # number of replications
A = replicate(r, tandm())

# Gathering maxima
memMax = rep(0,length(nvals))
runMax = rep(0,length(nvals))
for (i in 1:length(nvals)){
  runMax[i]=max(A[1,i,])
  memMax[i]=max(A[2,i,])
}

## Plotting the result
# Create some extra space to the right
par(mar= c(5, 4, 4, 6) + 0.1)

# Plot the run times
plot(nvals, runMax, pch=16, axes=FALSE, xlab="", ylab="", new=TRUE,
     type="b",col="black", main="The Cholesky decomposition")
axis(2, ylim=c(0,max(runMax)),col="black",las=1)
mtext("Processing time (sec)",side=2,line=2.5)

# Create a box around the current graphics and allow for more drawing
box()
par(new=TRUE)

# Plot the memory use with axes on the right
plot(nvals, memMax, pch=15,  xlab="", ylab="",
     axes=FALSE, type="b", col="red3")
axis(4, ylim=c(min(memMax),max(memMax)), col="red3",col.axis="red3",las=1)
mtext("Memory use (bytes)",side=4,col="red2",line=4)

# Create the first axis and legens
axis(1,xlim=c(min(nvals),max(nvals)))
```

```
mtext("Height and width of the square matrix",side=1,col="black",line=2.5)
legend("topleft",legend=c("Timing","Memory usage"),
       text.col=c("black","red3"),pch=c(16,15),col=c("black","red3"))
```

Problem 3

(a) The computations for this problem is included in code 2. The order of the timing of the three methods was as expected. The inverse is slowest as always, and it is a lot slower than the other. Cholesky decomposition was slightly faster than the built-in R *solve*() for the 5000×5000 matrices with a column vector. This is not surprising since we do not tell R that the matrices are positive definite when applying *solve*(). When we use the *chol*(), *forwardsolve*() and *backsolve*() we utilize that we now about the structure of the matrices, that it is PD. The timings and discrepancies in the result are presented in table 1.

Table 1: Timing for three methods of solving a linear system of a positive definite matrix. The differences in the solutions are also included. The infinity norm is used.

	Inverse(1)	Solve aug. mat.(2)	Cholesky Dec.(3)
Timing(sec)	116.38	24.37	21.65
	1 vs 2	1 vs 3	2 vs 3
Difference in b	$1.72 \cdot 10^{-13}$	$1.70 \cdot 10^{-6}$	$1.70 \cdot 10^{-6}$

(b) The differences in the computation of b for the three methods are included in table 1. The method using the inverse and the method using *solve*() had a maximum element-wise difference of order 10^{-13}. The CD had a maximum different of order 10^{-6} to both the other methods. Machine precision is of order 10^{-16}. The conclusion is then that the solutions are not very close compared to machine precision and the CD stand out from the other.

Code 2: Comparison of three methods in R for solving linear systems of positive definite matrices.

```
set.seed(314)
# Creating a function that returns the run time of 3 methods
  # of solving a dense linear system
triSolve = function(N){
  # Creating a random NxN matrix and Nx1 vector
  X = spd(N, maxmin = 1000)
  y = matrix(runif(N, -1000, 1000), ncol=1)
    # note: unname(proc.time()[3]) return elapsed time as numeric

  # 1. Inverse
  start = proc.time()
  Xinv = solve(X)
  b1 = Xinv%*%y
  t1 = unname((proc.time()-start)[3])

  # 2. Solve augmented matrix
  t2 = unname(system.time(b2 <- solve(X,y))[3])

  # 3. Cholesky
  start = proc.time()
  C = chol(X)
  d = forwardsolve(t(C),y)
  b3 = backsolve(C,d)
  t3 = unname((proc.time()-start)[3])

  # Calculating the differences in the inf-norm between solutions
```

```
    d1 = max(abs(b2-b1))
    d2 = max(abs(b3-b1))
    d3 = max(abs(b3-b2))

    return(matrix(c(t1,t2,t3,d1,d2,d3),nrow=3))
  }

  triSolve(1000)
  ## Comparing methods of solving linear system
  # Doing r replications and taking extracting max values
  r=10
  B=replicate(r, triSolve(5000)) # Approx half an hour
  # Creating a summary using inf. norm
  timings = rep(0,3)
  diffs = rep(0,3)
  for (i in 1:3){
    timings[i]=max(B[i,1,])
    diffs[i]=max(B[i,2,])
  }
  # Summary
  timings
    # [1] 116.38  24.37  21.65
  diffs
    # [1] 1.723066e-13 1.696592e-06 1.696592e-06
```

Problem 4

We want to avoid doing inverses explicitly when estimating $\hat{\beta} = (X^T\Sigma^{-1}X)^{-1}X^T\Sigma^{-1}Y$. It is also convenient to make use of that Σ is symmetric and in practice positive definite. When solving a linear system of SPD matrices you want to use the Cholesky decomposition. A plan for computing $\hat{\beta}$ can be written in pseudo-code as

1. Find CD(Σ).
   Solve the lower triangular system of the CD with X.
   Solve the upper triangular system to get $\Sigma^{-1}X$.

2. Doing the analogue to 1. for finding $\Sigma^{-1}y$.

3. Matrix multiplication of X^T and $\Sigma^{-1}X$.
   Matrix multiplication of X^T and $\Sigma^{-1}y$.
   Doing Gauss-Jordan to solve $(X^T\Sigma^{-1}X)\beta = X^T\Sigma^{-1}Y$.

The R function for doing the procedure explained above is included as code 3. Input to *gls*() are the design matrix X, the response vector y and the conditional variance of the error term given X, *omega*. The returned value from the function is a vector of coefficients for the generalized least squares.

Code 3: An R function for calculating the general least squares estimator.

```
gls = function(X,y,omega){
  C = chol(omega, pivot=T)
  d = forwardsolve(t(C),X, upper.tri=F)
  SinvX = backsolve(C,d, upper.tri=T)
  d = forwardsolve(t(C),y, upper.tri=F)
  Sinvy = backsolve(C,d, upper.tri=T)
  return(solve(t(X)%*%SinvX,t(X)%*%Sinvy))
}
```

Problem 5

(a) Let us assume X real. The singular value decomposition of X is

$$X = UDV^T ,$$

where U and V are unitary matrices and D is diagonal with the singular values of X. Then we can write the matrix product of the transpose of X with X as

$$\begin{aligned} X^TX &= (UDV^T)^T(UDV^T) = VDU^TUDV^T = VDU^{-1}UDV^T \\ &= VDIDV^T = VD^2V^T = VD^2V^{-1} . \end{aligned}$$

This is just the eigendecomposition of X^TX. Hence the eigenvalues of X^TX is the squares of the singular values of X. The eigenvectors of X^TX is the columns of V, which is the right singular vectors of X.

(b) Let the eigendecomposition be $X = Q\Lambda Q^{-1}$. The *eigenvalue equation* of Z can be written and rewritten as

$$\begin{aligned} Z\vec{v} &= \lambda_z\vec{v} \\ (X + cI)\vec{v} &= \lambda_z\vec{v} \\ X\vec{v} + cI\vec{v} &= \lambda_z\vec{v} \\ X\vec{v} &= \underbrace{(\lambda_z - c)}_{\lambda_x}\vec{v} . \end{aligned}$$

The last equation can be recognized as the eigenvalue equation of X. This equation is true for all the eigenvalues of X and the corresponding eigenvalues of Z. We can then write it in a vectorized fashion as $\vec{\lambda_z} = \vec{\lambda_x} + \vec{c}$, where adding $\vec{c}$ means adding c to every term of $\vec{\lambda_x}$. Since there is n eigenvalues of each matrix, calculating the eigenvalues of Z requires n computations.