Brain data and Music data application.

Chanwoo Lee

Dec 15, 2019

1 Algorithm modification for Brain data.

When I applied our algorithm in the brain data, there are some problems I had to cope with. The first problem was memory issue. My codes kept getting killed by the statistic server saying that it is out of the memory. Main reason for this error is that our algorithm generates really large matrices whose dimension increase to 628864×2000 (the size is almost 10 GB) to update the core tensor. This matrix is made to get a gradient value for each iteration and has the following form.

$$W = A_3 \otimes A_2 \otimes A_1.$$

However, this matrix is needed to find a gradient and Hessian of the loss function. I tried to find a way not using this large matrix to calculate a gradient but there is no way to get gradient without W because gradient of the core tensor has the following elements.

$$\sum_{k\in[K]} \sum_{i\in[d_1 d_2 d_3]} W_i q_{ik}. \tag{1}$$

where W_i is the i-th row of the matrix W and q_{ik} is a constant. Instead, I modified the gradient algorithm not saving whole W.

The first trial is to calculate each row of W and add each above term without calculating whole W and saving it. For this method, I used 'for' loop and made new algorithm to do parallel computing. However, this approach becomes so slow that I cannot implement this algorithm for the brain data size: the server spent 3 days to get an output but failed.

The second approach is from the fact that it doesn't take too much space to save matrices

W_1, W_2, W_3 where

$$W_1 = A_3 \otimes A_2, \quad W_2 = A_3 \otimes A_1, \quad W_3 = A_2 \otimes A_1$$

. To get (1), I saved the matrix W_3 and do parallel computing based on the following formula.

$$\sum_{k \in [K]} \sum_{j \in d_3} \left([A_3]_j \otimes W_3 \right)^T q_{jk}. \tag{2}$$

where $[A_3]_j$ is j-th row of A_3 and q_{jk} is a constant vector. Although this approach is still slower than saving the whole matrix W, running times is improved a lot from the first method and I put this algorithm in the server now. Also, I changed to use 'BFGS' method instead of using Newton's method not to use Hessian matrix. The following is the modified algorithm to get cost function and gradient.

```
########### cost function  with arbitrary k ###########
# previous cost function
h1 = function(A_1,W1,ttnsr,omega){
  k = length(omega)
  thet =W1%*%c(A_1)
  p = matrix(nrow = length(thet),ncol = k)
  for (i in 1:k) {
    p[,i] = as.numeric(logistic(thet + omega[i]))
  }
  p =  cbind(p,rep(1,length(thet)))-cbind(rep(0,length(thet)),p)
  l = lapply(1:(k+1),function(i) -log(p[which(c(ttnsr)==i),i]))
  return(sum(unlist(l)))
}

# current cost function not saving W
hc = function(A_1,A_2,A_3,C,ttnsr,omega){
  k = length(omega)
  thet = c(ttl(C,list(A_1,A_2,A_3),ms=1:3)@data)
  p = matrix(nrow = length(thet),ncol = k)
  for (i in 1:k) {
    p[,i] = as.numeric(logistic(thet + omega[i]))
  }
```

```
  p =  cbind(p,rep(1,length(thet)))-cbind(rep(0,length(thet)),p)
  l = lapply(1:(k+1),function(i) -log(p[which(c(ttnsr)==i),i]))
  return(sum(unlist(l)))
}


########### gradient with arbitrary k ###########
# previous gradient function
g1 = function(A_1,W,ttnsr,omega){
  k = length(omega)
  thet =W%*%c(A_1)
  p = matrix(nrow = length(thet),ncol = k)
  for (i in 1:k) {
    p[,i] = as.numeric(logistic(thet + omega[i]))
  }
  q = matrix(nrow = length(thet),ncol = k+1)
  q[,1] <- p[,1]-1
  for (i in 2:k) {
    q[,i] <-  (p[,i]*(1-p[,i])-p[,i-1]*(1-p[,i-1]))/(p[,i-1]-p[,i])
  }
  q[,k+1] <- p[,k]
  l <- Reduce("+",lapply(1:(k+1),function(i) apply(rbind(W[which(c(ttnsr)
   ==i),])*q[which(c(ttnsr)==i),i],2,sum)))
  return(l)
}

# current gradient function not saving W
gc = function(A_1,A_2,A_3,C,ttnsr,omega){
  k = length(omega)
  thet = c(ttl(C,list(A_1,A_2,A_3),ms=1:3)@data)
  p = matrix(nrow = length(thet),ncol = k)
  for (i in 1:k) {
    p[,i] = as.numeric(logistic(thet + omega[i]))
  }
  q = matrix(nrow = length(thet),ncol = k+1)
  q[,1] <- p[,1]-1
  for (i in 2:k) {
    q[,i] <-  (p[,i]*(1-p[,i])-p[,i-1]*(1-p[,i-1]))/(p[,i-1]-p[,i])
  }
```

```
  q[,k+1] <- p[,k]
  W = kronecker(A_2,A_1)
  d = c(nrow(A_1),nrow(A_2),nrow(A_3))
  cl <- makeCluster(20)
  registerDoParallel(cl)
  l <- foreach(j = 1:d[3],.combine = "+") %dopar% {
    Reduce("+",lapply(1:(k+1),function(i) apply(rbind(kronecker(A_3[j,,
   drop = F],W)[which(c(ttnsr)[(d[1]*d[2]*(j-1)+1):(d[1]*d[2]*(j-1)+d[1]*d
   [2])]==i),])*
                                                q[which(c(ttnsr)[(d[1]*d
   [2]*(j-1)+1):(d[1]*d[2]*(j-1)+d[1]*d[2])]==i)+d[1]*d[2]*(j-1),i],2,sum)
   ))
  }
  stopCluster(cl)
  return(l)
}
```

2 Current result for the brain data.

For small rank, saving whole W matrix and calculations based on it require almost 20 GB memory which is doable in the server. So I got the result from the rank (5,5,5) to the rank (5,5,20) where the first rank is fixed as 5. Also, I obtained the result from the rank (6,6,4) to (21,21,4) where the second rank is fixed as 4. The result is in the following figures. Based on this figure, my guess for the best rank according to BIC would be around (23,23,6) because in the second plot we can expect that decreasing rate becomes almost zero around the first rank being 21. Also, from the first plot, we can expect that the best second rank would not be large.

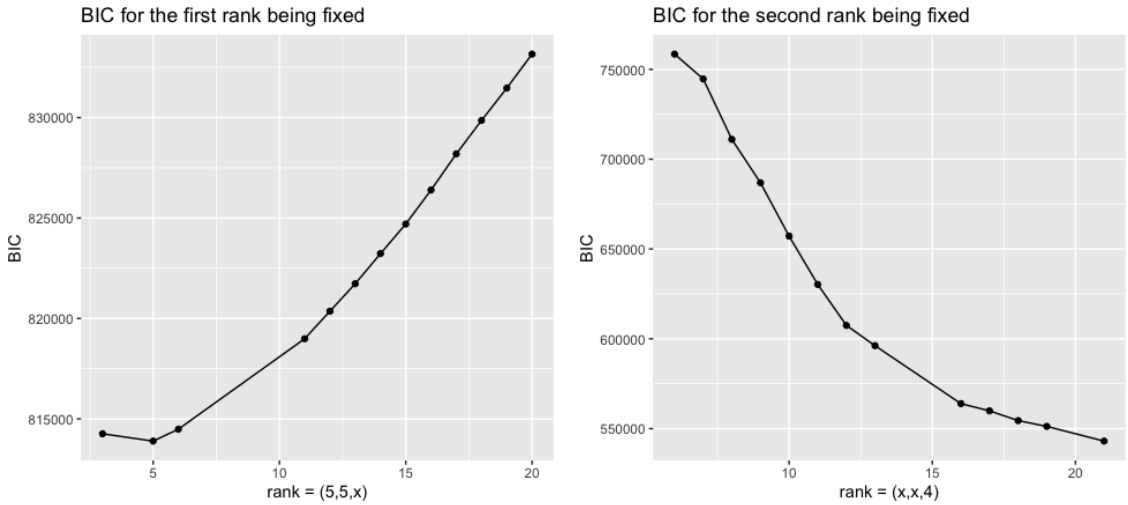

Figure 1: BIC result for one rank being fixed.

3 Questions and discussions for the Brain data application

1. Expected rank size issue: Based on the outputs I get from the server, my guess of the rank is around (23,23,6). However, if you consider the dimension of the brain data (68,68,136), I do not know how to interpret this rank size based on the meaning of each mode.

2. Symmetry issue: From the data output with the rank (21,21,4), I realized A_1 and A_2 are not necessarily the same. The main reason for this is that A_1 and A_2 are not identifiable which means I can manipulate A_1 and A_2 without chaanging $\Theta = \mathcal{C} \times_1 A_1 \times_2 A_2 \times_3 A_3$. However, if I check $\Theta_{\cdot i \cdot} - \Theta_{i \cdot \cdot}$ where $\Theta_{\cdot i \cdot}$ is a slice with the second mode being fixed as i, the maximum entry for all $i \in [68]$ is 0.91. Considering that the maximum entry of Θ is 145, We can say that Θ is almost symmetric. The proportion between the two is 0.006239625

3. Clustering issue: Based on the following formula, if a row of $(A_1)_{i_1 \cdot}$ and a row of $(A_1)_{i_2 \cdot}$ is close enough, $\theta_{i_1 jk}$ and $\theta_{i_2 jk}$ have close value. However, it is not necessary condition and since the range of C is much larger than the range of A_1, we need to consider the

value of C_{i_1jk} and C_{i_2jk}. So I am thinking about do clustering with weight $C_{\cdot jk}$

$$\theta_{ijk} = \sum_{p=1}^{68}\sum_{q=1}^{68}\sum_{r=1}^{136} \mathcal{C}_{pqr}(A_1)_{ip}(A_2)_{jq}(A_3)_{kr}$$

4 Missing data handling

4.1 Algorithmic problem

If I do not set threshold α when implementing the algorithm with the music survey data, The error pops up saying that the parameter for estimation goes infinity. If you consider simple case estimation with missing data, you can figure out why this error happens. Let us consider the following model.

$$P(y_{ijk} = 1) = \text{logit}(a_i b_j c_k) \text{ where } i, j, k \in \{1, 2\} \tag{3}$$

Suppose we have data such that y_{1jk} have only 2 data available and the values are 1 while y_{2jk} have all data available. Since a_1 is only component for y_{1jk} and do not affect any values in y_{2jk}, the MLE for a_1 should go to infinity. This situation happens in the Music data. Therefore, we have to set the threshold α to make the algorithm run without errors.

4.2 Method of the estimation

My method to infer the missing data us to input missing data with EM. This approach consists of mainly two steps.

1. In expectation step, we compute expectation over missing values using our algorithm with threshold α from initial music data.

2. In maximization step, we update our data set replacing missing values with the values we estimated in expectation step. Based on this new data set, we obtain optimal parameters that minimize the loss function with the new data set.

3. Repeat expectation steps and maximization steps until converge.

The algorithm for the misssing data is listed below.

```
source("functions2.R")
load("InCar_Music.RData")

fit_missing = function(tensor,ttnsr,C,A_1,A_2,A_3,omega=TRUE,alph){
  alphbound <- alph+10^-4
  result = list()
  error<- 3
  iter = 0
  cost=NULL
  omg = omega
  while ((error > 10^-4)&(iter<50) ) {
    iter = iter +1

    #update omega
    prevtheta <- ttl(C,list(A_1,A_2,A_3),ms=1:3)@data
    if(sum(omg)==TRUE) {
      omega <- polr(as.factor(c(ttnsr[ttnsr>0]))~offset(-c(prevtheta[ttnsr
   >0])))$zeta
    }


    prev <- likelihood(ttnsr[tensor>0],prevtheta[tensor>0],omega)


    #update A_1
    W =kronecker(A_3,A_2)%*%t(k_unfold(C,1)@data)
    A_1 <- comb(A_1,W,ttnsr,1,omega,alphbound)
    #orthognalize A_1
    qr_res=qr(A_1)
    A_1=qr.Q(qr_res)
    C=ttm(C,qr.R(qr_res),1)
    new <- likelihood(ttnsr[tensor>0],theta[tensor>0],omega)
    if(max(abs(ttl(C,list(A_1,A_2,A_3),ms=1:3)@data))>=alph) break

    # update A_2
    W <- kronecker(A_3,A_1)%*%t(k_unfold(C,2)@data)
    A_2 <- comb(A_2,W,ttnsr,2,omega,alphbound)
    #orthognalize A_2
    qr_res=qr(A_2)
```

```
    A_2=qr.Q(qr_res)
    C=ttm(C,qr.R(qr_res),2)
    new <- likelihood(ttnsr[tensor>0],theta[tensor>0],omega)
    if(max(abs(ttl(C,list(A_1,A_2,A_3),ms=1:3)@data))>=alph) break


    # update A_3
    W <- kronecker(A_2,A_1)%*%t(k_unfold(C,3)@data)
    A_3 <- comb(A_3,W,ttnsr,3,omega,alphbound)
    #orthognalize A_3
    qr_res=qr(A_3)
    A_3=qr.Q(qr_res)
    C=ttm(C,qr.R(qr_res),3)
    if(max(abs(ttl(C,list(A_1,A_2,A_3),ms=1:3)@data))>=alph) break

    # update C
    W <- kronecker(kronecker(A_3,A_2),A_1)
    C <- corecomb(C,W,c(ttnsr),omega)
    new <- likelihood(ttnsr[tensor>0],theta[tensor>0],omega)

    theta <- ttl(C,list(A_1,A_2,A_3),ms=1:3)@data
    new <- likelihood(ttnsr[tensor>0],theta[tensor>0],omega)
    cost = c(cost,new)
    error <- abs((new-prev)/prev)
    if(max(abs(ttl(C,list(A_1,A_2,A_3),ms=1:3)@data))>=alph) break
  }
  result$C <- C; result$A_1 <- A_1; result$A_2 <- A_2; result$A_3 <- A_3
  result$iteration <- iter
  result$cost = cost; result$omega=omega; result$lastcost = new
  return(result)
}



fit_ordinal_missing = function(tensor,C,A_1,A_2,A_3,omega = TRUE,alph =
    20){

  ttnsr = tensor
```

```
  output = list()
  error<- 3
  iter = 0
  cost=NULL
  # Maximization step
  result = fit_missing(tensor,ttnsr,C,A_1,A_2,A_3,omega=TRUE,alph =20)
  pcost = result$lastcost
  theta = ttl(result$C,list(result$A_1,result$A_2,result$A_3),ms=1:3)@data
  # Expectation step
  ttnsr[which(tensor==-1)] = realization(theta,result$omega)@data[which(
   tensor==-1)]
  while ((error > 10^-4)&(iter<50)) {
    # Maximization step
    result = fit_missing(tensor,ttnsr,C,A_1,A_2,A_3,omega=TRUE,alph =20)
    ncost = result$lastcost
    theta = ttl(result$C,list(result$A_1,result$A_2,result$A_3),ms=1:3)
   @data
    # Expectation step

    ttnsr[which(tensor==-1)] = realization(theta,result$omega)@data[which(
   tensor==-1)]
    error = abs((ncost-pcost)/ncost)
    cost = c(cost,ncost)
    pcost = ncost
    iter = iter+1

  }
  output$C = result$C; output$A_1 = result$A_1; output$A_2 = result$A_2;
   output$A_3 = result$A_3
  output$iteration = iter; output$error= error; output$costvar = cost
  return(output)
}



#Initialization
A_1 = randortho(d[1])[,1:r[1]]
A_2 = randortho(d[2])[,1:r[2]]
```

```
A_3 = randortho(d[3])[,1:r[3]]
C = rand_tensor(modes = r)
result = fit_ordinal_missing(tensor,C,A_1,A_2,A_3,omega=TRUE,alph =20)
```