Brain data and Music data application 3

Chanwoo Lee

Dec 25, 2019

1 Missing data handling

In the new algorithm, we define $Q(\Theta, \Theta_0) = \frac{1}{4}\|(\Theta - (\Theta_0 - 2\text{Gradient}(\Theta_0))\|_F^2$ and updated Θ as a $\hat{\Theta}$ such that

$$\hat{\Theta} = \underset{\Theta:\text{rank}(\Theta)=r}{\arg\min} \ |(\Theta - (\Theta_0 - 2\text{Gradient}(\Theta_0))\|_F^2.$$

When Θ is complete, then we can use Tucker decomposition. In particular,

$$\hat{\Theta} = \text{Tucker}(\Theta_0 - 2\text{Gradient}(\Theta_0)).$$

However, when Θ is not complete, we have to find minimizer for

$$\sum_{(i,j,k)\in\Omega} \frac{1}{4}(\theta_{ijk} - \big((\theta_0)_{ijk} - 2\text{Gradient}((\theta_0)_{ijk})\big))^2.$$

where Ω is a index set of available data. In this case we can rephrase the above term by defining $w_{ijk} = \begin{cases} 1, & \text{when } (i,j,k) \in \Omega \\ 0, & \text{when } (i,j,k) \in \Omega^c \end{cases}$.
Then the minimizer Θ can be obtained by

$$\hat{\Theta} = \underset{\Theta:\text{rank}(\Theta)=r}{\arg\min} \frac{1}{2}\|(W * \big(\Theta - (\Theta_0 - 2\text{Gradient}(\Theta_0))\big)\|_F^2.$$

We can find the minimizer numerically using gradients. Let us denote $\Theta = \mathcal{C}\times_1 A_1 \times_2 A_2 \times_3 A_3$ and $f_W(\Theta) \overset{\text{def}}{=} \frac{1}{2}\|(W * \big(\Theta - (\Theta_0 - 2\text{Gradient}(\Theta_0))\big)\|_F^2 = \frac{1}{2}\|(W * \big(\Theta - M)\|_F^2$. Then gradients for each factor are as follows.

$$\nabla_{A_1} f_W = [W * (\mathcal{C} \times_1 A_1 \times_2 A_2 \times_3 A_3 - M)]_{(1)}[\mathcal{C} \times_2 A_2 \times_3 A_3]_{(1)}^T. \tag{1}$$

$$\nabla_{\mathcal{C}} f_W = W * (\mathcal{C} \times_1 A_1 \times_2 A_2 \times_3 A_3 - M) \times_1 A_1^T \times_2 A_2^T \times_3 A_3^T. \tag{2}$$

Based on those gradients, we can minimize f_W with block optimization, where in f_W is optimized with respect to one set of parameters with other parameters being fixed. However, in this case, the computation cost is expensive considering that updating Θ here is a just one iteration out of whole iterations. In particular, our previous algorithm uses block optimization as a main iteration. However, this sub iteration algorithm can be used as a Tucker decomposition for tensors with missing values.

2 Continuous Tucker algorithm with the missing values

I constructed continuous Tucker algorithm for the missing values. This algorithm is a gradient based optimization using the gradients (1) and (2) with M replaced by D which is the data tensor. It took about 30 mins for this algorithm to converge in the application for the brain data with the rank (23,23,8). The algorithm code is as follows.

```
library(MASS)
library(rTensor)
library(pracma)


# initial point
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)
prevtheta <- ttl(C,list(A_1,A_2,A_3),ms=1:3)@data
theta=prevtheta-2*gradient_tensor(A_1,A_2,A_3,C,ttnsr,omega)

# constant term for the missing values
W = array(0, dim(tensor))
W[tensor>0] = 1

costf = function(C,A_1,A_2,A_3,theta,W){
return(sum((W*(ttl(C,list(A_1,A_2,A_3),ms = 1:3)@data-theta))^2)/2)
}

cgm = function(C,A_1,A_2,A_3,theta,W,i){
val = as.tensor(W*(ttl(C,list(A_1,A_2,A_3),ms = 1:3)@data-theta))
```

```
if(i == 1){
g = k_unfold(val,1)@data%*%t(k_unfold(ttl(C,list(A_2,A_3),ms = c(2,3)),1)
    @data)
}else if(i == 2){
g = k_unfold(val,2)@data%*%t(k_unfold(ttl(C,list(A_1,A_3),ms = c(1,3)),2)
    @data)
}else if(i == 3){
g = k_unfold(val,3)@data%*%t(k_unfold(ttl(C,list(A_1,A_2),ms = c(1,2)),3)
    @data)
}else{
g = ttl(val,list(t(A_1),t(A_2),t(A_3)),ms = 1:3)
}
return(g)
}

cgma = function(A,G,theta,W,i){
g = (k_unfold(as.tensor(W),i)@data*(A%*%G-k_unfold(as.tensor(theta),i)
    @data))%*%t(G)
return(g)
}


tucker_missing = function(A_1,A_2,A_3,C,theta,W,alph = TRUE){
alphbound <- alph+10^-4
result = list()
error<- 3
iter = 0
cost=NULL
if (alph == TRUE) {
while ((error > 10^-4)&(iter<50) ) {
  iter = iter +1
  (prev = costf(C,A_1,A_2,A_3,theta,W))

  # update A_1
  # tic()
  # l <- lapply(1:nrow(A_1),
  #             function(i){optim(A_1[i,],
  #                               function(x) costf(C,matrix(x,ncol =
   length(x)),A_2,A_3,theta[i,,,drop = F],W[i,,,drop= F]),
```

```
#                                 function(x) cgm(C,matrix(x,ncol = length
 (x)),A_2,A_3,theta[i,,,drop = F],W[i,,,drop = F],1),
#                                 method = "BFGS")$par})
# #A_1 = matrix(unlist(l),nrow = nrow(A_1),byrow = T) #It is slower
# toc()
#
# tic()
# G = k_unfold(C,1)@data%*%t(kronecker(A_3,A_2))
# f = function(x) return(costf(C,matrix(x,nrow = dim(A_1)[1]),A_2,A_3,
 theta,W))
# g = function(x) return(cgma(matrix(x,nrow = dim(A_1)[1]),G,theta,W,1))
# A_1 <- matrix(optim(c(A_1),f,g,method ="BFGS" )$par,nrow = dim(A_1)
 [1])

# l <- lapply(1:nrow(A_1),
#             function(i){optim(A_1[i,],
#                                 function(x) costf(C,matrix(x,ncol =
 length(x)),A_2,A_3,theta[i,,,drop = F],W[i,,,drop= F]),
#                                 function(x) cgma(x,G,theta[i,,,drop = F
 ],W[i,,,drop = F],1),
#                                 method = "BFGS")$par}) # It is slower
# toc()
#
f = function(x) return(costf(C,matrix(x,nrow = dim(A_1)[1]),A_2,A_3,
 theta,W))
g = function(x) return(cgm(C,matrix(x,nrow = dim(A_1)[1]),A_2,A_3,theta,
 W,1))
A_1 <- matrix(optim(c(A_1),f,g,method ="BFGS" )$par,nrow = dim(A_1)[1])
#orthognalize A_1
qr_res=qr(A_1)
A_1=qr.Q(qr_res)
C=ttm(C,qr.R(qr_res),1)

# update A_2

f = function(x) return(costf(C,A_1,matrix(x,nrow = dim(A_2)[1]),A_3,
 theta,W))
g = function(x) return(cgm(C,A_1,matrix(x,nrow = dim(A_2)[1]),A_3,theta,
 W,2))
```

```
  A_2 <- matrix(optim(c(A_2),f,g,method = "BFGS")$par,nrow = dim(A_2)[1])

  #orthognalize A_2
  qr_res=qr(A_2)
  A_2=qr.Q(qr_res)
  C=ttm(C,qr.R(qr_res),2)

  # update A_3
  f = function(x) return(costf(C,A_1,A_2,matrix(x,nrow = dim(A_3)[1]),
   theta,W))
  g = function(x) return(cgm(C,A_1,A_2,matrix(x,nrow = dim(A_3)[1]),theta,
   W,3))
  A_3 <- matrix(optim(c(A_3),f,g,method = "BFGS")$par,nrow = dim(A_3)[1])
  #orthognalize A_3
  qr_res=qr(A_3)
  A_3=qr.Q(qr_res)
  C=ttm(C,qr.R(qr_res),3)

  # update C
  f = function(x) return(costf(new("Tensor",C@num_modes,C@modes,x),A_1,A_
   2,A_3,theta,W))
  g = function(x) return(c(cgm(new("Tensor",C@num_modes,C@modes,x),A_1,A_
   2,A_3,theta,W,4)@data))
  C <- new("Tensor",C@num_modes,C@modes,data =optim(c(C@data),f,g,method =
    "BFGS")$par)
  (new = costf(C,A_1,A_2,A_3,theta,W))
  cost = c(cost,new)
  (error <- abs((new-prev)/prev))
}
}else{
while ((error > 10^-4)&(iter<50) ) {
  iter = iter +1
  (prev = costf(C,A_1,A_2,A_3,theta,W))
  #update A_1
  G = k_unfold(C,1)@data%*%t(kronecker(A_3,A_2))
  l <- lapply(1:nrow(A_1),
              function(i){constrOptim(A_1[i,],
                                      function(x) costf(C,matrix(x,ncol =
   length(x)),A_2,A_3,theta[i,,,drop = F],W[i,,,drop= F]),
```

```
                                        function(x) cgma(x,G,theta[i,,,drop
  = F],W[i,,,drop = F],1),
                                        ui = as.matrix(rbind(t(G),-t(G))),ci
   = rep(-alph,2*nrow(t(G))),
                                        method = "BFGS")$par})
A_1 = matrix(unlist(l),nrow = nrow(A_1),byrow = T)

#orthognalize A_1
qr_res=qr(A_1)
A_1=qr.Q(qr_res)
C=ttm(C,qr.R(qr_res),1)
if(max(round(abs(ttl(C,list(A_1,A_2,A_3),ms=1:3)@data)),digits = 3)>=
  alph) break

# update A_2
G = k_unfold(C,2)@data%*%t(kronecker(A_3,A_1))
l <- lapply(1:nrow(A_2),
            function(i){constrOptim(A_2[i,],
                                        function(x) costf(C,A_1,matrix(x,
  ncol =length(x)),A_3,theta[,i,,drop = F],W[,i,,drop= F]),
                                        function(x) cgma(x,G,theta[,i,,drop
  = F],W[,i,,drop = F],2),
                                        ui = as.matrix(rbind(t(G),-t(G))),ci
   = rep(-alph,2*nrow(t(G))),
                                        method = "BFGS")$par})
A_2 = matrix(unlist(l),nrow = nrow(A_2),byrow = T)
#orthognalize A_2
qr_res=qr(A_2)
A_2=qr.Q(qr_res)
C=ttm(C,qr.R(qr_res),2)
if(max(round(abs(ttl(C,list(A_1,A_2,A_3),ms=1:3)@data)),digits = 3)>=
  alph) break

# update A_3
G = k_unfold(C,3)@data%*%t(kronecker(A_2,A_1))
l <- lapply(1:nrow(A_3),
            function(i){constrOptim(A_3[i,],
                                        function(x) costf(C,A_1,A_2,matrix(x
  ,ncol =length(x)),theta[,,i,drop = F],W[,,i,drop= F]),
```

```
                                        function(x) cgma(x,G,theta[,,i,drop
    = F],W[,,i,drop = F],3),
                                        ui = as.matrix(rbind(t(G),-t(G))),ci
     = rep(-alph,2*nrow(t(G))),
                                        method = "BFGS")$par})
  A_3 = matrix(unlist(l),nrow = nrow(A_3),byrow = T)
  #orthognalize A_3
  qr_res=qr(A_3)
  A_3=qr.Q(qr_res)
  C=ttm(C,qr.R(qr_res),3)
  if(max(round(abs(ttl(C,list(A_1,A_2,A_3),ms=1:3)@data)),digits = 3)>=
    alph) break

  # update C
  f = function(x) return(costf(new("Tensor",C@num_modes,C@modes,x),A_1,A_
    2,A_3,theta,W))
  g = function(x) return(c(cgm(new("Tensor",C@num_modes,C@modes,x),A_1,A_
    2,A_3,theta,W,4)@data))
  C <- new("Tensor",C@num_modes,C@modes,data =optim(c(C@data),f,g,method =
     "BFGS")$par)
  (new = costf(C,A_1,A_2,A_3,theta,W))
  cost = c(cost,new)
  (error <- abs((new-prev)/prev))
}
}
result$C <- C; result$A_1 <- A_1; result$A_2 <- A_2; result$A_3 <- A_3
result$iteration <- iter
result$cost = cost
return(result)
}
```

3 Current algorithm modification

Our main problem for the algorithm was the memory and time cost for getting gradients for the core tensor. The reason for the expensive cost is that the algorithm uses kronecker product to get the gradients. Instead, the modified algorithm uses Tensor n-mode product

to get a gradient which is a tensor structure. Previous gradient has the following formula.

$$\text{Gradient} = q(A_3 \otimes A_2 \otimes A_1) \text{ where } q \in \mathbb{R}^{d_1 d_2 d_3}.$$

This gradient can have different formula if I made q as a tensor. The modified gradient has the following formula.

$$\text{Gradient} = \text{Tensor}(q) \times_1 A_1^T \times_2 A_2^T \times_3 A_3^T \text{ where } \text{Tensor}(q) \in \mathbb{R}^{d_1 \times d_2 \times d_3}.$$

From this formula, our algorithm does not need to compute kronecker product and takes less memory and time in the end.

The following is the modified gradient algorithm.

```
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[,i] <- p[,i]+p[,i-1]-1
}
q[,k+1] <- p[,k]
g = ttnsr
for(i in 1:(k+1)){
g[which(ttnsr==i)] = q[which(ttnsr==i),i]
}

g = ttl(as.tensor(g),list(t(A_1),t(A_2),t(A_3)),ms = 1:3)

# 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(g)
}
```

4 BIC result from the new algorithm and modified algorithm

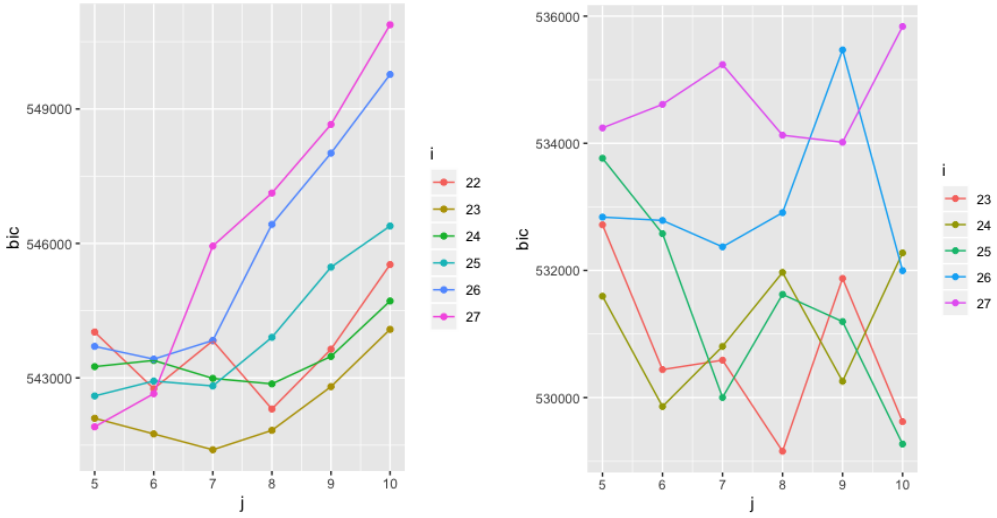

Figure 1: the left figure is the BIC from the new algorithm and the right figure is the BIC from the modified figure. Both figure show the rank around (23,23,8) is good.

5 Cross Validation result

I did 5-fold cross validation to compare performances of the two methods: Continuous Tucker method and Ordinal GLM Tucker Method. The following is the result for Continuous Tucker model and ordinal GLM Tucker model with the rank (25,25,7).

MSE	MAE	Error_rate
0.1593824	0.1591757	0.1590723
0.1575139	0.1574344	0.1573947
0.1587304	0.1586668	0.1586350
0.1576093	0.1575298	0.1574901
0.1592472	0.1591041	0.1590325

Table 1: Continuous Tucker decomposition

MSE	MAE	Error_rate
0.1504456	0.1503343	0.1502787
0.1495392	0.1493961	0.1493246
0.1491338	0.1490383	0.1489906
0.1507557	0.1506444	0.1505888
0.1510102	0.1509306	0.1508909

Table 2: Ordinal GLM Tucker decomposition

6 Clustering

I used loading based clustering for the output with the rank (24,24,8). Also, I used 'tucker' function to make the output normalized before doing clustering. The largest difference between entries of before and after 'tucker' decomposition is 1.477929e-12, which means θ values are almost the same. I choose the number of cluster groups 8 because it does not have the tendency that most of points are included in the same one group as you check in the following figure. I used 'kmeans' function with **nstart** = 20 option to reproduce the same result.

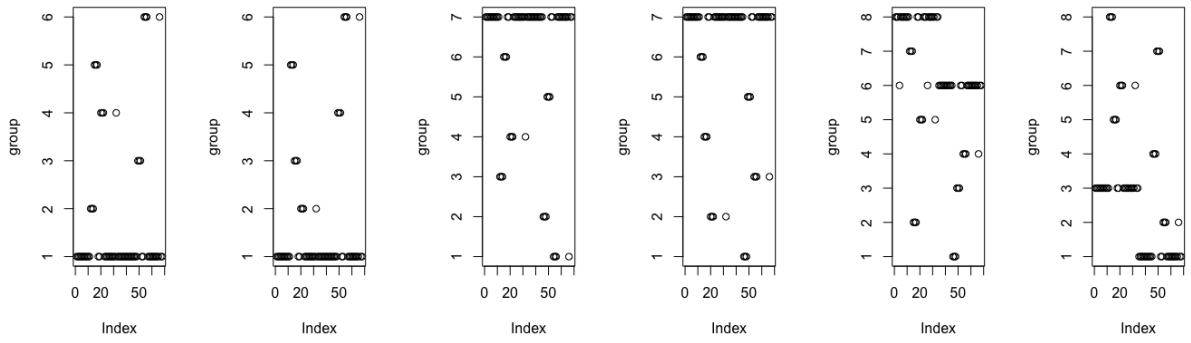

Figure 2: The number of the groups is from 6 to 8 in order. You can see that cluster with mode 1 and with mode 2 have exactly the same clusters.

7 Music data application

There are some changes I made in the algorithm for the music data.

1. I made algorithm that change the order of updating factor matrices at every iteration. The main reason for this is that it usually hits the boundary α in one factor update.

2. The algorithm estimates the missing value as a mean value of realization in expectation step.

However, it sometimes gives me error saying that initial value hits the boundary. This is because after each iteration, output value becomes really close to the boundary and 'constrOptim' function is quite sensitive to this initial value. So I am working on fixing the bug now. One thing I want to discuss is that continuous Tucker method works algorithmically without any modification. One reason for this is that the parameters of continuous Tucker method cannot be infinity while the parameters of ordinal GLM method can be infinity because it only becomes probability 1 in realization. Therefore, if we use EM algorithm for the missing data in ordinal GLM method, it is not fair to compare with continuous Tucker method because we do one extra step more. My suggestion for this problem is that we set α, make all factor matrices and core tensor updated at least one time and estimate the missing value as the mean value from the given parameters.