Simulation of Proposed Method

Introduction

In this note, we give a result of simulation using our proposed method, which uses spike and slab prior in the SSSL and Laplace approximation to compute posterior. Since it is impossible to search over all possible models in practically, we used Metropolis-Hastings algorithm to obtain MCMC smaple of edge inclusion vector. This algorithm was proposed by Chang Liu and Ryan Martin(2019).

In this algorithm, Liu and Martin considered a symmetric proposal distribution $q(G|G')$, where G, G' are two graphs, which samples $G' = (V, E')$ uniformly from the graphs that differ from $G = (V, E)$ in one position. To be more specific, if we use binary indicators γ_{ij} and γ'_{ij} to represent presence/absence of the edge between vertices i and j in G and G' respectively, in each iteration, there is an equal probability to either set $(i, j) = 1$ for one pair of (i, j) uniformly sampled from $V\ddot{O}V \setminus E$, i.e., $\gamma(i, j) = 0$, or set $\gamma'(i, j) = 0$ for one pair of (i, j) uniformly sampled from E, i.e., $\gamma(i, j) = 1$. Using this proposal distribution, we propose following Metropolis-Hastings algorithm to sample the model with the data $X = (X_1, \ldots, X_n)$ given.

1. Initial covariance: $\Sigma^{(0)}$ and corresponding edge inclusion vector: $\Gamma^{(0)}$.
2. for i $= 1, 2, \ldots, k$
3. sample Γ^{cand} using the described symmetric proposal distribution with $\Gamma^{(i-1)}$ given.
4. calculate $\hat{\Sigma}^{(\text{cand})}$ using block coordinate algorithm under the model Γ^{cand}.
5. $\alpha_i = \min\{1, \frac{p^*(\Gamma^{\text{cand}}|X)}{p^*(\Gamma^{(i-1)}|X)}\}$: $p^*(\cdot|X)$ denotes the proposed posterior.
6. $U_i \sim \text{Unif}(0, 1)$.
7. if $U_i \leq \alpha_i$: $\Gamma^{(i)} = \Gamma^{\text{cand}|X}$.
8. else $\Gamma^{(i)} = \Gamma^{(i-1)}$.

Using this algorithm, we give a result of simulation using the covariance matrix that mimics the currency exchange rate in Wang(2015). Our competing methods in this simulation are covariance graphical lasso, which was proposed by Bien and Tibshirani(2010), and sample covariance. The measures of performance are rmse($||\hat{\Sigma} - \Sigma||_F/p$), mnorm(maximum norm), spectral norm, and specificity(sp), sensitivity(se), Matthew Corrleation Coefficient(MCC).

```
#construction of our true covariance matrix
curr_ex=matrix(rep(0,12^2),nrow=12)
diag(curr_ex)=c(0.239,1.554,0.362,0.199,0.349,0.295,0.715,0.164,0.518,0.379,0.159,0.207)
curr_ex[1,2]=0.117; curr_ex[1,8]=0.031; curr_ex[3,4]=0.002; curr_ex[4,5]=0.094
curr_ex[5,12]=-0.036; curr_ex[6,7]=-0.229; curr_ex[6,8]=0.002; curr_ex[8,9]=0.112
curr_ex[8,10]=-0.028; curr_ex[8,11]=-0.008; curr_ex[9,10]=-0.193; curr_ex[9,11]=-0.090
curr_ex[10,11]=0.167

curr_ex=curr_ex+t(curr_ex)-diag(diag(curr_ex))
true_edge=edge_ind(curr_ex,0.1^3)
```

Simulation

The given code calculate the performance for each measure.

```
library(dplyr)
library(mvtnorm)
source("mcmc_slabspike.R")

#median probability model
mpp=function(mat){
  r=nrow(mat); c=ncol(mat)
  result=numeric(length=c)
  for(i in 1:c){
    result[i]=as.numeric(mean(mat[,i])>=0.5)
  }
  return(result)
}

#maximum a probability
mmp=function(mat){
  temp=as.data.frame(mat)
  temp=temp%>%group_by_all%>%count
  l=ncol(temp)

  n=temp$n
  ind=which(n==max(n))
  model=as.numeric(temp[ind,1:(l-1)])
  return(model)
}

#rmse
rmse=function(mat1,mat2){
  p=nrow(mat1)
  val=sqrt(sum((mat1-mat2)^2))/p
  return(val)
}

#mnorm
mnorm=function(mat1,mat2){
  temp=abs(mat1-mat2)
  val=max(temp)
  return(val)
}

#spectral norm
spec_norm=function(mat1,mat2){
  temp=mat1-mat2
  temp=t(temp)%*%temp
  val=sqrt(max(eigen(temp)$values))
  return(val)
}

#accuracy
accuracy=function(edge1,edge2){
```

```
  edge_ind1=which(edge1==1)
  edge_ind0=which(edge1==0)

  true_edge1=which(edge2==1)
  true_edge0=which(edge2==0)


  tp=length(intersect(edge_ind1,true_edge1))
  tn=length(intersect(edge_ind0,true_edge0))
  fp=length(intersect(edge_ind1,true_edge0))
  fn=length(intersect(edge_ind0,true_edge1))

  sp=tn/(tn+fp)
  se=tp/(tp+fn)
  mcc=(tp*tn-fp*fn)/sqrt((tp+fp)*(tp+fn)*(tn+fp)*(tn+fn))
  return(c(sp,se,mcc))

}

#performance
bayesian_measure=function(n,v,lambda,tol,size,burnin,q,rep,true_cov){
  temp1=numeric(length=rep); temp2=numeric(length=rep)
  temp3=numeric(length=rep); temp4=numeric(length=rep)
  temp5=numeric(length=rep); temp6=numeric(length=rep)
  temp7=numeric(length=rep); temp8=numeric(length=rep)
  temp9=numeric(length=rep); temp10=numeric(length=rep)
  temp11=numeric(length=rep); temp12=numeric(length=rep)

  true_edge=edge_ind(true_cov,0.1^3)

  for(i in 1:rep){
   print(paste(c("repetition: ",i),sep=""))
   samp=rmvnorm(n,sigma=true_cov)
   s=t(samp)%*%samp/n
   mcmc_samp=mcmc(n,s,v,lambda,tol,size,burnin,q)
   mcmc_mpp=mpp(mcmc_samp)
   mcmc_mmp=mmp(mcmc_samp)
   mpp_bcd=bcd_covariance2(s,n,v,lambda,10000,0.1^3,mcmc_mpp)
   mmp_bcd=bcd_covariance2(s,n,v,lambda,10000,0.1^3,mcmc_mmp)

   temp1[i]=rmse(mpp_bcd,true_cov)
   temp2[i]=mnorm(mpp_bcd,true_cov)
   temp3[i]=spec_norm(mpp_bcd,true_cov)

   acc_mpp=accuracy(mcmc_mpp,true_edge)

   temp4[i]=acc_mpp[1]
   temp5[i]=acc_mpp[2]
   temp6[i]=acc_mpp[3]

   temp7[i]=rmse(mmp_bcd,true_cov)
   temp8[i]=mnorm(mmp_bcd,true_cov)
   temp9[i]=spec_norm(mmp_bcd,true_cov)
```

```
   acc_mmp=accuracy(mcmc_mmp,true_edge)

   temp10[i]=acc_mmp[1]
   temp11[i]=acc_mmp[2]
   temp12[i]=acc_mmp[3]
  }
  measure=list(mpp_rmse=temp1,mpp_mnorm=temp2,mpp_spec=temp3,mpp_sp=temp4,mpp_se=temp5,
               mpp_mcc=temp6, mmp_rmse=temp7, mmp_mnorm=temp8, mmp_spec=temp9, mmp_sp=temp10,
               mmp_se=temp11,mmp_mcc=temp12)
  return(measure)
}
```

Following code finds the minimizer of our objective function using our proposed Block Coordinate Descent method and implements our proposed Metropolis-Hastings algorithm. We obtained 15000 samples with 3000 burn-in. We repeat this procedure for 100 times.

```
bcd_covariance1=function(samp,n,v0,v1,lambda,tol_it,tol,edge){

  p=nrow(samp)

  #edge penalty
  edge_penalty=numeric(length=choose(p,2))
  edge_penalty[which(edge==0)]=1/(n*v0^2)
  edge_penalty[which(edge==1)]=1/(n*v1^2)
  Lambda=matrix(rep(0,p^2),nrow=p,ncol=p)
  Lambda[upper.tri(Lambda)]=edge_penalty
  Lambda=Lambda+t(Lambda)

  rho=lambda/n
  init_cov=diag(diag(samp))+rho*diag(p)

  # permutation matrix for looping through columns & rows
  perms=matrix(NA,nrow=p-1,ncol=p)
  permInt=1:p
  for(i in 1:ncol(perms))
  {
    perms[,i]<-permInt[-i]
  }

  cov=init_cov
  k=0

  for(i in 1:tol_it){

    cov_temp=cov

    for(j in 1:p){

      cov11=cov_temp[perms[,j],perms[,j]]; omega11=solve(cov11)
      cov12=cov_temp[perms[,j],j]

      s11=samp[perms[,j],perms[,j]]
      s12=samp[perms[,j],j]
      s22=samp[j,j]
```

```
      Lambda12=diag(Lambda[perms[,j],j])

      omega_s11=omega11%*%s11%*%omega11
      u=t(cov12)%*%omega_s11%*%cov12-2*t(s12)%*%omega11%*%cov12+s22
      u=as.numeric(u)

      gamma=(-1+sqrt(1+4*u*rho))/(2*rho)
      beta=solve(Lambda12+rho*omega11+omega_s11/gamma)%*%omega11%*%s12/gamma

      cov_temp[j,j]=gamma+t(beta)%*%omega11%*%beta
      cov_temp[perms[,j],j]=beta
      cov_temp[j,perms[,j]]=beta

    }

    error=sqrt(sum((cov_temp-cov)^2))
    k=k+1
    if(error<tol){
      return(cov_temp)
    }else{
      cov=cov_temp
    }
  }
  return(cov)
}

bcd_covariance2=function(samp,n,v,lambda,tol_it,tol,edge){

  p=nrow(samp)

  #edge penalty
  edge_penalty=rep(0,choose(p,2))
  edge_penalty[which(edge==1)]=1/(n*v^2)
  Lambda=matrix(rep(0,p^2),nrow=p,ncol=p)
  Lambda[upper.tri(Lambda)]=edge_penalty
  Lambda=Lambda+t(Lambda)

  edge_list=matrix(rep(0,p^2),nrow=p,ncol=p)
  edge_list[upper.tri(edge_list)]=edge
  edge_list=edge_list+t(edge_list)

  rho=lambda/n
  init_cov=diag(diag(samp))+rho*diag(p)

  # permutation matrix for looping through columns & rows
  perms=matrix(NA,nrow=p-1,ncol=p)
  permInt=1:p
  for(i in 1:ncol(perms))
  {
    perms[,i]<-permInt[-i]
  }

  cov=init_cov
```

```
  k=0

  for(i in 1:tol_it){

    cov_temp=cov

    for(j in 1:p){

      cov11=cov_temp[perms[,j],perms[,j]]; omega11=solve(cov11)
      cov12=cov_temp[perms[,j],j]

      s11=samp[perms[,j],perms[,j]]
      s12=samp[perms[,j],j]
      s22=samp[j,j]

      edge12=as.vector(edge_list[perms[,j],j])
      edge12_list=which(edge12==1)
      Lambda12=diag(Lambda[perms[,j],j])

      omega_s11=omega11%*%s11%*%omega11
      u=t(cov12)%*%omega_s11%*%cov12-2*t(s12)%*%omega11%*%cov12+s22
      u=as.numeric(u)

      gamma=(-1+sqrt(1+4*u*rho))/(2*rho)

      if(length(edge12_list)!=0){
        quad_mat=(Lambda12+rho*omega11+omega_s11/gamma)[edge12_list,edge12_list]
        obj_vec=(omega11%*%s12)[edge12_list]
        beta=solve(quad_mat)%*%obj_vec/gamma
        cov12[edge12_list]=beta
      }

      cov_temp[j,j]=gamma+t(cov12)%*%omega11%*%cov12
      cov_temp[perms[,j],j]=cov12
      cov_temp[j,perms[,j]]=cov12

    }

    error=sqrt(sum((cov_temp-cov)^2))
    k=k+1
    if(error<tol){
      return(cov_temp)
    }else{
      cov=cov_temp
    }
  }
  return(cov)
}

#objective function
objec=function(m,n,samp,edge,v,lambda){
  m_upper=m[upper.tri(m)]
  ind=which(edge==1)
```

```
  if(length(ind)>0){
    m_upper=m_upper[which(edge==1)]
  }else{
    m_upper=0
  }

  omega=solve(m)
  value=log(det(m))+sum(diag(samp%*%omega))+1/(n*v^2)*sum(m_upper^2)+lambda/n*sum(diag(m))
  return(value)
}

generate_t=function(p,edge){
  s=p+choose(p,2)
  ind=1:p
  off=(p+1):s
  off=off[which(edge==1)]
  total=c(ind,off)
  l=length(total)
  t=matrix(rep(0,p^2*l),nrow=p^2)
  x=diag(ind)
  x[upper.tri(x)]=(p+1):s
  x=x+t(x)-diag(diag(x))
  u=as.vector(x)
  for(i in 1:l){
    t[,i]=as.numeric(u==total[i])
  }
  return(t)
}

hessian=function(m,s,edge,n,v){
  p=nrow(m)
  t_full=generate_t(p,edge)
  m_inv=solve(m)
  full_hessian=(2*m_inv%*%s%*%m_inv-m_inv)%x%m_inv
  hessian_edge=t(t_full)%*%full_hessian%*%t_full
  l=nrow(hessian_edge)

  for(i in (p+1):l){
    hessian_edge[i,i]=hessian_edge[i,i]+2/(n*v^2)
  }

  return(hessian_edge)
}

#calculating posterior ratio
post_ratio=function(m1,m2,n,samp,edge1,edge2,v,lambda,q){
  post1=objec(m1,n,samp,edge1,v,lambda); post2=objec(m2,n,samp,edge2,v,lambda)
  post_diff=post1-post2
  edge_diff=sum(edge1)-sum(edge2)
  hessian1=hessian(m1,samp,edge1,n,v)
  hessian2=hessian(m2,samp,edge2,n,v)

  eigen1=eigen(hessian1)$values; eigen2=eigen(hessian2)$values
```

```
  l1=length(eigen1); l2=length(eigen2)
  if(l1>=l2){
    eigen2[(l2+1):l1]=1
  }else{
    eigen1[(l1+1):l2]=1
  }
  eigen_ratio=abs(prod(eigen1/eigen2))
  result=(q/((1-q)*v)*sqrt(4*pi/n))^edge_diff*exp(-n/2*post_diff)*1/sqrt(eigen_ratio)
  return(result)
}

unif_samp=function(edge,p){
  l=choose(p,2)
  samp=sample(1:l,1)
  result=edge
  if(edge[samp]==1){
    result[samp]=0
  }
  if(edge[samp]==0){
    result[samp]=1
  }
  return(result)
}

#our proposed Metropolis-Hastings algorithm
mcmc=function(n,s,v,lambda,tol,size,burnin,q){
  p=nrow(s)

  cov_init=s
  edge_init=edge_ind(s,0.1^3)

  edge_temp=edge_init
  cov_temp=cov_init

  total_samp=matrix(rep(0,size*choose(p,2)),nrow=size)

  for(i in 1:(size+burnin)){
    if(i<=burnin){
    edge_cand=unif_samp(edge_temp,p)
    cov_cand=bcd_covariance2(s,n,v,lambda,10000,tol,edge_cand)
    alpha=min(c(1,post_ratio(cov_cand,cov_temp,n,s,edge_cand,edge_temp,v,lambda,q)))
    u=runif(1,0,1)
    if(u<=alpha){
      edge_temp=edge_cand
      cov_temp=cov_cand
    }
    if(i%%100==0){
      print(paste(i,"th burnin",sep=" "))
    }
    }else{
      edge_cand=unif_samp(edge_temp,p)
      cov_cand=bcd_covariance2(s,n,v,lambda,10000,0.1^3,edge_cand)
      alpha=min(c(1,post_ratio(cov_cand,cov_temp,n,s,edge_cand,edge_temp,v,
```

```
                                lambda,q)))
      u=runif(1,0,1)
      if(u<=alpha){
        edge_temp=edge_cand
        cov_temp=cov_cand
      }
      total_samp[i-burnin,]=edge_temp
      if(i%%100==0){
        print(paste((i-burnin),"th sample",sep=" "))
      }
    }
  }
  return(total_samp)
}

#simulation using the proposed method
set.seed(1)
sssl_perf=bayesian_measure(250,1,250*0.06,0.1^3,12000,3000,2/11,100,curr_ex)
```

Following code implements covariance graphical lasso.

```
rmse_samp=numeric(length=100); mnorm_samp=numeric(length=100); spec_samp=numeric(length=100)
sp_samp=numeric(length=100); se_samp=numeric(length=100); mcc_samp=numeric(length=100)

step.size=100
tol=1e-3
P=matrix(1,12,12)
diag(P)=0
lam=0.06

for(i in 1:100){
  samp=rmvnorm(n=250,sigma=curr_ex)
  s=t(samp)%*%samp/250

  mm=spcov(Sigma=s,S=s,lambda=lam*P,step.size=step.size,n.inner.steps=200,
           thr.inner=0,tol.outer=tol,trace=1)$Sigma

  gl_edge=edge_ind(mm,0.1^3)

  rmse_samp[i]=rmse(s,curr_ex)
  mnorm_samp[i]=mnorm(s,curr_ex)
  spec_samp[i]=spec_norm(s,curr_ex)

  acc=accuracy(gl_edge,true_edge)
  sp_samp[i]=acc[1]
  se_samp[i]=acc[2]
  mcc_samp[i]=acc[3]
}
```

Following code gives the result of performance using sample covariance method.

```
set.seed(1)
for(i in 1:100){
  samp=rmvnorm(250,sigma=curr_ex)
  s=t(samp)%*%samp/250
```

```
  s_edge=edge_ind(s,0.1^3)

  rmse_samp[i]=rmse(s,curr_ex)
  mnorm_samp[i]=mnorm(s,curr_ex)
  spec_samp[i]=spec_norm(s,curr_ex)

  acc=accuracy(s_edge,true_edge)
  sp_samp[i]=acc[1]
  se_samp[i]=acc[2]
  mcc_samp[i]=acc[3]
}
```

Now, we provide the result. In our proposed method, we chose final model using Median Probability Model(MPP) or Maximum a Posteriori(MAP). We prove the result for both cases The value in parentheses denotes the standard deviation.

```
## # A tibble: 4 x 4
##   method          rmse         mnorm        spec_norm
##   <chr>           <chr>        <chr>        <chr>
## 1 proposed(mpp)   0.020(0.006) 0.147(0.072) 0.179(0.076)
## 2 proposed(map)   0.020(0.006) 0.147(0.072) 0.180(0.075)
## 3 graphical_lasso 0.029(0.006) 0.138(0.069) 0.237(0.063)
## 4 samp_cov        0.030(0.006) 0.154(0.076) 0.253(0.079)
```

```
## # A tibble: 4 x 4
##   method          sp           se           mcc
##   <chr>           <chr>        <chr>        <chr>
## 1 proposed(mpp)   0.999(0.006) 0.555(0.064) 0.702(0.051)
## 2 proposed(map)   0.998(0.007) 0.554(0.065) 0.698(0.052)
## 3 graphical_lasso 0.288(0.064) 0.962(0.051) 0.232(0.074)
## 4 samp_cov        0.042(0.030) 0.984(0.035) 0.056(0.092)
```

Overall, the simulation results do not show significant difference between proposed(mpp) and proposed(map). In terms of rmse, spectral norm, specificity, and mcc, our proposed method showed better performance than other competing methods. However, in terms of mnorm and sensitivity, our proposed method showed poor performance than other competing methods. Especially, for sensitivity, the poor performance may be due to our edge acceptance probability, which is $2/(p-1)$. Note that sample covariance method showed the poorest peformance compared to other methods, except for sensitivity. To sum up, our proposed method may perform well than other competing methods in terms of some measures.