CV result and modified SMMK alogrithm

Chanwoo Lee, October 25, 2020

1 Cross validation result

The probability estimation method is based on the reference paper (Method 2). I have checked that the cumulative sum based probability estimation (Method 1) performs worse than Method 2. Figure 1 shows the cross validation result on VSPLOT brain dataset. It shows that ADMM performs better than SMMK method based on training datset results while SMMK outperforms ADMM on test datsets. The best combination of rank and sparsity on test datsets from SMMK is (rank,sparsity) = (2,8) while (rank,sparsity) = (1,59) has greatest log-likelihood from ADMM. SMMK algorithm has region that both test and training performance better than lasso-logistic regression while ADMM has one point that has similar performance with lasso-logistic regression on test datsets and outperforms on trainig datasets.

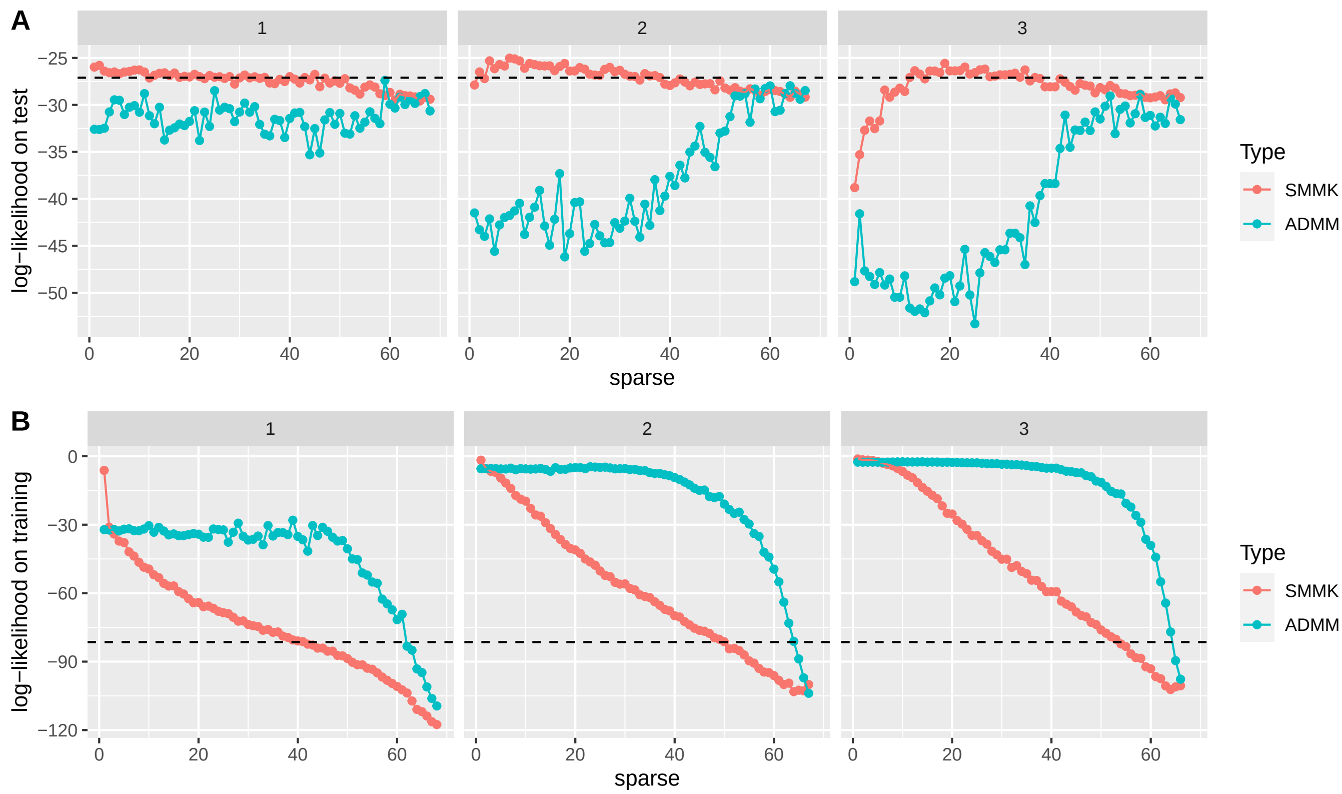


Figure 1: Cross validation results accross the rank and sparsity. Figure A shows the averaged log-likelihood values on test datsets while Figure B on training datasets. Dotted lines is the cross validation result based on lasso logistic regression.

2 Modification of SMMK algorithm

We have checked SMMK algorithm does not work well compare to ADMM. I found there are some reasons for the poor performance. In the algorithm, the following part makes the algorithm poor.

```
if(sparse>=1){
        B=sparse_matrix(B,r,sparse,sparse)
        P_row=svd(B)$u[,1:r]
        P_col=svd(B)$v[,1:r]
      }
```

There are two parts that need to be modified. The above codes are for post processing of coefficient matrix $\boldsymbol{B}$ after alternative updates. It consists of two parts.

1. Post-processing (`sparse_matrix`)

2. After the processing(`P_row=svd(B)$u[,1:r]`).

First, post-processing part chooses sparse structure and approximate the coefficient matrix $\boldsymbol{B}$ with sparse structure to the low rank matrix. As we discussed in the last meeting, sparse structure can not be learned once we choose sparse structure. Therefore, updating sparsity after each update is not needed. In addition, the way we approximate to the low rank matrix with sparsity is not finding the best matrix that minimizes the loss value but finding the closest matrix with respect to Frobenius norm. In these reasons, he function `sparse_matrix` makes the algorithm less accurate.

Second, notice that

$$\boldsymbol{B} = \boldsymbol{P}_{\text{row}}\boldsymbol{P}_{\text{row}}^T\left(\sum_{i=1}^n \alpha_i y_i \boldsymbol{X}_i\right) + \left(\sum_{i=1}^n \alpha_i y_i \boldsymbol{X}_i\right)\boldsymbol{P}_{\text{col}}\boldsymbol{P}_{\text{col}}^T. \tag{1}$$

(1) shows that right singular matrix of $\boldsymbol{B}$ is not necessarily $\boldsymbol{P}_{\text{row}}$ especially when $\boldsymbol{P}_{\text{row}}$ and $\boldsymbol{P}_{\text{col}}$ are near optimal where

$$\boldsymbol{P}_{\text{row}}\boldsymbol{P}_{\text{row}}^T\left(\sum_{i=1}^n \alpha_i y_i \boldsymbol{X}_i\right) \approx \left(\sum_{i=1}^n \alpha_i y_i \boldsymbol{X}_i\right)\boldsymbol{P}_{\text{col}}\boldsymbol{P}_{\text{col}}^T. \tag{2}$$

Therefore, setting $\boldsymbol{P}_{\text{row}}$ and $\boldsymbol{P}_{\text{col}}$ as the right and left singular matrices of the post processed $\boldsymbol{B}$ makes the update worse.

To improve those problems in the algorithm, I divided whole algorithm into two procedure. The first procedure is to decide the sparsity of the matrix. I followed the same procedure in the old algorithm. The second procedure is to find the best low rank matrix with given sparsity. In this procedure, we do not use low rank approximation but use alternating updates given non-zero columns and rows.

There are two shortcomings of the new modification.

1. Choosing right sparsity structure dominates all performance.

2. Coefficient matrix $\boldsymbol{B}$ is not exactly the rank r.

The previous algorithm has the same problem as the first problem. I verified that when the proportion of zero rows is less than around 0.5, the new algorithm finds non-zero rows successfully while it works poorly when the sparsity is high (Figure **??**).

The second problem arises because (2) is numerically deviated so that $\boldsymbol{B}$ from (1) has slightly higher rank than r. In previous algorithm, the low rank approximation forces the matrix $\boldsymbol{B}$ have rank r but this approximation makes the output far from the optimal point. To avoid this rank disparity, I added an option to choose between the following two models with setting Option 2 as default,

$$\text{Option 1: } y_i = \text{sign}\left(\langle \boldsymbol{C}\boldsymbol{P}^T, \phi(\boldsymbol{X}_i)\rangle + b\right),$$
$$\text{Option 2: } y_i = \text{sign}\left(\langle \boldsymbol{C}_{\text{row}}\boldsymbol{P}_{\text{row}}^T, \phi_{\text{row}}(\boldsymbol{X}_i)\rangle + \langle \boldsymbol{C}_{\text{col}}\boldsymbol{P}_{\text{col}}^T, \phi_{\text{col}}(\boldsymbol{X}_i)\rangle + b\right),$$

where $\phi : \mathbb{R}^{d_1\times d_2} \to \mathcal{H}_{\text{row}}$ is a feature mapping. In linear case, $\phi_{\text{row}}(\boldsymbol{X}_i) = \boldsymbol{X}_i$ and $\phi_{\text{col}}(\boldsymbol{X}_i) = \boldsymbol{X}_i^T$. New algorithm is in Section 3. There are one main function and two sub functions. `SMM` is the sub function for Option 1 while `SMMK` is for Option 2. `SMMK_sparse` is the main function.

3 Comparison and sanity check

3.1 New algorithm vs old algorithm

I briefly checked the performance between the old algorithm and new one. For VSPLOT dataset, I only checked classification performance at rank = 2 and sparsity = 20. New algorithm perfectly separated the dataset with 0 training error while old algorithm has 0.056 training error.

Another comparison is from a simple simulation. I generated feature matrices $\boldsymbol{X}_i \in \mathbb{R}^{10\times 10}$ i = 1,...,100. I assign the label response as,

$$y_i \overset{\text{ind}}{\sim} \text{Ber}\left(\text{logistic}(4 * \langle \boldsymbol{B}, \boldsymbol{X}_i\rangle)\right), \tag{3}$$

where the coefficient matrix $\boldsymbol{B} \in \mathbb{R}^{10\times 10}$ has rank 3 and 5 non-zero columns and rows (sparsity = 5). Figure 2 shows that new algorithm improve the estimation performance of coefficient matrix $\boldsymbol{B}$. Here Option 2 is used for the new algorithm.

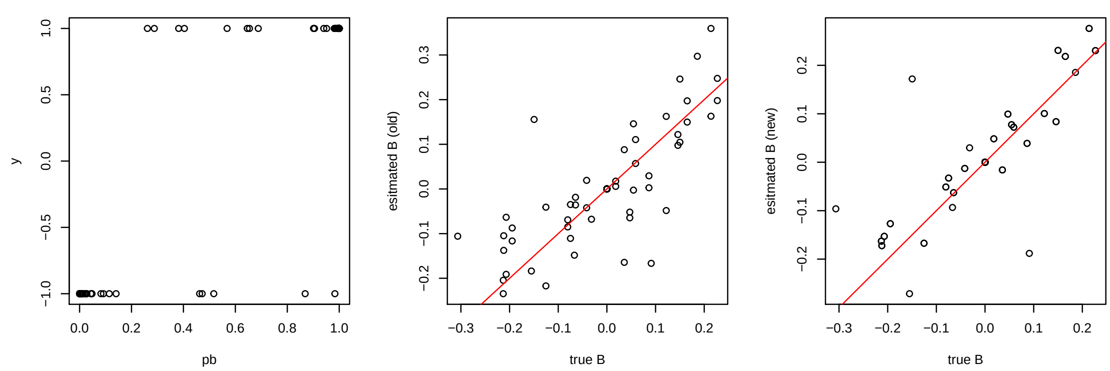


Figure 2: The first figure shows the true probabilities and the label responses in dataset. The second and third figure shows the estimation accuracy of the coefficient matrix $\boldsymbol{B}$ based on old and new algorithm in order.

The classifcation errors summarized in the following table.

	Old SMMK	New SMMK (Option 1)	New SMMK (Option 2)
classification error	0.28	0	0.02

3.2 Option 1 vs Option 2

I compare the performance of Option 1 and Option 2 in new algorithm. I use the same simulation setting as in (3) with the same sample size. I changed the sparsity in $\{1, 3, 5, 7\}$ and checked the performance with respect to classification on training datasets and estimation of the coefficient matrix.

The following table shows that the classification performance on the dataset. It shows that the performance of Option 1 and Option 2 are basically similar except when the sparsity is 7 out 10.

	sparse = 1	sparse = 3	sparse = 5	sparse = 7
Option 1	0.02	0	0	0.05
Option 2	0	0	0.04	0.27

Table 1: Classification errors according to sparsity and options

Figure 3 shows the performance in estimating the coefficient matrix $\boldsymbol{B}$ across different sparsities and different options. In addition, I added the case when I use Option 2 and use low-rank approximation from the output. It seems that Option 2 works slightly better than Option 1. When sparsity is high, we have poor estimation result as we expected. Angles between the coefficient matrix $\boldsymbol{B}$ and $\hat{\boldsymbol{B}}$ is 0 in most cases. When sparsity = 1,3,5, our algorithm successfully finds true non-zero columns and rows while the algorithm finds two non-zero columns and rows correctly when there are only 3 non-zero columns and rows.

4 New algorithm

```

################################### Option 1 ###################################
SMM = function(X,y,r,kernel_row = c("linear","poly","exp","const"),kernel_col = c(
    "linear","poly","exp","const"),cost = 10, rep = 1, p = .5){
  result = list()

  # Default is linear kernel.
  kernel_row <- match.arg(kernel_row)
  if (kernel_row == "linear") {
    kernel_row = linearkernel
  }else if(kernel_row == "poly"){
    kernel_row = polykernel
  }else if(kernel_row =="exp"){
    kernel_row = expkernel
  }else if(kernel_row == "const"){
    kernel_row = constkernel
  }

  kernel_col <- match.arg(kernel_col)
  if (kernel_col == "linear") {
    kernel_col = linearkernel
  }else if(kernel_col == "poly"){
    kernel_col = polykernel
```

```
  }else if(kernel_col =="exp"){
    kernel_col = expkernel
  }else if(kernel_col == "const"){
    kernel_col = constkernel
  }

  d1 = nrow(X[[1]]); d2 = ncol(X[[1]]); n = length(X)
  K = Karray(X,kernel_row,type="row")
  #K_col = Karray(X,kernel_col,type="col")
  compareobj = 10^10


  for(nsim in 1:rep){
    error = 10; iter = 0; obj = 10^10
    # initialize P_row,P_col
    P =  randortho(d1)[,1:r,drop = F]



    while((iter < 20)&(error >10^-3)){
      # update C
      W = P%*%t(P);# W_col = P_col%*%t(P_col)
      Dmat=matrix(unfold(K,c(1,2),c(3,4))@data%*%c(W),nrow=n,ncol=n)

      dvec = rep(1,n)
      Dmat = Makepositive((y%*%t(y))*Dmat)
      Amat = cbind(y,diag(1,n),-diag(1,n))
      bvec = c(rep(0,1+n),ifelse(y==1,-cost*(1-p),-cost*p))
      res = solve.QP(Dmat,dvec,Amat,bvec,meq =1)
      alpha=res$solution

      CPh=ttl(K,list(t(as.matrix(y*alpha)),t(P)),ms=c(1,3))
      #CPh_col=ttl(K_col,list(t(as.matrix(y*alpha)),t(P_col)),ms=c(1,3))

      CC=ttl(CPh,list(t(as.matrix(y*alpha)),t(P)),ms=c(2,4))
      #CC_col=ttl(CPh_col,list(t(as.matrix(y*alpha)),t(P_col)),ms=c(2,4))

      CC=as.matrix(CC@data[1,1,,])
      #CC_col=as.matrix(CC_col@data[1,1,,])

      factors=unfold(ttm(CPh,sqrtm(Makepositive(CC))$Binv,3),2,c(1,3,4))@data
      #factor_col=unfold(ttm(CPh_col,sqrtm(Makepositive(CC_col))$Binv,3),2,c(1,3,4))@data
      Dmat=factors%*%t(factors)#+factor_col%*%t(factor_col)


      dvec = rep(1,n)
      Dmat = Makepositive((y%*%t(y))*Dmat)
      Amat = cbind(y,diag(1,n),-diag(1,n))
      bvec = c(rep(0,1+n),ifelse(y==1,-cost*(1-p),-cost*p))
      res = solve.QP(Dmat,dvec,Amat,bvec,meq =1)
      alpha=res$solution
      obj=c(obj,-res$value)
      iter = iter+1
      error = abs(-obj[iter+1]+obj[iter])/obj[iter]

      # P formula
      P=ttm(CPh,t(as.matrix(alpha*y)),2)[1,1,,]@data
      #P_col=ttm(CPh_col,t(as.matrix(alpha*y)),2)[1,1,,]@data
```

```
    P=matrix(P,nrow=r)
    #P_col=matrix(P_col,nrow=r)

    P = svd(P)$v
    #P_col = svd(P_col)$v

    #### sparse model

    B=0
    A = 0
    for(i in 1:n){
      B=B+alpha[i]*y[i]*P%*%t(P)%*%X[[i]]#+alpha[i]*y[i]*X[[i]]%*%P_col%*%t(P_col)
      A = A+alpha[i]*y[i]*X[[i]]
    }


  }
  if(compareobj>obj[iter+1]){
    P_optimum=P; #P_col_optimum=P_col;
    obj_optimum=obj;
    compareobj=obj[iter+1]
  }
}



P= P_optimum;# P_col= P_col_optimum;
W = P%*%t(P);# W_col = P_col%*%t(P_col);

Dmat=matrix(unfold(K,c(1,2),c(3,4))@data%*%c(W),nrow=n,ncol=n)

dvec = rep(1,n)
Dmat = Makepositive((y%*%t(y))*Dmat)
Amat = cbind(y,diag(1,n),-diag(1,n))
bvec = c(rep(0,1+n),ifelse(y==1,-cost*(1-p),-cost*p))
res = solve.QP(Dmat,dvec,Amat,bvec,meq =1)
alpha=res$solution

slope = function(Xnew){
  newK = rep(0,n)
  for( i in 1:n){
    newK[i] = sum(W*kernel_row(t(Xnew),t(X[[i]])))
  }

  return(sum(alpha*y*newK))
}

# intercept part estimation (update b)
yfit=Dmat%*%(alpha*y) ## faster than lapply

B=0;
for(i in 1:n){
  B=B+alpha[i]*y[i]*P%*%t(P)%*%X[[i]]#+alpha[i]*y[i]*X[[i]]%*%P_col%*%t(P_col)
}


positive=min(yfit[y==1])
```

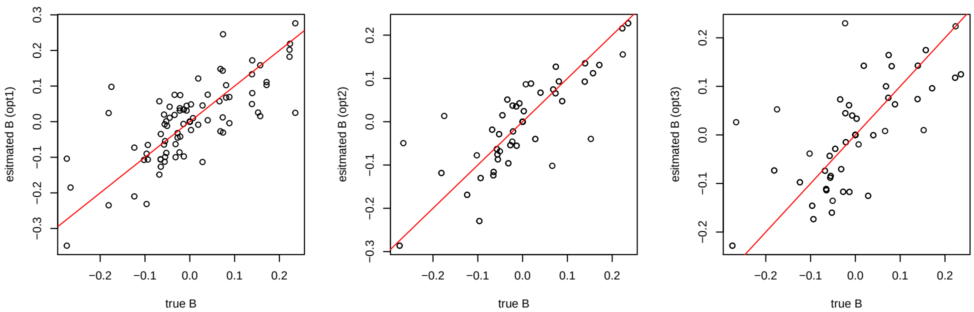
```
  negative=max(yfit[y==-1])
  if ((1-positive)<(-1-negative)) {
    intercept = -(positive+negative)/2
  }else{
    gridb0 = seq(from = -1-negative,to = 1-positive,length = 100)
    intercept = gridb0[which.min(sapply(gridb0,function(b) objective(b,yfit,y,p = p)))]
  }
  compareobj = obj[iter+1]
  predictor = function(Xnew) sign(slope(Xnew)+intercept)

  result$alpha = alpha
  result$slope = slope; result$predict = predictor
  result$intercept = intercept;
  result$P = P; #result$P_col = P_col;
  result$obj = obj[-1]; result$iter = iter;
  result$error = error;
  result$fitted=yfit+intercept; ## add fitted value as a criterium to select cost
  result$B=B
  return(result)
}




################################### Option 2 ###################################

SMMK = function(X,y,r,kernel_row = c("linear","poly","exp","const"),kernel_col = c("linear","poly","exp","const"), cost = 10, rep = 1, p = .5){
  result = list()

  # Default is linear kernel.
  kernel_row <- match.arg(kernel_row)
  if (kernel_row == "linear") {
    kernel_row = linearkernel
  }else if(kernel_row == "poly"){
    kernel_row = polykernel
  }else if(kernel_row =="exp"){
    kernel_row = expkernel
  }else if(kernel_row == "const"){
    kernel_row = constkernel
  }

  kernel_col <- match.arg(kernel_col)
  if (kernel_col == "linear") {
    kernel_col = linearkernel
  }else if(kernel_col == "poly"){
    kernel_col = polykernel
  }else if(kernel_col =="exp"){
    kernel_col = expkernel
  }else if(kernel_col == "const"){
    kernel_col = constkernel
  }

  d1 = nrow(X[[1]]); d2 = ncol(X[[1]]); n = length(X)
  K_row = Karray(X,kernel_row,type="row")
  K_col = Karray(X,kernel_col,type="col")
  compareobj = 10^10

```

```
# Choose non zero columns and rows



for(nsim in 1:rep){
  error = 10; iter = 0; obj = 10^10
  # initialize P_row,P_col
  if (d1 == d2) {
    P_row <- P_col <- randortho(d1)[,1:r,drop = F]
  }else{
    P_row = randortho(d1)[,1:r,drop = F]; P_col = randortho(d2)[,1:r,drop = F]
  }


  while((iter < 20)&(error >10^-3)){
    # update C
    W_row = P_row%*%t(P_row); W_col = P_col%*%t(P_col)
    Dmat=matrix(unfold(K_row,c(1,2),c(3,4))@data%*%c(W_row)+unfold(K_col,c(1,2),c(3,4))@data%*%c(W_col),nrow=n,ncol=n)

    dvec = rep(1,n)
    Dmat = Makepositive((y%*%t(y))*Dmat)
    Amat = cbind(y,diag(1,n),-diag(1,n))
    bvec = c(rep(0,1+n),ifelse(y==1,-cost*(1-p),-cost*p))
    res = solve.QP(Dmat,dvec,Amat,bvec,meq =1)
    alpha=res$solution

    CPh_row=ttl(K_row,list(t(as.matrix(y*alpha)),t(P_row)),ms=c(1,3))
    CPh_col=ttl(K_col,list(t(as.matrix(y*alpha)),t(P_col)),ms=c(1,3))

    CC_row=ttl(CPh_row,list(t(as.matrix(y*alpha)),t(P_row)),ms=c(2,4))
    CC_col=ttl(CPh_col,list(t(as.matrix(y*alpha)),t(P_col)),ms=c(2,4))

    CC_row=as.matrix(CC_row@data[1,1,,])
    CC_col=as.matrix(CC_col@data[1,1,,])

    factor_row=unfold(ttm(CPh_row,sqrtm(Makepositive(CC_row))$Binv,3),2,c(1,3,4))@data
    factor_col=unfold(ttm(CPh_col,sqrtm(Makepositive(CC_col))$Binv,3),2,c(1,3,4))@data
    Dmat=factor_row%*%t(factor_row)+factor_col%*%t(factor_col)


    dvec = rep(1,n)
    Dmat = Makepositive((y%*%t(y))*Dmat)
    Amat = cbind(y,diag(1,n),-diag(1,n))
    bvec = c(rep(0,1+n),ifelse(y==1,-cost*(1-p),-cost*p))
    res = solve.QP(Dmat,dvec,Amat,bvec,meq =1)
    alpha=res$solution
    obj=c(obj,-res$value)
    iter = iter+1
    error = abs(-obj[iter+1]+obj[iter])/obj[iter]

    # P formula
    P_row=ttm(CPh_row,t(as.matrix(alpha*y)),2)[1,1,,]@data
    P_col=ttm(CPh_col,t(as.matrix(alpha*y)),2)[1,1,,]@data

    P_row=matrix(P_row,nrow=r)
    P_col=matrix(P_col,nrow=r)
```

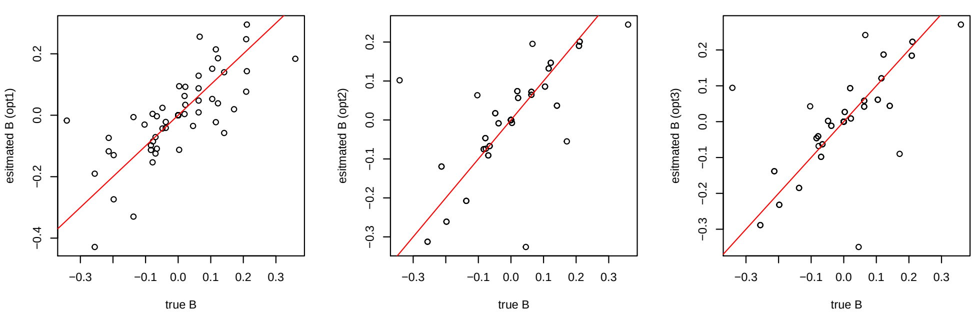
```

    P_row = svd(P_row)$v
    P_col = svd(P_col)$v



    B=0
    for(i in 1:n){
      B=B+alpha[i]*y[i]*P_row%*%t(P_row)%*%X[[i]]+alpha[i]*y[i]*X[[i]]%*%P_col%*%t(P_col)
    }

  }
  if(compareobj>obj[iter+1]){
    P_row_optimum=P_row; P_col_optimum=P_col;
    obj_optimum=obj;
    compareobj=obj[iter+1]
  }
}



P_row= P_row_optimum; P_col= P_col_optimum;
W_row = P_row%*%t(P_row); W_col = P_col%*%t(P_col);

Dmat=K=matrix(unfold(K_row,c(1,2),c(3,4))@data%*%c(W_row)+unfold(K_col,c(1,2),c(3,4))@data%*%c(W_col),nrow=n,ncol=n)

dvec = rep(1,n)
Dmat = Makepositive((y%*%t(y))*Dmat)
Amat = cbind(y,diag(1,n),-diag(1,n))
bvec = c(rep(0,1+n),ifelse(y==1,-cost*(1-p),-cost*p))
res = solve.QP(Dmat,dvec,Amat,bvec,meq =1)
alpha=res$solution

slope = function(Xnew){
  newK = rep(0,n)
  for( i in 1:n){
    newK[i] = sum(W_row*kernel_row(t(Xnew),t(X[[i]])))+
      sum(W_col*kernel_col(Xnew,X[[i]]))
  }

  return(sum(alpha*y*newK))
}

# intercept part estimation (update b)
yfit=K%*%(alpha*y) ## faster than lapply

B=0;
for(i in 1:n){
  B=B+alpha[i]*y[i]*P_row%*%t(P_row)%*%X[[i]]+alpha[i]*y[i]*X[[i]]%*%P_col%*%t(P_col)
}


positive=min(yfit[y==1])
negative=max(yfit[y==-1])
if ((1-positive)<(-1-negative)) {
  intercept = -(positive+negative)/2
```

```
  }else{
    gridb0 = seq(from = -1-negative,to = 1-positive,length = 100)
    intercept = gridb0[which.min(sapply(gridb0,function(b) objective(b,yfit,y,p = p)))]
  }
  compareobj = obj[iter+1]
  predictor = function(Xnew) sign(slope(Xnew)+intercept)

  result$alpha = alpha
  result$slope = slope; result$predict = predictor
  result$intercept = intercept;
  result$P_row = P_row; result$P_col = P_col;
  result$obj = obj[-1]; result$iter = iter;
  result$error = error;
  result$fitted=yfit+intercept; ## add fitted value as a criterium to select cost
  result$B=B
  return(result)
}




###################################Combination##################################



SMMK_sparse = function(X,y,r,kernel_row = c("linear","poly","exp","const"),kernel_col = c("linear","poly","exp","const"),option = c("approximate","exact"), cost = 10, rep = 1, p = .5,sparse=0){
  result = list()

  option <- match.arg(option)

  if(sparse>0){
    d1 = nrow(X[[1]]); d2 = ncol(X[[1]]); n = length(X)
    res = SMMK(X,y,r,kernel_row,kernel_col,cost,rep,p)
    initB = res$B
    row_o = order(diag(initB%*%t(initB)),decreasing = T)[1:(d1-sparse)]
    col_o = order(diag(t(initB)%*%initB),decreasing = T)[1:(d1-sparse)]
  }else{
    row_o = 1:d1; col_o = 1:d2
  }



  X_sp = lapply(X,function(x) x[row_o,col_o])
  d1sp = nrow(X_sp[[1]]); d2sp = ncol(X_sp[[1]]); n = length(X_sp)
  if(option == "exact"){
    res = SMM(X_sp,y,r,kernel_row,kernel_col,cost,rep,p)
  }else{
    res = SMMK(X_sp,y,r,kernel_row,kernel_col,cost,rep,p)
  }


  B = matrix(0,nrow = d1,ncol = d2)
  B[row_o,col_o] = res$B

  slope = function(Xnew) res$slope(Xnew[row_o,col_o])
  intercept = res$intercept
```

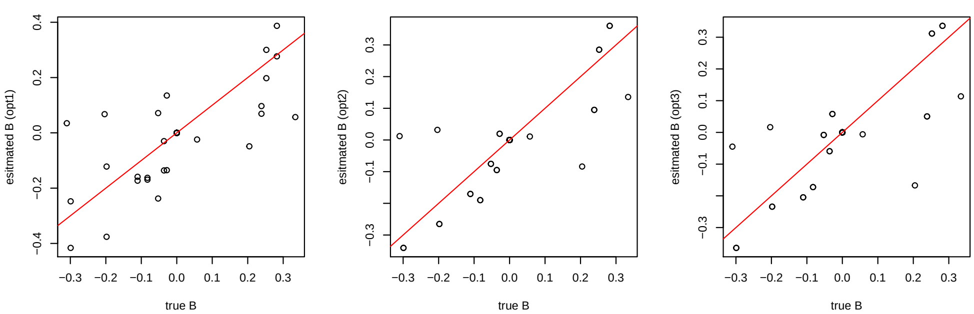
```
  predictor = function(Xnew) sign(slope(Xnew)+intercept)

  result$alpha = res$alpha
  result$slope = slope; result$predict = predictor
  result$intercept = intercept;
  result$P_row = res$P_row; result$P_col = res$P_col;
  result$obj = res$obj; result$iter = res$iter;
  result$error = res$error;
  result$fitted=res$fitted;
  result$B=B
  return(result)
}
```



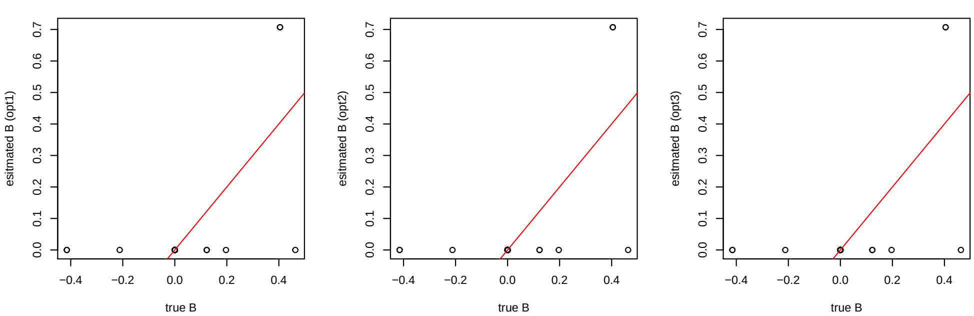
(a) When the number of non zero rows = 9 (sparsity = 1)



(b) When the number of non zero rows = 7 (sparsity = 3)



(c) When the number of non zero rows = 5 (sparsity = 5)



(d) When the number of non zero rows = 3 (sparsity = 7)

Figure 3: True coefficient $\boldsymbol{B}$ versus estimated coefficient $\hat{\boldsymbol{B}}$. The first column is when Option 1 is used while the second column is when Option 2 is used. The last column is when Option 2 is used and low rank approximation is implemented to the output.