

# CV result and modified SMMK algorithm

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 VSPLIT brain dataset. It shows that ADMM performs better than SMMK method based on training dataset results while SMMK outperforms ADMM on test datasets. The best combination of rank and sparsity on test datasets from SMMK is  $(\text{rank}, \text{sparsity}) = (2, 8)$  while  $(\text{rank}, \text{sparsity}) = (1, 59)$  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 datasets and outperforms on trainig datasets.

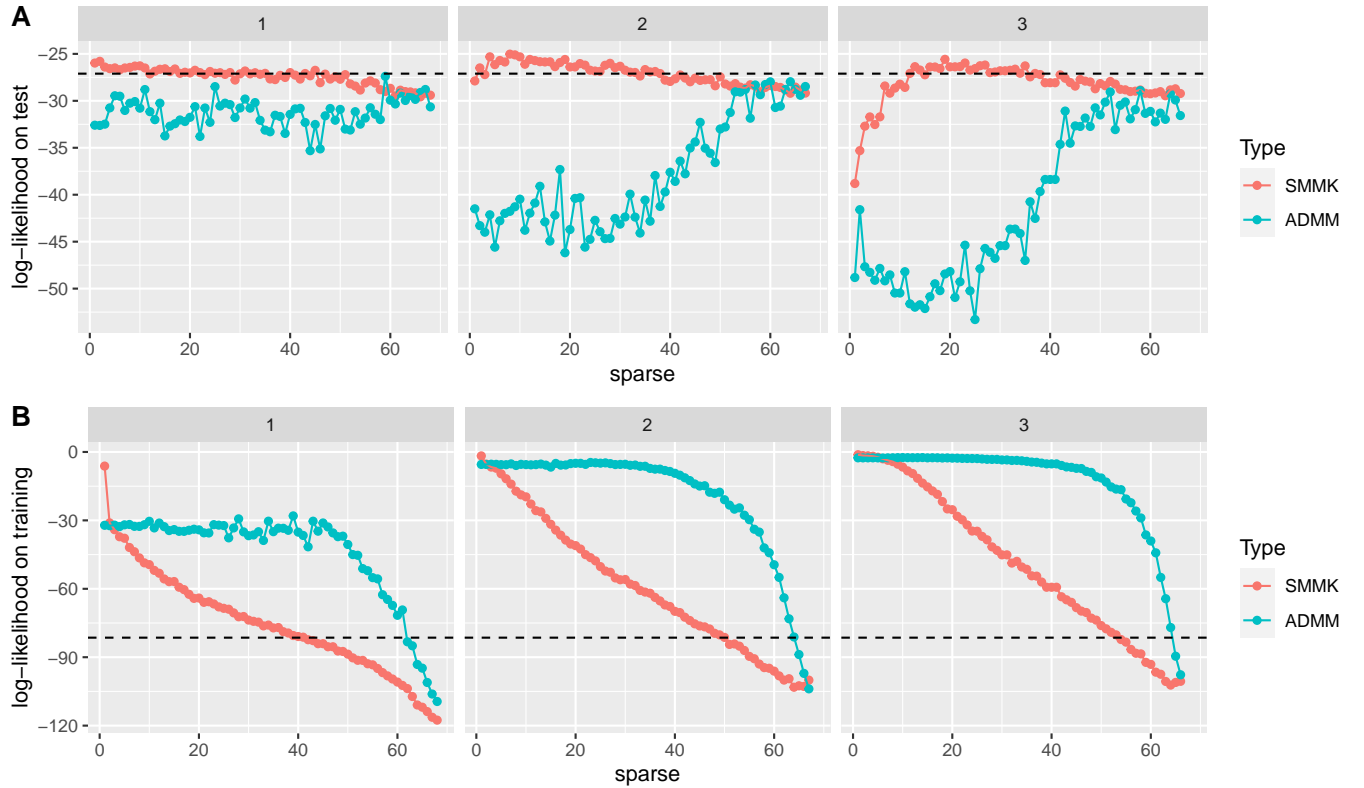


Figure 1: Cross validation results accross the rank and sparsity. Figure A shows the averaged log-likelihood values on test datasets while Figure B on training datasets. Dotted lines is the cross validation result based on lasso logistic regression.

## 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.

```

1 if(sparse>=1){
2     B=sparse_matrix(B,r,sparse,sparse)
3     P_row=svd(B)$u[,1:r]
4     P_col=svd(B)$v[,1:r]
5 }

```

There are two parts that need to be modified. The above codes are for post processing of coefficient matrix  $\mathbf{B}$  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  $\mathbf{B}$  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, the function `sparse_matrix` makes the algorithm less accurate.

Second, notice that

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

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

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

Therefore, setting  $\mathbf{P}_{\text{row}}$  and  $\mathbf{P}_{\text{col}}$  as the right and left singular matrices of the post processed  $\mathbf{B}$  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  $\mathbf{B}$  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  $\mathbf{B}$  from (1) has slightly higher rank than  $r$ . In previous algorithm, the low rank approximation forces the matrix  $\mathbf{B}$  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,

$$\text{Option 1: } y_i = \text{sign}(\langle \mathbf{C}\mathbf{P}^T, \phi(\mathbf{X}_i) \rangle + b),$$

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

where  $\phi : \mathbb{R}^{d_1 \times d_2} \rightarrow \mathcal{H}_{\text{row}}$  is a feature mapping. In linear case,  $\phi_{\text{row}}(\mathbf{X}_i) = \mathbf{X}_i$  and  $\phi_{\text{col}}(\mathbf{X}_i) = \mathbf{X}_i^T$ . 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 VSPLIT 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  $\mathbf{X}_i \in \mathbb{R}^{10 \times 10}$   $i = 1, \dots, 100$ . I assign the label response as,

$$y_i \stackrel{\text{ind}}{\sim} \text{Ber}(\text{logistic}(4 * \langle \mathbf{B}, \mathbf{X}_i \rangle)), \quad (3)$$

where the coefficient matrix  $\mathbf{B} \in \mathbb{R}^{10 \times 10}$  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  $\mathbf{B}$ . Here Option 2 is used for the new algorithm.

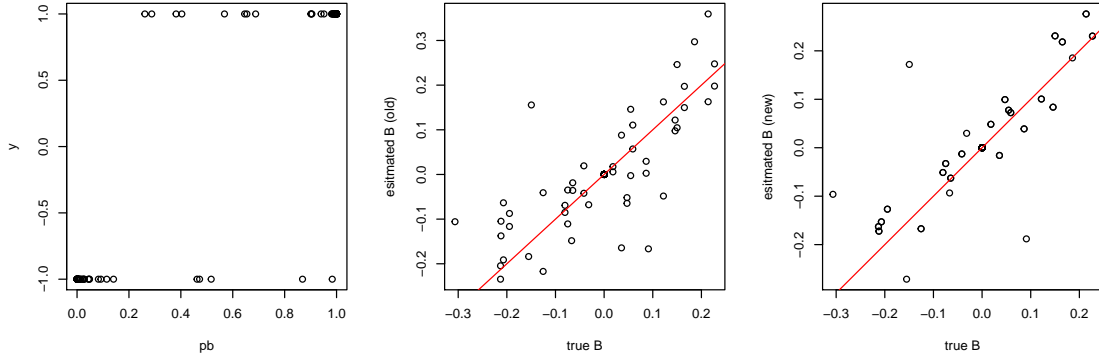


Figure 2: The first figure shows the true probabilities and the label responses in dataset. The second and third figure shows the estimation accuracy of the coefficient matrix  $\mathbf{B}$  based on old and new algorithm in order.

The classification 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  $\mathbf{B}$  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  $\mathbf{B}$  and  $\hat{\mathbf{B}}$  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

```

1 ##### Option 1 #####
2 SMM = function(X,y,r,kernel_row = c("linear","poly","exp","const"),kernel_col = c(
3   "linear","poly","exp","const"),cost = 10, rep = 1, p = .5){
4   result = list()
5
6   # Default is linear kernel.
7   kernel_row <- match.arg(kernel_row)
8   if (kernel_row == "linear") {
9     kernel_row = linearkernel
10  }else if(kernel_row == "poly"){
11    kernel_row = polykernel
12  }else if(kernel_row == "exp"){
13    kernel_row = expkernel
14  }else if(kernel_row == "const"){
15    kernel_row = constkernel
16  }
17
18  kernel_col <- match.arg(kernel_col)
19  if (kernel_col == "linear") {
20    kernel_col = linearkernel
21  }else if(kernel_col == "poly"){
22    kernel_col = polykernel

```

```

23 }else if(kernel_col == "exp"){
24     kernel_col = expkernel
25 }else if(kernel_col == "const"){
26     kernel_col = constkernel
27 }
28
29 d1 = nrow(X[[1]]); d2 = ncol(X[[1]]); n = length(X)
30 K = Karray(X, kernel_row, type="row")
31 #K_col = Karray(X, kernel_col, type="col")
32 compareobj = 10^10
33
34
35 for(nsim in 1:rep){
36     error = 10; iter = 0; obj = 10^10
37     # initialize P_row, P_col
38     P = randortho(d1)[,1:r, drop = F]
39
40
41
42     while((iter < 20) & (error > 10^-3)){
43         # update C
44         W = P%*%t(P); # W_col = P_col%*%t(P_col)
45         Dmat = matrix(unfold(K, c(1,2), c(3,4))@data%*%c(W), nrow=n, ncol=n)
46
47         dvec = rep(1,n)
48         Dmat = Makepositive((y%*%t(y))*Dmat)
49         Amat = cbind(y, diag(1,n), -diag(1,n))
50         bvec = c(rep(0,1+n), ifelse(y==1, -cost*(1-p), -cost*p))
51         res = solve.QP(Dmat, dvec, Amat, bvec, meq = 1)
52         alpha = res$solution
53
54         CPh = ttm(K, list(t(as.matrix(y*alpha)), t(P)), ms=c(1,3))
55         #CPh_col = ttm(K_col, list(t(as.matrix(y*alpha)), t(P_col)), ms=c(1,3))
56
57         CC = ttm(CPh, list(t(as.matrix(y*alpha)), t(P)), ms=c(2,4))
58         #CC_col = ttm(CPh_col, list(t(as.matrix(y*alpha)), t(P_col)), ms=c(2,4))
59
60         CC = as.matrix(CC@data[1,1,,])
61         #CC_col = as.matrix(CC_col@data[1,1,,])
62
63         factors = unfold(ttm(CPh, sqrtm(Makepositive(CC))$Binv, 3), 2, c(1,3,4))@data
64         #factor_col = unfold(ttm(CPh_col, sqrtm(Makepositive(CC_col))$Binv, 3), 2, c
65         (1,3,4))@data
66         Dmat = factors%*%t(factors) # + factor_col%*%t(factor_col)
67
68
69         dvec = rep(1,n)
70         Dmat = Makepositive((y%*%t(y))*Dmat)
71         Amat = cbind(y, diag(1,n), -diag(1,n))
72         bvec = c(rep(0,1+n), ifelse(y==1, -cost*(1-p), -cost*p))
73         res = solve.QP(Dmat, dvec, Amat, bvec, meq = 1)
74         alpha = res$solution
75         obj = c(obj, -res$value)
76         iter = iter+1
77         error = abs(-obj[iter+1] + obj[iter])/obj[iter]
78
79         # P formula
80         P = ttm(CPh, t(as.matrix(alpha*y)), 2)[1,1,,]@data
81         #P_col = ttm(CPh_col, t(as.matrix(alpha*y)), 2)[1,1,,]@data

```

```

81
82     P=matrix(P,nrow=r)
83     #P_col=matrix(P_col,nrow=r)
84
85     P = svd(P)$v
86     #P_col = svd(P_col)$v
87
88     #### sparse model
89
90     B=0
91     A = 0
92     for(i in 1:n){
93         B=B+alpha[i]*y[i]*P%%t(P)%%X[[i]]#+alpha[i]*y[i]*X[[i]]%%P_col%%t(P_
94         col)
95         A = A+alpha[i]*y[i]*X[[i]]
96     }
97
98 }
99 if(compareobj>obj[iter+1]){
100     P_optimum=P; #P_col_optimum=P_col;
101     obj_optimum=obj;
102     compareobj=obj[iter+1]
103 }
104 }
105
106
107
108 P= P_optimum;# P_col= P_col_optimum;
109 W = P%%t(P);# W_col = P_col%%t(P_col);
110
111 Dmat=matrix(unfold(K,c(1,2),c(3,4))@data%%c(W),nrow=n,ncol=n)
112
113 dvec = rep(1,n)
114 Dmat = Makepositive((y%%t(y))*Dmat)
115 Amat = cbind(y,diag(1,n),-diag(1,n))
116 bvec = c(rep(0,1+n),ifelse(y==1,-cost*(1-p),-cost*p))
117 res = solve.QP(Dmat,dvec,Amat,bvec,meq =1)
118 alpha=res$solution
119
120 slope = function(Xnew){
121     newK = rep(0,n)
122     for( i in 1:n){
123         newK[i] = sum(W*kernel_row(t(Xnew),t(X[[i]])))
124     }
125
126     return(sum(alpha*y*newK))
127 }
128
129 # intercept part estimation (update b)
130 yfit=Dmat%%(alpha*y) ## faster than lapply
131
132 B=0;
133 for(i in 1:n){
134     B=B+alpha[i]*y[i]*P%%t(P)%%X[[i]]#+alpha[i]*y[i]*X[[i]]%%P_col%%t(P_col)
135 }
136
137
138 positive=min(yfit[y==1])

```

```

139 negative=max(yfit[y== -1])
140 if ((1-positive)<(-1-negative)) {
141   intercept = -(positive+negative)/2
142 }else{
143   gridb0 = seq(from = -1-negative,to = 1-positive,length = 100)
144   intercept = gridb0[which.min(sapply(gridb0,function(b) objective(b,yfit,y,p =
145     p)))]
146 }
147 compareobj = obj[iter+1]
148 predictor = function(Xnew) sign(slope(Xnew)+intercept)
149
150 result$alpha = alpha
151 result$slope = slope; result$predict = predictor
152 result$intercept = intercept;
153 result$P = P; #result$P_col = P_col;
154 result$obj = obj[-1]; result$iter = iter;
155 result$error = error;
156 result$fitted=yfit+intercept; ## add fitted value as a criterium to select cost
157 result$B=B
158 return(result)
159 }
160
161
162
163 ##### Option 2 #####
164
165 SMMK = function(X,y,r,kernel_row = c("linear","poly","exp","const"),kernel_col = c
166   ("linear","poly","exp","const"), cost = 10, rep = 1, p = .5){
167   result = list()
168
169   # Default is linear kernel.
170   kernel_row <- match.arg(kernel_row)
171   if (kernel_row == "linear") {
172     kernel_row = linearkernel
173   }else if(kernel_row == "poly"){
174     kernel_row = polykernel
175   }else if(kernel_row == "exp"){
176     kernel_row = expkernel
177   }else if(kernel_row == "const"){
178     kernel_row = constkernel
179   }
180
181   kernel_col <- match.arg(kernel_col)
182   if (kernel_col == "linear") {
183     kernel_col = linearkernel
184   }else if(kernel_col == "poly"){
185     kernel_col = polykernel
186   }else if(kernel_col == "exp"){
187     kernel_col = expkernel
188   }else if(kernel_col == "const"){
189     kernel_col = constkernel
190   }
191
192   d1 = nrow(X[[1]]); d2 = ncol(X[[1]]); n = length(X)
193   K_row = Karray(X,kernel_row,type="row")
194   K_col = Karray(X,kernel_col,type="col")
195   compareobj = 10^10

```

```

196 # Choose non zero columns and rows
197
198
199
200 for(nsim in 1:rep){
201   error = 10; iter = 0; obj = 10^10
202   # initialize P_row,P_col
203   if (d1 == d2) {
204     P_row <- P_col <- randortho(d1)[,1:r,drop = F]
205   }else{
206     P_row = randortho(d1)[,1:r,drop = F]; P_col = randortho(d2)[,1:r,drop = F]
207   }
208
209
210   while((iter < 20)&(error >10^-3)){
211     # update C
212     W_row = P_row%*%t(P_row); W_col = P_col%*%t(P_col)
213     Dmat=matrix(unfold(K_row,c(1,2),c(3,4))@data%*%c(W_row)+unfold(K_col,c(1,2),
214 c(3,4))@data%*%c(W_col),nrow=n,ncol=n)
215
216     dvec = rep(1,n)
217     Dmat = Makepositive((y%*%t(y))*Dmat)
218     Amat = cbind(y,diag(1,n),-diag(1,n))
219     bvec = c(rep(0,1+n),ifelse(y==1,-cost*(1-p),-cost*p))
220     res = solve.QP(Dmat,dvec,Amat,bvec,meq =1)
221     alpha=res$solution
222
223     CPh_row=ttm(K_row,list(t(as.matrix(y*alpha)),t(P_row)),ms=c(1,3))
224     CPh_col=ttm(K_col,list(t(as.matrix(y*alpha)),t(P_col)),ms=c(1,3))
225
226     CC_row=ttm(CPh_row,list(t(as.matrix(y*alpha)),t(P_row)),ms=c(2,4))
227     CC_col=ttm(CPh_col,list(t(as.matrix(y*alpha)),t(P_col)),ms=c(2,4))
228
229     CC_row=as.matrix(CC_row@data[1,1,,])
230     CC_col=as.matrix(CC_col@data[1,1,,])
231
232     factor_row=unfold(ttm(CPh_row,sqrtm(Makepositive(CC_row)))$Binv,3),2,c(1,3,4)
233 )@data
234     factor_col=unfold(ttm(CPh_col,sqrtm(Makepositive(CC_col)))$Binv,3),2,c(1,3,4)
235 )@data
236     Dmat=factor_row%*%t(factor_row)+factor_col%*%t(factor_col)
237
238     dvec = rep(1,n)
239     Dmat = Makepositive((y%*%t(y))*Dmat)
240     Amat = cbind(y,diag(1,n),-diag(1,n))
241     bvec = c(rep(0,1+n),ifelse(y==1,-cost*(1-p),-cost*p))
242     res = solve.QP(Dmat,dvec,Amat,bvec,meq =1)
243     alpha=res$solution
244     obj=c(obj,-res$value)
245     iter = iter+1
246     error = abs(-obj[iter+1]+obj[iter])/obj[iter]
247
248     # P formula
249     P_row=ttm(CPh_row,t(as.matrix(alpha*y)),2)[1,1,,]@data
250     P_col=ttm(CPh_col,t(as.matrix(alpha*y)),2)[1,1,,]@data
251
252     P_row=matrix(P_row,nrow=r)
253     P_col=matrix(P_col,nrow=r)

```



```

252     P_row = svd(P_row)$v
253     P_col = svd(P_col)$v
254
255
256
257
258     B=0
259     for(i in 1:n){
260         B=B+alpha[i]*y[i]*P_row%*%t(P_row)%*%X[[i]]+alpha[i]*y[i]*X[[i]]%*%P_col%
261         %t(P_col)
262     }
263 }
264 if(compareobj>obj[iter+1]){
265     P_row_optimum=P_row; P_col_optimum=P_col;
266     obj_optimum=obj;
267     compareobj=obj[iter+1]
268 }
269 }
270
271
272
273 P_row= P_row_optimum; P_col= P_col_optimum;
274 W_row = P_row%*%t(P_row); W_col = P_col%*%t(P_col);
275
276 Dmat=K=matrix(unfold(K_row,c(1,2),c(3,4))@data%*%c(W_row)+unfold(K_col,c(1,2),c
277 (3,4))@data%*%c(W_col),nrow=n,ncol=n)
278
279 dvec = rep(1,n)
280 Dmat = Makepositive((y%*%t(y))*Dmat)
281 Amat = cbind(y,diag(1,n),-diag(1,n))
282 bvec = c(rep(0,1+n),ifelse(y==1,-cost*(1-p),-cost*p))
283 res = solve.QP(Dmat,dvec,Amat,bvec,meq =1)
284 alpha=res$solution
285
286 slope = function(Xnew){
287     newK = rep(0,n)
288     for( i in 1:n){
289         newK[i] = sum(W_row*kernel_row(t(Xnew),t(X[[i]])))+
290             sum(W_col*kernel_col(Xnew,X[[i]]))
291     }
292     return(sum(alpha*y*newK))
293 }
294
295 # intercept part estimation (update b)
296 yfit=K%*(alpha*y) ## faster than lapply
297
298 B=0;
299 for(i in 1:n){
300     B=B+alpha[i]*y[i]*P_row%*%t(P_row)%*%X[[i]]+alpha[i]*y[i]*X[[i]]%*%P_col%*%t(P
301     _col)
302 }
303
304 positive=min(yfit[y==1])
305 negative=max(yfit[y== -1])
306 if ((1-positive)<(-1-negative)) {
307     intercept = -(positive+negative)/2

```

```

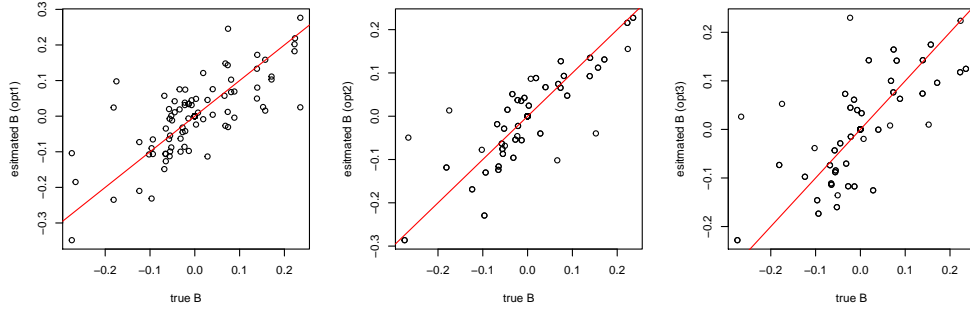
308 }else{
309     gridb0 = seq(from = -1-negative,to = 1-positive,length = 100)
310     intercept = gridb0[which.min(sapply(gridb0,function(b) objective(b,yfit,y,p =
311         p)))]
312 }
313 compareobj = obj[iter+1]
314 predictor = function(Xnew) sign(slope(Xnew)+intercept)
315
316 result$alpha = alpha
317 result$slope = slope; result$predict = predictor
318 result$intercept = intercept;
319 result$P_row = P_row; result$P_col = P_col;
320 result$obj = obj[-1]; result$iter = iter;
321 result$error = error;
322 result$fitted=yfit+intercept; ## add fitted value as a criterium to select cost
323 result$B=B
324 return(result)
325 }
326
327
328
329 #####Combination#####
330
331
332
333 SMMK_sparse = function(X,y,r,kernel_row = c("linear","poly","exp","const"),kernel_
334     col = c("linear","poly","exp","const"),option = c("approximate","exact"), cost
335     = 10, rep = 1, p = .5,sparse=0){
336     result = list()
337
338     option <- match.arg(option)
339
340     if(sparse>0){
341         d1 = nrow(X[[1]]); d2 = ncol(X[[1]]); n = length(X)
342         res = SMMK(X,y,r,kernel_row,kernel_col,cost,rep,p)
343         initB = res$B
344         row_o = order(diag(initB%%t(initB)),decreasing = T)[1:(d1-sparse)]
345         col_o = order(diag(t(initB)%%initB),decreasing = T)[1:(d1-sparse)]
346     }else{
347         row_o = 1:d1; col_o = 1:d2
348     }
349
350     X_sp = lapply(X,function(x) x[row_o,col_o])
351     d1sp = nrow(X_sp[[1]]); d2sp = ncol(X_sp[[1]]); n = length(X_sp)
352     if(option == "exact"){
353         res = SMM(X_sp,y,r,kernel_row,kernel_col,cost,rep,p)
354     }else{
355         res = SMMK(X_sp,y,r,kernel_row,kernel_col,cost,rep,p)
356     }
357
358     B = matrix(0,nrow = d1,ncol = d2)
359     B[row_o,col_o] = res$B
360
361     slope = function(Xnew) res$slope(Xnew[row_o,col_o])
362     intercept = res$intercept

```

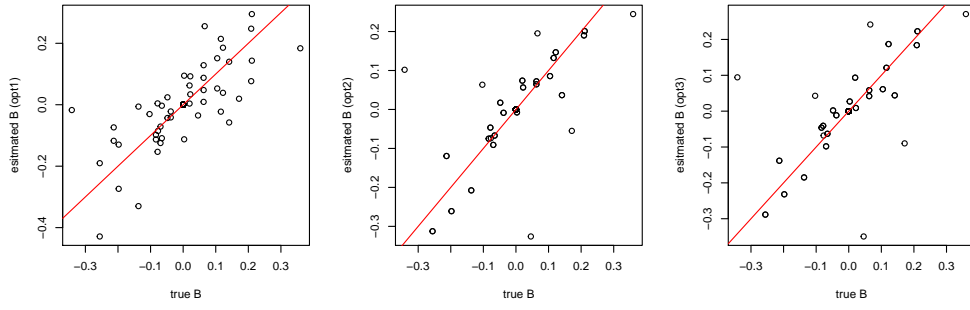
```

364
365 predictor = function(Xnew) sign(slope(Xnew)+intercept)
366
367 result$alpha = res$alpha
368 result$slope = slope; result$predict = predictor
369 result$intercept = intercept;
370 result$P_row = res$P_row; result$P_col = res$P_col;
371 result$obj = res$obj; result$iter = res$iter;
372 result$error = res$error;
373 result$fitted=res$fitted;
374 result$B=B
375 return(result)
376 }

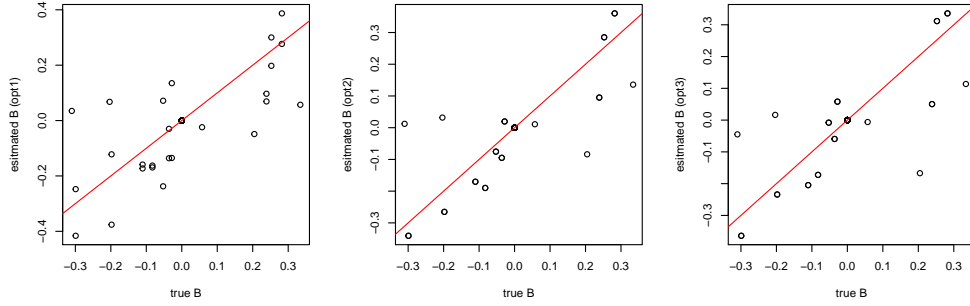
```



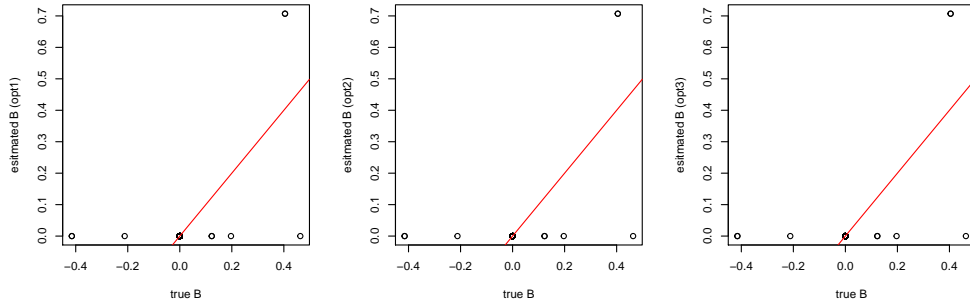
(a) When the number of non zero rows = 9 (sparsity = 1)



(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  $B$  versus estimated coefficient  $\hat{B}$ . 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.