Problem 1

a. Write R codes to solve the OC-SVM problem using the ‘quadprog’ package from R. We will use class 1 of ‘pb2.txt’ as our training set.

Here our algorithm is : 1) read the txt data file by ‘read.delim’ ; 2) partition the dataset into training and test parts (‘trainx’ and ‘testx’); 3) define the gaussian kernel function; 4) compute vectors and matrices required in ‘solve.QP’.

Step1: read the dataset and put it in ‘data’.

> library(quadprog)

> library(psych)

> data=read.delim(file.choose(),header=FALSE, sep="")

Step2: partition the dataset into training and test parts (‘trainx’ and ‘testx’).

> #partition data into training and test parts

> train=data[data[1]==1,]

> trainx=train[,-1]

> trainy=train[,1]

> test=data[data[1]==2,]

> testx=test[,-1]

> testy=test[,1]

> n=dim(trainx)[1] # the total number of training set

Step3: define the kernel function with three inputs (xi and xj are two observations, and sigma) and return the value of inner production of two observations.

> C=0.1 # set the penalty cost factor

> sigma=1 # set the width factor of gaussian kernel

> kern=function(xi,xj,sigma){

+ #three input

+ exp(-(dist(rbind(xi,xj))[1])^2/sigma^2)

+ }

Step4: formulate the Dmat matrix denoted by ‘H\_matrix’, the Amat matrix, and the dvec, bvec (‘b0’) vectors as below.

> #formulate H matrix

> H\_matrix=matrix(0,n,n)

> for (i in 1:n){

+ for (j in 1:n){

+ H\_matrix[i,j]=kern(trainx[i,],trainx[j,],sigma)}

+ }

> #formulate d vector

> dvec=as.vector(array(0,n))

> #formulate A matrix

> Amat=t((rbind(t(trainy),diag(n),-diag(n))))

> #formulate b0 vector

> b0=rbind(1,matrix(0,n),C\*matrix(-1,n))

Input all vectors and matrices required and output the results of alpha as below.

> alpha=solve.QP(H\_matrix,dvec,Amat,bvec=b0)$solution

> sum(alpha)

[1] 1

> alpha

[1] 0.03349411 0.03327106 0.03063813 0.03349411 0.03349411 0.03349411 0.02950052 0.03349411 0.03349411

[10] 0.03304521 0.03349409 0.03349411 0.01980848 0.03171078 0.03131701 0.02950100 0.03349411 0.03340931

[19] 0.03349384 0.03342742 0.03349411 0.03349411 0.03349411 0.03349313 0.03332841 0.03327097 0.02591747

[28] 0.03340998 0.03340978 0.03349411 0.03262404

b. Evaluate the performance of OC-SVM on the class 2 test set.

The procedures are 1) compute the average support vector p; 2) define the prediction function; 3) input the test dataset and obtain the prediction on it.

Step1: compute the average support vector ‘p’ as below.

> #compute the average support vector p

> p=sum(H\_matrix %\*% as.matrix(alpha))/n # use vectorization, not loop

Step2: define the prediction function with four inputs where ‘xnew’ means the new test observation, ‘trainx’ means the training datset, ‘alpha’ means the solution alpha vector and ‘p’ denotes the average support vector computed above.

> #define the prediction function

> pred=function(xnew,trainx,alpha,p){

+ kvec=rep(0,dim(trainx)[1])

+ for(i in 1:dim(trainx)[1]){

+ kvec[i]=kern(xnew,trainx[i,],sigma)

+ }

+ f=sign((t(alpha)%\*%kvec)[1]-p)

+ return(f)

+ }

Step3: input the test dataset and obtain the test accuracy with 100% as below.

> #test the predictive accuracy

> t2=0 # for test accuracy

> for (i in 1:dim(testx)[1]){

+ if(pred(testx[i,],trainx,alpha,p)!=1){

+ t2=t2+1

+ }

+ }

> cat("test accuracy(%):",(t2/dim(testx)[1])\*100)

test accuracy(%): 100

Problem 2

a. Write a R function to obtain the update inverse matrix .

Define a function ‘Hn\_inv’ to updata the inverse of Hn with three inputs where ‘hn\_1\_inv’ means the inverse of Hn-1 at the (n-1)th iteration, ‘data’ means the training dataset, ‘xn’ means a new observation. The function will return the inverse of Hn.

> #define a function to update the inverse of Hn matrix

> Hn\_inv=function(hn\_1\_inv,data,xn){

# three inputs: the inverse of Hn-1,(n-1)iteration data,new sample Xn

+ d=dim(data)[1] # the number of observation

+ delta\_n=matrix(0,d) # column vector delta\_n={k(xn,xi)}

+ for (i in 1:d){

+ delta\_n[i]=k(xn,data[i,],sigma)

+ }

+ delta\_nn=k(xn,xn,sigma) # the scalar delta\_nn by xn itself

+ an=hn\_1\_inv %\*% as.matrix(delta\_n) # (n-1)\*1 vector an

+ rn=delta\_nn+(1/(2\*C))-(t(delta\_n) %\*% an)[1]

+ hn\_inv=(1/rn)\*rbind(cbind(rn\*hn\_1\_inv+an %\*% t(an),-an),cbind(-t(an),1))

+ return(hn\_inv)

+ }

Then, we can test the function above by the following codes:1) read the ‘charlie’ dataset and partition it into training set ‘trainx’ and test set ‘testx’; 2) define the kernel function; 3) initialize the parameters ‘C’ and ‘sigma’ and test the function by computing the inverses of H1~H4.

Step1: read the ‘charlie’ dataset and partition it into training set ‘trainx’ and test set ‘testx’.

> #read the dataset

> charlie=read.csv(file.choose(),header=T)

> data=charlie[,c("Data","x1","x2","x3","x4")]

> #partition data into training and test sets

> train=data[data["Data"]=='Original',]

> trainx=train[,-1] # the training set we will use

> trainy=train[,1]

> test=data[data["Data"]=='New',]

> testx=test[,-1] # the test set we will use

> testy=test[,1]

Step2: define the kernel function.

> # define the kernel function

> k=function(xi,xj,sigma){

+ #three input

+ exp(-(dist(rbind(xi,xj))[1])^2/sigma^2)

+ }

Step3: initialize the parameters ‘C’ and ‘sigma’ and test the function by computing the inverses of H1~H4.

> C=0.01 # the penalty factor

> sigma=10 # the parameter of Gaussian Kernel

>

> h1=k(trainx[1,],trainx[1,],sigma)+1/(2\*C)

> h1\_inv=solve(h1) # the inverse of H1

> #a loop for computing the inverse of Hn

> hn\_1\_inv=h1\_inv # the inverse of H1

> n=4 # assum the number of observation=4

> for (i in 1:n){

+ cat("n=",i,",","inverse","of","H",i,"=","\n")

+ print(hn\_1\_inv)

+ hn\_inv=Hn\_inv(hn\_1\_inv,trainx[1:i,],trainx[i+1,])

+ hn\_1\_inv=hn\_inv

+ }

n= 1 , inverse of H 1 =

[,1]

[1,] 0.01960784

n= 2 , inverse of H 2 =

[,1] [,2]

[1,] 0.0196127355 -0.0003097634

[2,] -0.0003097634 0.0196127355

n= 3 , inverse of H 3 =

[,1] [,2] [,3]

[1,] 0.0196189860 -0.0003034942 -0.0003501955

[2,] -0.0003034942 0.0196190235 -0.0003512452

[3,] -0.0003501955 -0.0003512452 0.0196205795

n= 4 , inverse of H 4 =

[,1] [,2] [,3] [,4]

[1,] 0.0196225563 -0.0002987883 -0.0003456218 -0.0002646959

[2,] -0.0002987883 0.0196252263 -0.0003452167 -0.0003488910

[3,] -0.0003456218 -0.0003452167 0.0196264386 -0.0003390843

[4,] -0.0002646959 -0.0003488910 -0.0003390843 0.0196240091

b. Compare the training times (in sec) between the recursive and direct algorithms.

For the recursive method, we record the runtimes of computing the inverses of H2~H20 through ‘Sys.time()’. We use a ‘for’ loop to compute these inverses of matrices through the defined function ‘Hn\_inv( )’ above. The results are shown as below.

> # iterative method

> hn\_1\_inv=h1\_inv # inverse of H1

> N=19 # assume we have 19 observations

> t1=rep(0,N) # record runtimes

> start\_time1 = Sys.time()

> for (i in 1:N){

+ hn\_inv=Hn\_inv(hn\_1\_inv,trainx[1:i,],trainx[i+1,])

+ hn\_1\_inv=hn\_inv

+ t1[i]=Sys.time()-start\_time1

+ cat("iterative method timing:","H",i+1,",",Sys.time()-start\_time1, "seconds","\n")

+ }

iterative method timing: H 2 , 0.149009 seconds

iterative method timing: H 3 , 0.159009 seconds

iterative method timing: H 4 , 0.17101 seconds

iterative method timing: H 5 , 0.1880112 seconds

iterative method timing: H 6 , 0.2050121 seconds

iterative method timing: H 7 , 0.2260132 seconds

iterative method timing: H 8 , 0.2490141 seconds

iterative method timing: H 9 , 0.2750161 seconds

iterative method timing: H 10 , 0.304018 seconds

iterative method timing: H 11 , 0.33902 seconds

iterative method timing: H 12 , 0.374022 seconds

iterative method timing: H 13 , 0.4110241 seconds

iterative method timing: H 14 , 0.4480262 seconds

iterative method timing: H 15 , 0.4900281 seconds

iterative method timing: H 16 , 0.5350311 seconds

iterative method timing: H 17 , 0.5820341 seconds

iterative method timing: H 18 , 0.6340361 seconds

iterative method timing: H 19 , 0.6960402 seconds

iterative method timing: H 20 , 0.7540431 seconds

Similarly, for the direct method, we define a function ‘Hn’ to obtain Hn and use ‘inv( )’ to obtain its inverse. Then, we also use a ‘for’ loop to compute the inverses of H2~H20 and record the runtimes as below.

Define the function ‘Hn’ with only an input (‘data’ means observations) and return the Hn matrix.

> # direct approach

> #define a function of computing Hn directly

> Hn=function(data){

+ d=dim(data)[1] # the number of observation

+ Km=matrix(0,d,d) # K matrix named as Km

+ I=diag(d) # d dimensions identity matrix

+ for (i in 1:d){

+ for (j in 1:d) {

+ Km[i,j]=k(data[i,],data[j,],sigma)}

+ }

+ hn=Km+(1/(2\*C))\*I # compute the matrix Hn

+ return (hn)

+ }

Use a ‘for’ loop to compute the inverses of H2~H20 and record the runtimes as below.

> N=19 # assume we have 19 observations

> t2=rep(0,N) # record runtimes for direct method

> start\_time2 = Sys.time()

> for (i in 1:N){

+ hn\_inv=inv(Hn(trainx[1:(i+1),]))

+ t2[i]=Sys.time()-start\_time2

+ cat("direct approach timing:","H",i+1,",",Sys.time()-start\_time2,"seconds","\n")

+ }

direct approach timing: H 2 , 0.194011 seconds

direct approach timing: H 3 , 0.248014 seconds

direct approach timing: H 4 , 0.3040168 seconds

direct approach timing: H 5 , 0.3860219 seconds

direct approach timing: H 6 , 0.5030289 seconds

direct approach timing: H 7 , 0.660038 seconds

direct approach timing: H 8 , 0.8550489 seconds

direct approach timing: H 9 , 1.091062 seconds

direct approach timing: H 10 , 1.432082 seconds

direct approach timing: H 11 , 1.787102 seconds

direct approach timing: H 12 , 2.183125 seconds

direct approach timing: H 13 , 2.63115 seconds

direct approach timing: H 14 , 3.160181 seconds

direct approach timing: H 15 , 3.718212 seconds

direct approach timing: H 16 , 4.305246 seconds

direct approach timing: H 17 , 5.001286 seconds

direct approach timing: H 18 , 5.608321 seconds

direct approach timing: H 19 , 6.325362 seconds

direct approach timing: H 20 , 7.099406 seconds

Compare and plot the runtimes of two methods as below. We can observe that the recursive (iterative) method works faster than the direct method all the time.

> x <- rep(1:N) # the number of iterations

> plot(x,t2,type="l",col="blue",xlab="the number of iterations",

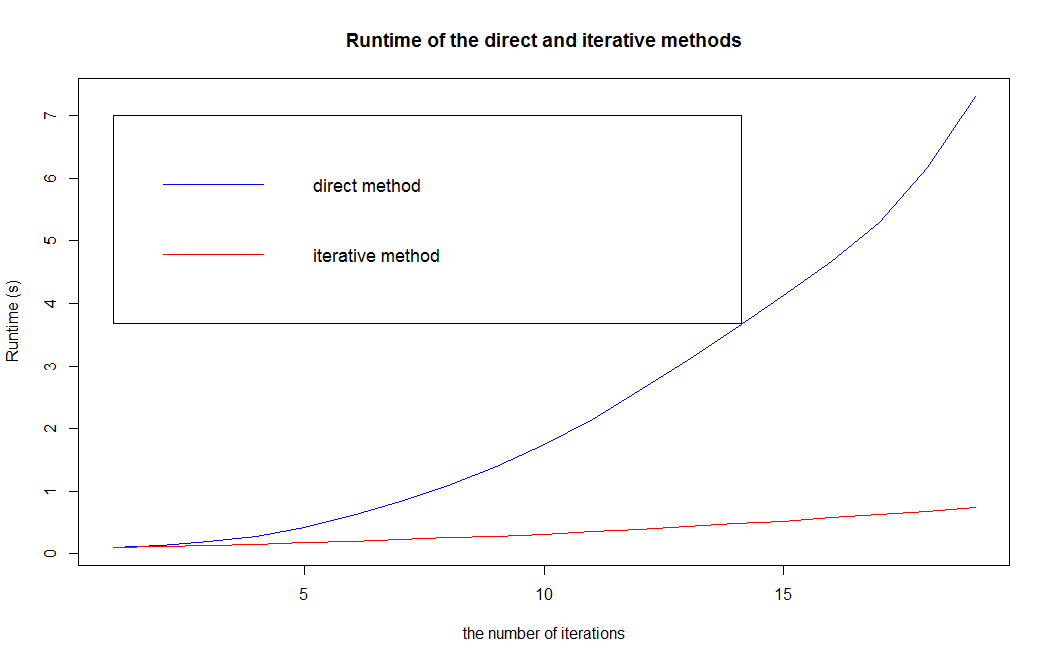
+ ylab="Runtime (s)", main="Runtime of the direct and iterative methods")

> #Add more data to the plot

> lines(x,t1,col="red") #add runtimes of iterative method

> legend(x=1,y=7,c("direct method","iterative method"),cex=1.1,

+ col=c("blue","red"),lty=c(1,1)) #add legend



c. Write a R function to obtain the updated solution

We define the function alpha1 (hn\_inv, data) with two inputs ( and n observations) and a return a value (the vector shown as ‘alpha’). Note that ‘kj’ means the vector, ‘e’ denotes vector.

> #define a function to compute alpha1 vector

> alpha1=function(hn\_inv,data){

+ # two inputs: the inverse of Hn and n observations

+ d=dim(data)[1] # the number of observation

+ e=rep(1,d) # n\*1 column vector

+ kj=rep(0,d) # n\*1, kn vector

+ for(i in 1:d){

+ kj[i]=k(data[i,],data[i,],sigma)

+ }

+ p1=2-t(e) %\*% hn\_inv %\*% kj

+ p2=t(e) %\*% hn\_inv %\*% e

+ alpha=0.5\*hn\_inv %\*% (kj + (p1[1]/p2[1])\*e)

+ return (alpha)

+ }

After defining the function alpha1 (hn\_inv, data), we can test it by inputting shown by ‘hn\_1\_inv’ at first and compute the vector named as ‘alpha\_n’ in a loop as below.

> hn\_1\_inv=h1\_inv # the inverse of Hn-1 starts from H1

> N=4 # assume we have 4 observations

> for (i in 1:N){

+ alpha\_n=alpha1(hn\_1\_inv,trainx[1:i,])

+ hn\_inv=Hn\_inv(hn\_1\_inv,trainx[1:i,],trainx[i+1,])

+ hn\_1\_inv=hn\_inv

+ cat("n=",i,",","alpha=","\n",alpha\_n,"\n")

+ }

n= 1 , alpha=

1

n= 2 , alpha=

0.5 0.5

n= 3 , alpha=

0.3336099 0.3335921 0.332798

n= 4 , alpha=

0.2508047 0.2497175 0.2492375 0.2502403

d. Apply your function to the set obtained with the first row and add sequentially the next 6 rows. Check that your code work correctly by comparing the ***α*** obtained using the recursive updates (i.e. ***α***2, ***α***3, ***α***4, ***α***5, ***α***6, ***α***7) to the actual ***α***’s using the direct approach (***α***2, ***α***3, ***α***4, ***α***5, ***α***6, ***α***7).

For the iterative method, we use a loop to compute both ‘alpha\_n’ and ‘hn\_inv’ in the iterative fashion. Assume we compute α1~ α7 shown as below.

> # iterative method

> hn\_1\_inv=h1\_inv # inverse of H1

> N=7 # add 7 observations sequentially

> for (i in 1:N) {

+ alpha\_n=alpha1(hn\_1\_inv,trainx[1:i,])

+ hn\_inv=Hn\_inv(hn\_1\_inv,trainx[1:i,],trainx[i+1,])

+ hn\_1\_inv=hn\_inv

+ cat("iterative method:n=",i,",","alpha=","\n",alpha\_n,"\n")

+ }

iterative method:n= 1 , alpha=

1

iterative method:n= 2 , alpha=

0.5 0.5

iterative method:n= 3 , alpha=

0.3336099 0.3335921 0.332798

iterative method:n= 4 , alpha=

0.2508047 0.2497175 0.2492375 0.2502403

iterative method:n= 5 , alpha=

0.2011741 0.1994352 0.1991807 0.1994354 0.2007746

iterative method:n= 6 , alpha=

0.1659874 0.1656548 0.1650548 0.1660808 0.1674513 0.1697709

iterative method:n= 7 , alpha=

0.1423895 0.1419132 0.1412181 0.1422339 0.1438585 0.1464801 0.1419066

Similarly, for the direct method, we use ‘Hn’ and ‘alpha1’ functions to compute ***α***2~ ***α***7 directly through a ‘for’ loop as below.

> # direct approach

> hn\_inv=h1\_inv # the inverse of Hn-1 starts from the inverse of H1

> N=7 # assume we have 7 observations

> for (i in 1:N){

+ alpha\_n=alpha1(hn\_inv,trainx[1:i,])

+ hn\_inv=inv(Hn(trainx[1:(i+1),]))

+ cat("direct method:n=",i,",","alpha=","\n",alpha\_n,"\n")

+ }

direct method:n= 1 , alpha=

1

direct method:n= 2 , alpha=

0.5 0.5

direct method:n= 3 , alpha=

0.33361 0.3335921 0.3327979

direct method:n= 4 , alpha=

0.2508047 0.2497175 0.2492375 0.2502403

direct method:n= 5 , alpha=

0.2011742 0.1994352 0.1991806 0.1994354 0.2007747

direct method:n= 6 , alpha=

0.1659872 0.1656548 0.1650547 0.1660809 0.1674514 0.1697709

direct method:n= 7 , alpha=

0.1423895 0.1419133 0.1412182 0.1422338 0.1438586 0.1464801 0.1419065

Therefore, we can see the results of the vector of both the iterative and direct methods are same.

e. Use your results from (c) to sequentially update the radius and *dz*. Check sequentially if the next 7 rows are targets or outliers by comparing *d***x***n* to . If an observation is an outlier, remove it from the training set. If it is a target, add it to the training set.

Our steps can be : (1) define the function R2(alpha\_n, train) with two inputs (the vector and the training dataset) and an output (the radius ) named as ‘r2’; (2) define the function DZ (alpha\_n,train,test) with three inputs (the vector, the training dataset and a new test vector) and an output dz value named as ‘dz’; (3) use a loop to check and .

Step (1) defines the function R2(alpha\_n, train) with two inputs (the vector and the training dataset) and an output (the radius ) named as ‘r2’.

> # compute R2 from the training dataset

> R2=function(alpha\_n,train){

+ # R2 consists of three average parts:ap1,ap2,ap3

+ d=dim(train)[1]

+ ap1=0

+ ap2=0

+ ap3=0

+ for (i in 1:d){

+ ap1=ap1+k(train[i,],train[i,],sigma) }

+

+ for (i in 1:d){

+ for (j in 1:d){

+ ap2=ap2+2\*alpha\_n[j]\*k(train[i,],train[j,],sigma) }

+ }

+ for( i in 1:d){

+ for (j in 1:d){

+ ap3=ap3+alpha\_n[i]\*alpha\_n[j]\*k(train[i,],train[j,],sigma) }

+ }

+ r2=(ap1-ap2)/d+ap3

+ return(r2)

+ }

Step (2) defines the function DZ (alpha\_n,train,test) with three inputs (the vector, the training dataset and a new test vector) and an output dz value named as ‘dz’.

> #compute dz of a test dataset

> DZ=function(alpha\_n,train,test){

+ # assume dz consists of three parts:p1,p2,p3

+ d=dim(train)[1]

+ p1=k(test,test,sigma)

+ p2=0

+ p3=0

+ for (i in 1:d){

+ p2=p2+2\*alpha\_n[i]\*k(test,train[i,],sigma)}

+ for (i in 1:d){

+ for(j in 1:d){

+ p3=p3+alpha\_n[i]\*alpha\_n[j]\*k(train[i,],train[j,],sigma) }

+ }

+ dz=p1-p2+p3

+ return(dz)

+ }

Step (3) uses a loop to check and in an iterative fashion. Assume we have a first set with 7 observations and the next new 7 observations.

For the first set, we sequentially update and dz as below.

> C=0.01

> sigma=10

> tn1=7

> hn\_1\_inv=h1\_inv # the inverse of Hn-1 starts from the inverse of H1

> for (i in 1:tn1){

+ alpha\_n=alpha1(hn\_1\_inv,trainx[1:i,])

+ r2=R2(alpha\_n,trainx[1:i,])

+ dz=DZ(alpha\_n,trainx[1:i,],trainx[(i+1),])

+ hn\_inv=Hn\_inv(hn\_1\_inv,trainx[1:i,],trainx[(i+1),])

+ hn\_1\_inv=hn\_inv

+ cat("n=",i,"\n","R2=",r2,"\n","dz=",dz,"\n")

+ }

n= 1

R2= 0

dz= 0.3890128

n= 2

R2= 0.09725319

dz= 0.05068737

n= 3

R2= 0.07609933

dz= 0.2159493

n= 4

R2= 0.0975555

dz= 0.3241374

n= 5

R2= 0.1298884

dz= 0.7589654

n= 6

R2= 0.2137611

dz= 0.1429457

n= 7

R2= 0.2005882

dz= 0.3080146

For the next 7 new rows, we check them iteratively as below.

> #check the next 7 rows

> tn2=14

> hn\_1\_inv=h1\_inv # the inverse of Hn-1 starts H1

> for (i in 1:tn2){

+ alpha\_n=alpha1(hn\_1\_inv,trainx[1:i,]) # iterative alpha\_n

+ r2=R2(alpha\_n,trainx[1:i,]) # iterative R2 (n)

+ dz=DZ(alpha\_n,trainx[1:i,],trainx[1:(i+1),]) # dz of test vector xn

+ hn\_inv=Hn\_inv(hn\_1\_inv,trainx[1:i,],trainx[(i+1),])

+ hn\_1\_inv=hn\_inv

+ if (i>6){ #predict the next 7 rows

+ cat("n=",i,"\n","R2=",r2,"\n","dz=",dz,"\n")

+ if (dz<=r2){

+ print("It's a target")}

+ else {

+ print("It's an outlier")}

+ }

+ }

n= 7

R2= 0.2005882

dz= -0.00800355

[1] "It's a target"

n= 8

R2= 0.2092869

dz= -0.0165703

[1] "It's a target"

n= 9

R2= 0.189696

dz= 0.002581427

[1] "It's a target"

n= 10

R2= 0.1847253

dz= 0.007651512

[1] "It's a target"

n= 11

R2= 0.1726818

dz= 0.01941209

[1] "It's a target"

n= 12

R2= 0.1857478

dz= 0.00638475

[1] "It's a target"

n= 13

R2= 0.1918417

dz= 0.0003574698

[1] "It's a target"

n= 14

R2= 0.1820956

dz= 0.009911494

[1] "It's a target"

Problem 3

a. Starting with the first row, add sequentially the next 9 rows and use your updating codes from problem 2 to obtain **.**

Here we use the function ‘Hn\_inv’ we define in problem 2 to compute iteratively as below.

> # iterative method

> h1=k(trainx[1,],trainx[1,],sigma)+1/(2\*C)

> h1\_inv=solve(h1) # the inverse of H1

> hn\_1\_inv=h1\_inv # the inverse of Hn-1 starts from H1

> N=9 # add 9 observations sequentially

> for (i in 1:N){

+ hn\_inv=Hn\_inv(hn\_1\_inv,trainx[1:i,],trainx[i+1,])

+ hn\_1\_inv=hn\_inv

+ }

> cat("iterative method:","inverse of H10=","\n")

> print(hn\_inv)

iterative method: inverse of H10=

[,1] [,2] [,3] [,4] [,5] [,6] [,7]

[1,] 9.146083e-01 4.463202e-07 -3.130743e-04 2.020667e-08 -1.328664e-11 -2.901629e-07 1.293478e-05

[2,] 4.463202e-07 9.091267e-01 1.512084e-03 -8.582908e-04 5.612921e-07 -2.902188e-13 2.050644e-05

[3,] -3.130743e-04 1.512084e-03 1.019688e+00 -6.245380e-05 4.105073e-08 9.596104e-11 -3.221944e-02

[4,] 2.020667e-08 -8.582908e-04 -6.245380e-05 9.090922e-01 -5.987784e-04 -6.368099e-15 -2.487222e-04

[5,] -1.328664e-11 5.612921e-07 4.105073e-08 -5.987784e-04 9.090913e-01 4.188040e-18 1.638230e-07

[6,] -2.901629e-07 -2.902188e-13 9.596104e-11 -6.368099e-15 4.188040e-18 9.090909e-01 -4.474992e-12

[7,] 1.293478e-05 2.050644e-05 -3.221944e-02 -2.487222e-04 1.638230e-07 -4.474992e-12 9.104777e-01

[8,] 1.644303e-05 -9.305774e-08 2.069029e-07 8.259175e-11 -5.400108e-14 3.751424e-11 4.976556e-09

[9,] -4.929922e-05 -5.954581e-03 -3.336280e-01 1.427357e-05 -9.352936e-09 4.068011e-11 -7.770354e-03

[10,] -7.103591e-02 7.013177e-07 4.096929e-05 -1.780085e-09 1.166629e-12 -1.610427e-07 8.344960e-07

[,8] [,9] [,10]

[1,] 1.644303e-05 -4.929922e-05 -7.103591e-02

[2,] -9.305774e-08 -5.954581e-03 7.013177e-07

[3,] 2.069029e-07 -3.336280e-01 4.096929e-05

[4,] 8.259175e-11 1.427357e-05 -1.780085e-09

[5,] -5.400108e-14 -9.352936e-09 1.166629e-12

[6,] 3.751424e-11 4.068011e-11 -1.610427e-07

[7,] 4.976556e-09 -7.770354e-03 8.344960e-07

[8,] 9.090910e-01 -6.399420e-07 -2.129178e-04

[9,] -6.399420e-07 1.018651e+00 -1.201689e-04

[10,] -2.129178e-04 -1.201689e-04 9.146082e-01

b. Next, at each future step, we add a new observation and discard the oldest observation. This is sometimes called the “Add and Forget Algorithm”. Write an R function to update for the “Add and Forget Algorithm”, i.e. after adding a new observation, using eq(7), and removing an old observation, using eq(9).

For Add and Forget Algorithm, we define a function ‘U\_inv’ to implement this method. This function has three inputs (‘h10\_inv’ means the the inverse of H10, ‘traindata’ means the training set, and ‘newdata’ means the next sequential dataset) and a return value of ‘U10\_inv’. In the function, we use a ‘for’ loop to simulate the process of adding a new sample and removing an oldest sample. When adding a new sample, we use ‘Hn\_inv’ function in eq (7) to get the inverse of H11. When removing an oldest sample, we use ‘U10\_inv’ in eq (9) in the loop to obtain the inverse of U10.

> #problem 3b: add a new sample and remove an oldest sample

> #define a function to update the inverse of U10 in eq(9)

> U\_inv=function(h10\_inv,traindata,newdata){

+ d=dim(newdata)[1]

+ for (i in 1:d){

+ h11\_inv=Hn\_inv(h10\_inv,traindata,newdata[i,])

+ row1=h11\_inv[1,] # first row of h11\_inv

+ row10=h11\_inv[-1,] # last ten rows of h11\_inv

+ e=row1[1] # the constant e in block of Hn

+ ft=row1[-1] # the transpose of f vector

+ f=row10[,1] # f vector

+ D=row10[,-1] # D matrix

+ U10\_inv=D-(1/e)\*f%\*%t(ft) # inverse of U10 in eq(9)

+ # remove the oldest one and add a new one

+ traindata=rbind(traindata[-1,],newdata[i,])

+ h10\_inv=U10\_inv

+ }

+ return(U10\_inv)

+ }

Then, we test the function ‘U\_inv’ above to compute ‘U10\_inv’ as below.

> h10\_inv=hn\_inv # the inverse of H10 at first time

> U10\_inv=U\_inv(h10\_inv,trainx[1:10,],trainx[11:20,])

> cat("the inverse of U10=","\n")

> print(U10\_inv)

the inverse of U10=

[,1] [,2] [,3] [,4] [,5] [,6] [,7]

[1,] 9.093492e-01 -1.127443e-07 -8.306584e-08 2.710858e-04 -2.354682e-03 2.278697e-09 -1.106719e-08

[2,] -1.127443e-07 9.094687e-01 7.319440e-10 -3.782697e-04 -7.062223e-10 -1.849592e-02 -1.147443e-03

[3,] -8.306584e-08 7.319440e-10 9.090921e-01 -1.929057e-06 1.445796e-10 -2.397760e-11 7.949126e-11

[4,] 2.710858e-04 -3.782697e-04 -1.929057e-06 9.094828e-01 1.698073e-06 7.645285e-06 -3.743666e-05

[5,] -2.354682e-03 -7.062223e-10 1.445796e-10 1.698073e-06 9.090970e-01 1.427360e-11 -7.047845e-11

[6,] 2.278697e-09 -1.849592e-02 -2.397760e-11 7.645285e-06 1.427360e-11 9.094672e-01 2.329628e-05

[7,] -1.106719e-08 -1.147443e-03 7.949126e-11 -3.743666e-05 -7.047845e-11 2.329628e-05 9.090924e-01

[8,] 3.565563e-06 1.507151e-07 -1.058799e-03 -3.629794e-04 4.821049e-08 -3.061583e-09 1.494370e-08

[9,] -1.514645e-02 7.849817e-06 4.564483e-07 -1.887423e-02 -1.041222e-04 -1.586542e-07 7.714222e-07

[10,] -3.463097e-14 3.295126e-07 3.967663e-15 -1.161799e-10 -2.168808e-16 -3.705204e-04 -3.565604e-08

[,8] [,9] [,10]

[1,] 3.565563e-06 -1.514645e-02 -3.463097e-14

[2,] 1.507151e-07 7.849817e-06 3.295126e-07

[3,] -1.058799e-03 4.564483e-07 3.967663e-15

[4,] -3.629794e-04 -1.887423e-02 -1.161799e-10

[5,] 4.821049e-08 -1.041222e-04 -2.168808e-16

[6,] -3.061583e-09 -1.586542e-07 -3.705204e-04

[7,] 1.494370e-08 7.714222e-07 -3.565604e-08

[8,] 9.090924e-01 -3.630425e-04 5.258587e-14

[9,] -3.630425e-04 9.097349e-01 2.411148e-12

[10,] 5.258587e-14 2.411148e-12 9.090911e-01

c. Compare the “Training accuracy (%)”, “Testing accuracy (%)” and “Training Time (s)’ between the recursive and direct algorithms. Note that the direct approach will be containing 11, 12, 13, ..., 20 observations at each iteration while the updating algorithm will only have 10 observations at each iteration.

For the recursive method, we define a function ‘ite\_runtime’ based on the ‘U\_inv’ function to record the iterative runtimes ‘runtime’. The function ‘ite\_runtime’ contains the same three inputs of ‘U\_inv’ and almost same procedures but has one more variables ‘runtime’ in the loop.

> # iterative method

> ite\_runtime=function(h10\_inv,traindata,newdata){

+ d=dim(newdata)[1]

+ runtime=rep(0,d)

+ start\_time=proc.time() # recording runtime starts

+ for (i in 1:d){

+ h11\_inv=Hn\_inv(h10\_inv,traindata,newdata[i,])

+ row1=h11\_inv[1,] # first row of h11\_inv

+ row10=h11\_inv[-1,] # last ten rows of h11\_inv

+ e=row1[1] # the constant e in block of Hn

+ ft=row1[-1] # the transpose of f vector

+ f=row10[,1] # f vector

+ D=row10[,-1] # D matrix

+ U10\_inv=D-(1/e)\*f%\*%t(ft) # inverse of U10 in eq(9)

+ # remove the oldest one and add a new one

+ traindata=rbind(traindata[-1,],newdata[i,])

+ h10\_inv=U10\_inv

+ runtime[i]=proc.time()-start\_time

+ cat(i,"iterations","runtime=",runtime[i],"seconds","\n")

+ }

+ return(runtime)

+ }

> h10\_inv=hn\_inv # the inverse of H10 at first time

> runtime1=ite\_runtime(h10\_inv,trainx[1:10,],trainx[11:20,])

1 iterations runtime= 0.03 seconds

2 iterations runtime= 0.11 seconds

3 iterations runtime= 0.14 seconds

4 iterations runtime= 0.19 seconds

5 iterations runtime= 0.22 seconds

6 iterations runtime= 0.25 seconds

7 iterations runtime= 0.28 seconds

8 iterations runtime= 0.3 seconds

9 iterations runtime= 0.33 seconds

10 iterations runtime= 0.36 seconds

Similarly, for the direct method, we use ‘Hn’ function to compute each new inverse of Hn directly through a ‘for’ loop as below.

> #direct method

> N=10 # add 10 observations sequentially

> runtime2=rep(0,N) # record the runtime for direct method

> start\_time2 = proc.time()

> for (i in 1:N){

+ hn\_inv=solve(Hn(trainx[1:(i+10),]))

+ runtime2[i]=proc.time()-start\_time2

+ cat(i,"iterations","runtime=",runtime2[i],"seconds","\n")

+ }

1 iterations runtime= 0.45 seconds

2 iterations runtime= 0.87 seconds

3 iterations runtime= 1.34 seconds

4 iterations runtime= 1.87 seconds

5 iterations runtime= 2.51 seconds

6 iterations runtime= 3.23 seconds

7 iterations runtime= 4.04 seconds

8 iterations runtime= 4.91 seconds

9 iterations runtime= 5.91 seconds

10 iterations runtime= 7.03 seconds

Finally, we compare and plot the results of runtimes in both methods as below.

> #plot the runtimes of two methods

> x <- rep(1:N)

> plot(x,runtime2,type="l",col="blue",xlab="the number of iterations",

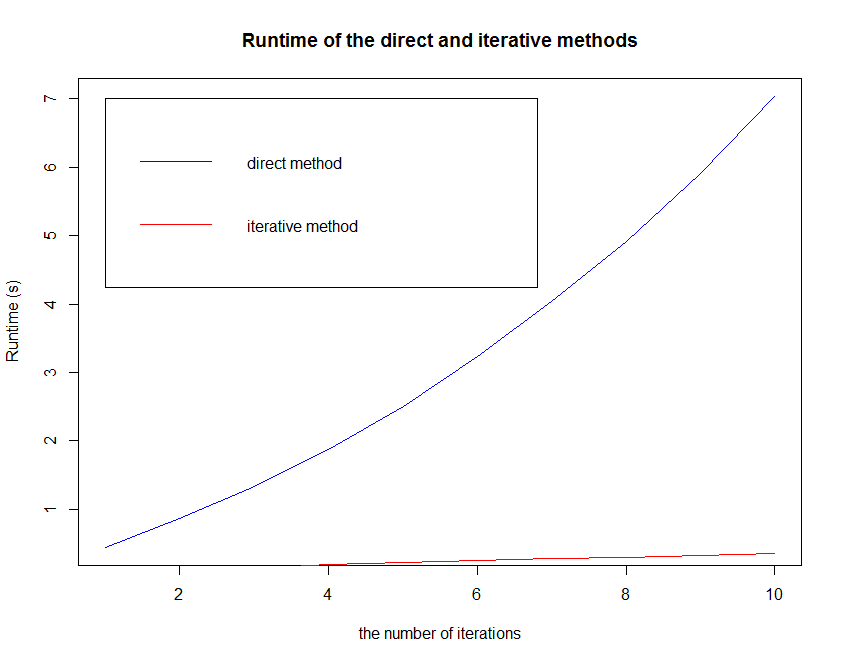
+ ylab="Runtime (s)", main="Runtime of the direct and iterative methods")

> #Add more data to the plot

> lines(x,runtime1,col="red") #add runtimes of iterative method

> legend(x=1,y=7,c("direct method","iterative method"),cex=1.,

+ col=c("blue","red"),lty=c(1,1)) #add legend



For the training and test accuracy in iterative method, we also can get the following results by a ‘for’ loop. Note that the final training set here may be the updated ten samples ‘trainx[11:20,]’.

> #training accuracy and test accuracy in iterative method

> hn\_inv=U\_inv(h10\_inv,trainx[1:10,],trainx[11:20,])#the inverse of H10

> alpha\_n=alpha1(hn\_inv,trainx[11:20,]) # iterative alpha\_n

> ap3=ap(alpha\_n,trainx[11:20,]) # the constant term in R2 and dz

> r2=R2(alpha\_n,trainx[11:20,],ap3) # trained radius R2

>

> #compute the training accuracy in iterative method

> ta=0 # set the counter of prediction

> for (i in 1:dim(trainx[11:20,])[1]){

+ dz=DZ(alpha\_n,trainx[11:20,],trainx[11:20,][i,],ap3)

+ if (dz<=r2){

+ ta=ta+1

+ cat("n=",i,"It's a target","\n")}

+ else{

+ cat("n=",i,"It's an outlier","\n")}

+ }

n= 1 It's a target

n= 2 It's a target

n= 3 It's an outlier

n= 4 It's a target

n= 5 It's an outlier

n= 6 It's a target

n= 7 It's an outlier

n= 8 It's an outlier

n= 9 It's a target

n= 10 It's an outlier

> train\_accuracy=ta/dim(trainx[11:20,])[1]

> cat(" Training accuracy(%):",train\_accuracy\*100,"\n")

Training accuracy(%): 50

Similarly, we can get the test accuracy in iterative method as below through a ‘for’ loop.

> # compute the test accuracy in iterative method

> tt=0 # set the counter of prediction

> for (i in 1:dim(testx)[1]){

+ dz=DZ(alpha\_n,trainx[11:20,],testx[i,],ap3)

+ if (dz<=r2){

+ tt=tt+1

+ cat("n=",i,"It's a target","\n")}

+ else{

+ cat("n=",i,"It's an outlier","\n")}

+ }

n= 1 It's an outlier

n= 2 It's an outlier

n= 3 It's an outlier

n= 4 It's an outlier

n= 5 It's an outlier

n= 6 It's an outlier

n= 7 It's an outlier

n= 8 It's an outlier

n= 9 It's an outlier

n= 10 It's an outlier

> test\_accuracy=(dim(testx)[1]-tt)/dim(testx)[1]

> cat("Testing accuracy(%):",test\_accuracy\*100,"\n")

Testing accuracy(%): 100

For the direct method, we also use the similar procedures to compute its training and test accuracy as below. Note that here we use the full training set ‘trainx’.

> #compute the training accuracy in direct method

>

> alpha\_n=alpha1(hn\_inv,trainx) # iterative alpha\_n

> ap3=ap(alpha\_n,trainx) # the constant term in R2 and dz

> r2=R2(alpha\_n,trainx,ap3) # trained radius R2

> ta=0 # set the counter of prediction

> for (i in 1:dim(trainx)[1]){

+ dz=DZ(alpha\_n,trainx,trainx[i,],ap3)

+ if (dz<=r2){

+ ta=ta+1

+ cat("n=",i,"It's a target","\n")}

+ else{

+ cat("n=",i,"It's an outlier","\n")}

+ }

n= 1 It's a target

n= 2 It's an outlier

n= 3 It's a target

n= 4 It's an outlier

n= 5 It's an outlier

n= 6 It's an outlier

n= 7 It's a target

n= 8 It's an outlier

n= 9 It's a target

n= 10 It's a target

n= 11 It's a target

n= 12 It's an outlier

n= 13 It's an outlier

n= 14 It's a target

n= 15 It's a target

n= 16 It's an outlier

n= 17 It's an outlier

n= 18 It's an outlier

n= 19 It's a target

n= 20 It's an outlier

> train\_accuracy=ta/dim(trainx)[1]

> cat(" Training accuracy(%):",train\_accuracy\*100,"\n")

Training accuracy(%): 45

Similarly, we can get the test accuracy of direct method as below.

> # compute the test accuracy in direct method

> tt=0 # set the counter of prediction

> for (i in 1:dim(testx)[1]){

+ dz=DZ(alpha\_n,trainx,testx[i,],ap3)

+ if (dz<=r2){

+ tt=tt+1

+ cat("n=",i,"It's a target","\n")}

+ else{

+ cat("n=",i,"It's an outlier","\n")}

+ }

n= 1 It's an outlier

n= 2 It's an outlier

n= 3 It's an outlier

n= 4 It's an outlier

n= 5 It's an outlier

n= 6 It's an outlier

n= 7 It's an outlier

n= 8 It's an outlier

n= 9 It's an outlier

n= 10 It's an outlier

> test\_accuracy=(dim(testx)[1]-tt)/dim(testx)[1]

> cat("Testing accuracy(%):",test\_accuracy\*100,"\n")

Testing accuracy(%): 100

Problem 4

a. Write an R function to obtain the block update inverse matrix .

The main procedures are : (1) read the Group I data file and extract the target independent variables (x1~x9); (2) define the Gaussian kernel k (xi, xj, sigma) with three inputs (i-th and j-th observations, the width of the Gaussian kernel); (3) define the function HNK\_inv2 (hn\_inv, data, newdata) to update inverse matrix , where the three inputs of this function denote , the N observations and the K new observation.

Step (1) reads the data file by pandas, kept in ‘group1’ firstly. Then, split string lines into multiple columns including (x1~x9) and the response variable (positive or negative), kept in ‘group1\_data’. Next, we transform categorical levels (‘x’, ‘o’, ‘b’) into continuous ones (1, 2, 3), kept in ‘group1\_data’. Also, we separate the dataset into the ‘positive’ part in ‘data\_pos’ and the ‘negative’ part in ‘data\_neg’. Eventually, we extract the target independent variables (x1~x9) and convert them into ‘int’ type, kept in ‘data\_pos\_x’ and ‘data\_neg\_x’.

> #read the datafile

> group1=read.delim(file.choose(),header=FALSE, sep=",")

> group1\_data=matrix(0,dim(group1)[1],dim(group1)[2])

Transform categorical levels (‘x’, ‘o’, ‘b’) into continuous ones (3, 2, 1) as below (in later question, we will normalize all variables into (-1,+1)). And separate ‘group1\_data’ into positive and negative parts, ‘data\_pos’ and ‘data\_neg’ .

> # transform categorical variables into dummy variables

> for (i in 1:dim(group1\_data)[1]){

+ for (j in 1:dim(group1\_data)[2]){

+ if (group1[i,j]=='x'){

+ group1\_data[i,j]=as.numeric(group1[i,j])} #default level x=3,

+ if (group1[i,j]=='o'){

+ group1\_data[i,j]=as.numeric(group1[i,j])} #default level o=2

+ if (group1[i,j]=='b'){

+ group1\_data[i,j]=as.numeric(group1[i,j])} #default level b=1

+ else {group1\_data[i,j]=as.numeric(group1[i,j])} #default positive=2,negative=1

+ }

+ }

> # separate into positive and negative datasets

> data\_pos=group1\_data[group1\_data[,10]==2,] # 626,default positive=2

> data\_neg=group1\_data[group1\_data[,10]==1,] # 332,default negative=1

we extract the target independent variables (x1~x9), kept in ‘data\_pos\_x’ and ‘data\_neg\_x’.

> data\_pos\_x=data\_pos[,-10] #only contain x in class positive

> data\_neg\_x=data\_neg[,-10] #only contain x in class negative

> data\_pos\_x=as.data.frame(data\_pos\_x) # dataframe type

> data\_neg\_x=as.data.frame(data\_neg\_x)

Step (2) defines the Gaussian kernel k (xi, xj, sigma) with three inputs (i-th and j-th observations, the width of the Gaussian kernel).

> # kernel function

> k=function(xi,xj,sigma){

+ #three input

+ exp(-(dist(rbind(xi,xj))[1])^2/sigma^2)

+ }

step (3) defines the function HNK\_inv2 (hn\_inv, data, newdata) to update inverse matrix , where the three inputs of this function denote , the N observations and the K new observations. Note that andmatriceare named as ‘B’ and ‘D’; ‘w1~w4’ are four entries in . The return value in function HNK\_inv2 is shown as ‘hnk\_inv’.

> #define a function to update the block inverse matrix

> HNK\_inv2=function(hn\_inv,data,newdata){

+ # input Hn, n observations and new K observations

+ N=dim(as.matrix(data))[1]

+ K=dim(as.matrix(newdata))[1]

+ B=matrix(0,K,N) # matrix B: K\*N

+ D=matrix(0,K,K) # matrix D: K\*K

+ I=diag(K) #identity matrix

+

+ for (i in 1:K){

+ for (j in 1:N){

+ B[i,j]=k(newdata[i,],data[j,],sigma)}

+ }

+ for (i in 1:K){

+ for (j in 1:K){

+ D[i,j]=k(newdata[i,],newdata[j,],sigma)}

+ }

+ D=D+(1/(2\*C))\*I

+ hn\_inv=as.matrix(hn\_inv)

+ w1=hn\_inv-hn\_inv%\*%t(B)%\*%solve(-D+B %\*% hn\_inv %\*% t(B)) %\*% B %\*% hn\_inv

+ w2=hn\_inv %\*% t(B)%\*%solve(B%\*%hn\_inv%\*%t(B)-D)

+ w3=solve(B%\*%hn\_inv%\*%t(B)-D)%\*% B %\*% hn\_inv

+ w4=solve(D-B%\*%hn\_inv%\*%t(B))

+ hnk\_inv=rbind(cbind(w1,w2),cbind(w3,w4))

+ return(hnk\_inv)

+ }

b. Check that the results match the ones using the direct approach.

For the iterative block method, we can test HNK\_inv2 function and assume here K=1, C=5, sigma=1. Then we can start from named as ‘h1\_inv’, then use a loop to compute . Here we only display the part of results from n=1 to n=5.

> #if K=1, the results should be same as the inverse of Hn in problem 2

> C=5 # the penalty factor

> sigma=1 # the parameter of Gaussian Kernel

> h1=k(data\_pos\_x[1,],data\_pos\_x[1,],sigma)+1/(2\*C)

> h1\_inv=solve(as.matrix(h1)) # the inverse of H1

> #the iterative method

> hn\_inv=h1\_inv # the inverse of Hn starts from the inverse of H1

> n=5

> #a loop for computing the inverse of Hn+1, K=1

> for (i in 1:n){

+ # compute Hn by the iterative block method, K=1

+ cat("K=1,n=",i,",","inverse","of","H",i,"=","\n")

+ print(hn\_inv)

+ hnk\_inv=HNK\_inv2(hn\_inv,data\_pos\_x[1:i,],data\_pos\_x[(i+1),])

+ hn\_inv=hnk\_inv

+ }

K=1,n= 1 , inverse of H 1 =

[,1]

[1,] 0.9090909

K=1,n= 2 , inverse of H 2 =

[,1] [,2]

[1,] 0.9230632 -0.1135664

[2,] -0.1135664 0.9230632

K=1,n= 3 , inverse of H 3 =

[,1] [,2] [,3]

[1,] 0.9342764 -0.1023532 -0.1023532

[2,] -0.1023532 0.9342764 -0.1023532

[3,] -0.1023532 -0.1023532 0.9342764

K=1,n= 4 , inverse of H 4 =

[,1] [,2] [,3] [,4]

[1,] 0.9361322 -0.1023322782 -0.1023322782 -0.0411165986

[2,] -0.1023323 0.9342766065 -0.1023529941 -0.0004642119

[3,] -0.1023323 -0.1023529941 0.9342766065 -0.0004642119

[4,] -0.0411166 -0.0004642119 -0.0004642119 0.9109575733

K=1,n= 5 , inverse of H 5 =

[,1] [,2] [,3] [,4] [,5]

[1,] 0.936155698 -0.102542448 -0.1023343621 -0.0416900474 0.0046628893

[2,] -0.102542448 0.936155698 -0.1023343621 0.0046628893 -0.0416900474

[3,] -0.102334362 -0.102334362 0.9342767912 -0.0004133745 -0.0004133745

[4,] -0.041690047 0.004662889 -0.0004133745 0.9249468671 -0.1137512793

[5,] 0.004662889 -0.041690047 -0.0004133745 -0.1137512793 0.9249468671

In the similar way, we can compute ‘alpha\_n’ vector from n=1 to n=5 by the iterative method.

> hn\_inv=h1\_inv # the inverse of Hn starts from the inverse of H1

> n=5

> for (i in 1:n){

+ # compute alpha\_n by the iterative block method, K=1

+ alpha\_n=alpha1(hn\_inv,data\_pos\_x[1:i,])

+ hnk\_inv=HNK\_inv2(hn\_inv,data\_pos\_x[1:i,],data\_pos\_x[(i+1),])

+ hn\_inv=hnk\_inv

+ cat("n=",i,",","alpha",i,"=",alpha\_n,"\n")

+ }

n= 1 , alpha 1 = 1

n= 2 , alpha 2 = 0.5 0.5

n= 3 , alpha 3 = 0.3333333 0.3333333 0.3333333

n= 4 , alpha 4 = 0.2287811 0.2416314 0.2416314 0.2879561

n= 5 , alpha 5 = 0.1894381 0.1894381 0.1988601 0.2111319 0.2111319

For the direct method, assume K=1, C=5, sigma=1, then we compute based on the matrix K named as ‘Km’, C and the identity matrix I. we firstly define a function Hn(data) with an input (n observations) and an output (Hn named as ‘hn’). Then we use a loop to compute by inv(). Here we use 5 observations to compute and correspond to the results through the iterative block method.

> #define a function of computing Hn directly

> Hn=function(data){

+ d=dim(data)[1] # the number of observation,data is dataframe type

+ Km=matrix(0,d,d) # K matrix named as Km

+ I=diag(d) # d dimensions identity matrix

+ for (i in 1:d){

+ for (j in 1:d) {

+ Km[i,j]=k(data[i,],data[j,],sigma)}

+ }

+ hn=Km+(1/(2\*C))\*I # compute the matrix Hn

+ return (hn)

+ }

> N=5 # assume we have 5 observations

> for (i in 1:N){

+ #compute Hn by direct method

+ hnk\_inv=solve(Hn(data\_pos\_x[1:i,]))

+ cat("K=1,n=",i,",","inverse","of","H",i,"=","\n")

+ print(hnk\_inv)

+ }

K=1,n= 1 , inverse of H 1 =

[,1]

[1,] 0.9090909

K=1,n= 2 , inverse of H 2 =

[,1] [,2]

[1,] 0.9230632 -0.1135664

[2,] -0.1135664 0.9230632

K=1,n= 3 , inverse of H 3 =

[,1] [,2] [,3]

[1,] 0.9342764 -0.1023532 -0.1023532

[2,] -0.1023532 0.9342764 -0.1023532

[3,] -0.1023532 -0.1023532 0.9342764

K=1,n= 4 , inverse of H 4 =

[,1] [,2] [,3] [,4]

[1,] 0.9361322 -0.1023322782 -0.1023322782 -0.0411165986

[2,] -0.1023323 0.9342766065 -0.1023529941 -0.0004642119

[3,] -0.1023323 -0.1023529941 0.9342766065 -0.0004642119

[4,] -0.0411166 -0.0004642119 -0.0004642119 0.9109575733

K=1,n= 5 , inverse of H 5 =

[,1] [,2] [,3] [,4] [,5]

[1,] 0.936155698 -0.102542448 -0.1023343621 -0.0416900474 0.0046628893

[2,] -0.102542448 0.936155698 -0.1023343621 0.0046628893 -0.0416900474

[3,] -0.102334362 -0.102334362 0.9342767912 -0.0004133745 -0.0004133745

[4,] -0.041690047 0.004662889 -0.0004133745 0.9249468671 -0.1137512793

[5,] 0.004662889 -0.041690047 -0.0004133745 -0.1137512793 0.9249468671

In the similar way, we can compute ‘alpha\_n’ vector from n=1 to n=5 by the direct method.

> for (i in 1:n){

+ # compute alpha\_n by direct method

+ hnk\_inv=solve(Hn(data\_pos\_x[1:i,]))

+ alpha\_n=alpha1(hnk\_inv,data\_pos\_x[1:i,])

+ cat("n=",i,",","alpha",i,"=",alpha\_n,"\n")

+ }

n= 1 , alpha 1 = 1

n= 2 , alpha 2 = 0.5 0.5

n= 3 , alpha 3 = 0.3333333 0.3333333 0.3333333

n= 4 , alpha 4 = 0.2287811 0.2416314 0.2416314 0.2879561

n= 5 , alpha 5 = 0.1894381 0.1894381 0.1988601 0.2111319 0.2111319

Therefore, comparing the results of Hn and alpha\_n through the iterative block method and the direct method indicates that both methods obtain the same results.

c. We will test the speed and effffectiveness of the presented algorithm by applying LS-SVDD and LS-SVDD update (K = 1), and LS-SVDD and LS-SVDD block update (K = 15) to the following data sets available from the UCI Machine Learning Repository. All features for all experiments are normalized to the range [[1, +1]. We will use a Gaussian kernel with σ = 1. Also, we will set C to 5 and N to 1. Your results should include “Training accuracy (%)”, “Testing accuracy (%)” and “Training Time (s)”.

For K=1, Our steps are: (1) redefine the radius and dz functions named as R2(alpha\_n,train,ap3) and DZ(alpha\_n,train,test,ap3) for accelerating the later computations, where the R2() function has three inputs (the alpha\_n vector, the training set, and the constant only related to the training set); the DZ() function has four inputs (the alpha\_n vector, the training set, the test vector, and the constant only related to the training set); (2) split datasets into 90% ‘data\_pos\_x’for training and 10% ‘data\_pos\_x’ for test, so we get the ‘train’ and ‘test’ datasets; (3) initialize the parameters for training; (4) start the training process and record ‘Training time’; (5) compute the ‘training accuracy’ and the ‘testing accuracy’.

Step (1) redefines the radius and dz functions named as R2(alpha\_n,train,ap3) for accelerating the later computations, where the R2() function has three inputs (the alpha\_n vector, the training set, and the constant only related to the training set) and its return value is the radius value named as ‘r2’.

> #define R2

> R2=function(alpha\_n,train,ap3){

+ # R2 consists of three average parts:ap1,ap2,ap3

+ d=dim(train)[1]

+ p1=rep(0,d)

+ p2=matrix(0,d,d)

+ for (i in 1:d){

+ p1[i]=k(train[i,],train[i,],sigma)}

+ ap1=sum(p1)/d

+

+ for (i in 1:d){

+ for (j in 1:d){

+ p2[i,j]=k(train[i,],train[j,],sigma) }

+ }

+ ap2=sum(p2 %\*% as.matrix(alpha\_n))/d

+

+ r2=ap1-2\*ap2+ap3

+ return(r2)

+ }

Redefine DZ(alpha\_n,train,test,ap3) for accelerating the later computations, where the DZ() function has four inputs (the alpha\_n vector, the training set, the test vector, and the constant only related to the training set) and its return value is the dz value named as ‘dz’.

> #define DZ

> DZ=function(alpha\_n,train,test,ap3){

+ # assume dz consists of three parts:ap1,ap2,ap3

+ d=dim(train)[1]

+ ap1=k(test,test,sigma)

+ p2=rep(0,d)

+ for (i in 1:d) {

+ p2[i]=k(test,train[i,],sigma)}

+

+ ap2=(t(alpha\_n) %\*% p2)[1]

+ dz=ap1-2\*ap2+ap3

+ return(dz)

+ }

For the input ‘ap3’ as a constant only related to the training set, we display the vertorization way to compute it instead of using ‘for loop’ in function ‘ap’, which aims to save the execution time in the training step.

> # the third term in dz and R2 is a constant ap3

> #define the ap3 function

> ap=function(alpha\_n,train){

+ # the third term in dz/R2 is a constant

+ d=dim(train)[1]

+ p3=matrix(0,d,d)

+ for (i in 1:d){

+ for (j in 1:d){

+ p3[i,j]=k(train[i,],train[j,],sigma) }

+ }

+ ap3=sum(as.matrix(alpha\_n %\*% t(alpha\_n)) %\*% p3) # vectorization, not loop

+ return (ap3)

+ }

Step (2) splits datasets into ‘data\_pos\_x’ for training and ‘data\_pos\_x’ for test, so we get the ‘train’ and ‘test’ datasets.

> set.seed(10)

> index1=sample(c(1:dim(data\_pos\_x)[1]),106)

> train=data\_pos\_x[index1,] # for training

> index2=sample(c(1:dim(data\_neg\_x)[1]),31)

> test=data\_neg\_x[index2,] # for test

Step (3) initializes the parameters for training by giving H1 and its inverse ‘h1\_inv’, setting K=1 by ‘kk=1’ and C=5, sigma=1.

> h1=k(train[1,],train[1,],sigma)+1/(2\*C)

> h1\_inv=solve(h1) # the inverse of H1

> hn\_inv=h1\_inv # the inverse of Hn starts from the inverse of H1

> kk=1

> tn2=round((dim(train)[1]-1)/kk) # the number of iterations

> C=5 # the penalty factor

> sigma=1 # the parameter of Gaussian Kernel

Step (4) starts the training process and record ‘Training time’. Here we will compute the in a iterative fashion indicated by ‘hnk\_incv’, the ‘alpha\_n’ vector, the constant input ap3 in functions R2() and DZ(), the radius ‘r2’ and the ‘Training time’ indicated by ‘proc.time()-start\_time6’.

> #training starts at K=1,N=1

> start\_time6=proc.time()

> for (i in 1:tn2){

+ d1=dim(hn\_inv)[1]

+ hnk\_inv=HNK\_inv2(hn\_inv,train[1:d1,],train[(d1+1):(d1+kk),])

+ hn\_inv=hnk\_inv

+ cat("the number of iteration=",i,"time=",(proc.time()-start\_time6),"seconds","\n")

+ }

> alpha\_n=alpha1(hn\_inv,train) # iterative alpha\_n

> cat('Time now=',proc.time()-start\_time6,"seconds","\n")

Time now= 17.9 seconds

> ap3=ap(alpha\_n,train)

> cat('Time now=',proc.time()-start\_time6,"seconds","\n")

Time now= 48.64 seconds

> r2=R2(alpha\_n,train,ap3) # trained radius R2

> cat("R2=",r2,"\n","the sum of alpha",sum(alpha\_n),"\n",

+ "K=1, Training time (s):",proc.time()-start\_time6,"\n")

R2= 2.349632

the sum of alpha 1

K=1, Training time (s): 78.85

Step (5) computes the training accuracy denoted by ‘train\_accuracy’ (the number of targets ‘ta’ / the number of training observations ‘dim(train)[1]’).

> #compute the training accuracy when K=1

> ta=0

> for (i in 1:dim(train)[1]){

+ dz=DZ(alpha\_n,train,train[i,],ap3)

+ if (dz<=r2){

+ ta=ta+1

+ cat("n=",i,"It's a target","\n")}

+ else{

+ cat("n=",i,"It's an outlier","\n")}

+ }

n= 1 It's a target

n= 2 It's an outlier

n= 3 It's a target

n= 4 It's a target

…….

> train\_accuracy=ta/dim(train)[1]

> cat("K=1, Training accuracy(%):",train\_accuracy\*100,"\n")

K=1, Training accuracy(%): 49.0566

Similarly, the testing accuracy denoted by ‘test\_accuracy’ (the number of outliers ‘dim(test)[1]-tt’ / the number of training observations ‘dim(test)[1]’).

> # compute the test accuracy when K=1

> tt=0

> for (i in 1:dim(test)[1]){

+ dz=DZ(alpha\_n,train,test[i,],ap3)

+ if (dz<=r2){

+ tt=tt+1

+ cat("n=",i,"It's a target","\n")}

+ else{

+ cat("n=",i,"It's an outlier","\n")}

+ }

n= 1 It's an outlier

n= 2 It's an outlier

n= 3 It's an outlier

n= 4 It's an outlier

…….

n= 31 It's an outlier

> test\_accuracy=(dim(test)[1]-tt)/dim(test)[1]

> cat("K=1, Testing accuracy(%):",test\_accuracy\*100,"\n")

K=1, Testing accuracy(%): 100

For K=15, we have the almost same steps and codes like we do for K=1 case above, except now ‘kk=15’. Comparing the results when K=1 and K=15, we can observe that both cases have the same radius R2 and training/testing accuracy, while the training time at K=15 is more than K=1, which is different from the result we got in Python where less iterations, less runtime in total .

> cat("R2=",r2,"\n","the sum of alpha",sum(alpha\_n),"\n",

+ "K=15, Training time (s):",proc.time()-start\_time6,"\n")

> cat("K=15, Training accuracy(%):",train\_accuracy\*100,"\n")

> cat("K=15, Testing accuracy(%):",test\_accuracy\*100,"\n")

R2= 2.349632

the sum of alpha 1

K=15, Training time (s): 81.53

K=15, Training accuracy(%): 49.0566

K=15, Testing accuracy(%): 100

Problem 5

a. Using the ‘charlie.csv’ data set, starting with the first row, add a block of *k* = 4 samples rows and use your updating codes from problem 4 to obtain .

Here we use the function ‘HNK\_inv2’ we define in problem 4 to compute as below.

> C=5 # the penalty factor

> sigma=1 # the parameter of Gaussian Kernel

> h1=k(trainx[1,],trainx[1,],sigma)+1/(2\*C)

> h1\_inv=solve(h1) # the inverse of H1

> h5\_inv=HNK\_inv2(h1\_inv,trainx[1,],trainx[2:5,])#add 4 new samples

> cat("K=4,","inverse","of","H5=","\n")

> print(h5\_inv)

K=4, inverse of H5=

[,1] [,2] [,3] [,4] [,5]

[1,] 9.090910e-01 1.583941e-07 -3.286162e-04 2.431773e-08 -1.599209e-11

[2,] 1.583941e-07 9.090919e-01 -4.391082e-04 -8.582141e-04 5.612419e-07

[3,] -3.286162e-04 -4.391082e-04 9.090912e-01 -6.727224e-05 4.424032e-08

[4,] 2.431773e-08 -8.582141e-04 -6.727224e-05 9.090921e-01 -5.987784e-04

[5,] -1.599209e-11 5.612419e-07 4.424032e-08 -5.987784e-04 9.090913e-01

b. Next, at each future step, we add a chunk of *k* = 5 observations and discard the oldest *h* = 3 observations. This is sometimes called the “Block Add and Forget Algorithm”. Write an R function to updatefor the “Block Add and Forget Algorithm”, i.e. after adding *k* observations, using eq(11), and removing the oldest *h* observations, using eq(12).

For Block Add and Forget Algorithm, we define a function ‘D\_inv’ to implement this method. This function has three inputs (‘h5\_inv’ means the the inverse of H5, ‘traindata’ means the training set, and ‘newdata’ means the next sequential dataset) and a return value of ‘hn\_h\_inv’. In the function, we use a ‘for’ loop to simulate the process of adding ‘kk=5’ new samples and removing ‘hh=3’ oldest sample. When adding the new kk samples, we use ‘HNK\_inv2’ function in eq (11) to get the inverse of ‘hnk\_inv’. When removing the oldest hh samples, we use ‘hn\_h\_inv’ in the loop to obtain the inverse of D in eq (12).

> #define a function to update the inverse of Hn-k or D in eq(12)

> D\_inv=function(h5\_inv,traindata,newdata){

+ kk=5 # each adding 5 samples

+ hh=3 # each remove 3 samples

+ d=round(dim(newdata)[1]/kk) # the number of iterations

+ for (i in 1:d){

+ hnk\_inv=HNK\_inv2(h5\_inv,traindata,newdata[((i-1)\*kk+1):(kk\*i),])

+ row\_h=hnk\_inv[1:hh,] # h rows of hnk\_inv

+ row\_n\_h=hnk\_inv[-(1:hh),] # last n-h rows of hnk\_inv

+ u11=row\_h[,1:hh] # U11

+ u12=row\_h[,-(1:hh)] # U12

+ u12\_t=row\_n\_h[,1:hh] # t(U12)

+ un\_h=row\_n\_h[,-(1:hh)] # D matrix

+ hn\_h\_inv=un\_h-u12\_t %\*% solve(u11) %\*% u12 # inverse of U10 in eq(12)

+ # remove the oldest h and add new k ones

+ traindata=rbind(traindata[-(1:hh),],newdata[((i-1)\*kk+1):(kk\*i),])

+ h5\_inv=hn\_h\_inv

+ }

+ return(hn\_h\_inv)

+ }

Then, we test the function ‘D\_inv’ above to compute ‘H11\_inv’ on training set as below.

> #if we only use the first 20 training samples from class 1

> h5\_inv=HNK\_inv2(h1\_inv,trainx[1,],trainx[2:5,])#add 4 new samples

> H11\_inv=D\_inv(h5\_inv,trainx[1:5,],trainx[6:20,])

> cat("the inverse of H11=","\n")

> print(H11\_inv)

the inverse of H11=

[,1] [,2] [,3] [,4] [,5] [,6] [,7]

[1,] 9.153715e-01 1.923857e-05 -2.055502e-04 1.583766e-07 -7.472013e-02 1.203882e-07 4.166945e-06

[2,] 1.923857e-05 9.093492e-01 -1.170644e-07 -8.306251e-08 2.695154e-04 -2.354682e-03 2.366275e-09

[3,] -2.055502e-04 -1.170644e-07 9.094687e-01 6.963799e-10 -3.614910e-04 -7.332560e-10 -1.849592e-02

[4,] 1.583766e-07 -8.306251e-08 6.963799e-10 9.090921e-01 -1.941985e-06 1.446005e-10 -2.325664e-11

[5,] -7.472013e-02 2.695154e-04 -3.614910e-04 -1.941985e-06 9.155821e-01 1.688246e-06 7.305145e-06

[6,] 1.203882e-07 -2.354682e-03 -7.332560e-10 1.446005e-10 1.688246e-06 9.090970e-01 1.482163e-11

[7,] 4.166945e-06 2.366275e-09 -1.849592e-02 -2.325664e-11 7.305145e-06 1.482163e-11 9.094672e-01

[8,] -1.272737e-02 -2.785611e-07 -1.144585e-03 -2.122584e-09 1.001476e-03 -1.744361e-09 2.323834e-05

[9,] 2.994054e-05 3.566192e-06 1.439918e-07 -1.058799e-03 -3.654234e-04 4.821443e-08 -2.925288e-09

[10,] -9.443757e-04 -1.514647e-02 8.061880e-06 4.562849e-07 -1.879714e-02 -1.041223e-04 -1.629532e-07

[11,] 4.225990e-10 -2.574911e-14 3.295125e-07 4.040781e-15 -1.506759e-10 -1.613013e-16 -3.705204e-04

[,8] [,9] [,10] [,11]

[1,] -1.272737e-02 2.994054e-05 -9.443757e-04 4.225990e-10

[2,] -2.785611e-07 3.566192e-06 -1.514647e-02 -2.574911e-14

[3,] -1.144585e-03 1.439918e-07 8.061880e-06 3.295125e-07

[4,] -2.122584e-09 -1.058799e-03 4.562849e-07 4.040781e-15

[5,] 1.001476e-03 -3.654234e-04 -1.879714e-02 -1.506759e-10

[6,] -1.744361e-09 4.821443e-08 -1.041223e-04 -1.613013e-16

[7,] 2.323834e-05 -2.925288e-09 -1.629532e-07 -3.705204e-04

[8,] 9.092693e-01 -4.013510e-07 1.390207e-05 -3.566192e-08

[9,] -4.013510e-07 9.090924e-01 -3.630734e-04 6.640850e-14

[10,] 1.390207e-05 -3.630734e-04 9.097358e-01 1.975158e-12

[11,] -3.566192e-08 6.640850e-14 1.975158e-12 9.090911e-01

Also, we test the function ‘D\_inv’ above to compute ‘H15\_inv’ on the full set (contains ‘trainx’ and ‘testx’) as below.

> #if we use all data from class 1 and -1

> traind=rbind(trainx,testx) # 30 samples in total

> h5\_inv=HNK\_inv2(h1\_inv,trainx[1,],trainx[2:5,])#add 4 new samples

> H15\_inv=D\_inv(h5\_inv,traind[1:5,],traind[6:30,])

> cat("the inverse of H15=","\n")

> print(H15\_inv)

the inverse of H15=

[,1] [,2] [,3] [,4] [,5] [,6] [,7]

[1,] 9.173464e-01 2.350040e-08 -1.563336e-11 4.088054e-14 -2.437376e-04 -1.174724e-13 -8.702285e-02

[2,] 2.350040e-08 9.090909e-01 1.253299e-12 1.454851e-08 -3.427525e-08 -6.555494e-08 -6.624459e-07

[3,] -1.563336e-11 1.253299e-12 9.090911e-01 -3.749897e-04 4.153977e-15 1.464408e-05 1.482860e-12

[4,] 4.088054e-14 1.454851e-08 -3.749897e-04 1.002773e+00 -5.604538e-16 -3.064988e-01 -1.365953e-14

[5,] -2.437376e-04 -3.427525e-08 4.153977e-15 -5.604538e-16 9.090930e-01 2.506216e-15 -1.336406e-03

[6,] -1.174724e-13 -6.555494e-08 1.464408e-05 -3.064988e-01 2.506216e-15 1.002773e+00 5.617734e-14

[7,] -8.702285e-02 -6.624459e-07 1.482860e-12 -1.365953e-14 -1.336406e-03 5.617734e-14 9.173483e-01

[8,] -3.030103e-25 -8.086542e-20 6.490331e-15 -5.575285e-12 3.135325e-27 1.703761e-12 8.311202e-26

[9,] 1.937008e-22 -1.551302e-23 -1.126385e-11 4.647287e-15 -5.146871e-26 -1.817731e-16 -1.837297e-23

[10,] 2.391312e-19 -2.981593e-17 -1.390975e-08 -2.050039e-09 -6.241451e-23 6.276950e-10 -2.266211e-20

[11,] 4.207170e-22 3.393799e-23 -2.446499e-11 1.476004e-14 -1.117923e-25 -1.820050e-15 -3.990602e-23

[12,] -5.374591e-09 -1.376879e-16 -8.758814e-08 3.613718e-11 1.428273e-12 -1.448830e-12 5.096848e-10

[13,] 9.113033e-24 -9.349122e-25 -7.942932e-13 3.386922e-16 -2.421302e-27 -1.616992e-17 -8.644832e-25

[14,] 3.732327e-17 8.934038e-25 -4.491189e-14 1.852558e-17 -9.918491e-21 -7.231989e-19 -3.539451e-18

[15,] 8.805994e-19 -3.578482e-21 -1.640872e-10 -1.662666e-13 -2.340151e-22 6.824032e-14 -8.350932e-20

[,8] [,9] [,10] [,11] [,12] [,13] [,14]

[1,] -3.030103e-25 1.937008e-22 2.391312e-19 4.207170e-22 -5.374591e-09 9.113033e-24 3.732327e-17

[2,] -8.086542e-20 -1.551302e-23 -2.981593e-17 3.393799e-23 -1.376879e-16 -9.349122e-25 8.934038e-25

[3,] 6.490331e-15 -1.126385e-11 -1.390975e-08 -2.446499e-11 -8.758814e-08 -7.942932e-13 -4.491189e-14

[4,] -5.575285e-12 4.647287e-15 -2.050039e-09 1.476004e-14 3.613718e-11 3.386922e-16 1.852558e-17

[5,] 3.135325e-27 -5.146871e-26 -6.241451e-23 -1.117923e-25 1.428273e-12 -2.421302e-27 -9.918491e-21

[6,] 1.703761e-12 -1.817731e-16 6.276950e-10 -1.820050e-15 -1.448830e-12 -1.616992e-17 -7.231989e-19

[7,] 8.311202e-26 -1.837297e-23 -2.266211e-20 -3.990602e-23 5.096848e-10 -8.644832e-25 -3.539451e-18

[8,] 9.090909e-01 1.511528e-16 -2.879093e-07 6.538284e-13 -6.265786e-22 1.451838e-15 -9.009110e-23

[9,] 1.511528e-16 9.091059e-01 -4.772740e-10 -3.695631e-03 1.098622e-18 -5.641022e-08 -1.932091e-12

[10,] -2.879093e-07 -4.772740e-10 9.090909e-01 -2.064502e-06 1.343983e-15 -4.584267e-09 2.844677e-16

[11,] 6.538284e-13 -3.695631e-03 -2.064502e-06 9.091059e-01 2.356208e-18 6.687018e-11 5.854235e-15

[12,] -6.265786e-22 1.098622e-18 1.343983e-15 2.356208e-18 9.090909e-01 7.690531e-16 -6.313218e-09

[13,] 1.451838e-15 -5.641022e-08 -4.584267e-09 6.687018e-11 7.690531e-16 9.090909e-01 -5.640995e-08

[14,] -9.009110e-23 -1.932091e-12 2.844677e-16 5.854235e-15 -6.313218e-09 -5.640995e-08 9.090909e-01

[15,] 7.410108e-15 1.678861e-13 -2.339787e-08 3.497615e-14 -1.484726e-10 -2.704427e-06 3.106307e-14

[,15]

[1,] 8.805994e-19

[2,] -3.578482e-21

[3,] -1.640872e-10

[4,] -1.662666e-13

[5,] -2.340151e-22

[6,] 6.824032e-14

[7,] -8.350932e-20

[8,] 7.410108e-15

[9,] 1.678861e-13

[10,] -2.339787e-08

[11,] 3.497615e-14

[12,] -1.484726e-10

[13,] -2.704427e-06

[14,] 3.106307e-14

[15,] 9.090909e-01

c. Compare the “Training accuracy (%)”, “Testing accuracy (%)” and “Training Time (s)’ between the ‘Block Add and Forget Algorithm’ and direct algorithms. Note that the direct approach will be containing 5, 10, 15, 20 observations at each iteration while the updating algorithm will have 5, 7, 9, 11, ... observations at each iteration.

For the recursive method, we define a function ‘ite\_runtime’ based on the ‘D\_inv’ function to record the iterative runtimes ‘runtime’. The function ‘ite\_runtime’ contains the same three inputs of ‘D\_inv’ and almost same procedures but has one more variables ‘runtime’ in the loop.

> # iterative method

> ite\_runtime=function(h5\_inv,traindata,newdata){

+ kk=5 # each adding 5 samples

+ hh=3 # each remove 3 samples

+ d=round(dim(newdata)[1]/kk) # the number of iterations=5 if use 30 samples

+ runtime=rep(0,d)

+ start\_time=proc.time() # recording runtime starts

+ for (i in 1:d){

+ hnk\_inv=HNK\_inv2(h5\_inv,traindata,newdata[((i-1)\*kk+1):(kk\*i),])

+ row\_h=hnk\_inv[1:hh,] # h rows of hnk\_inv

+ row\_n\_h=hnk\_inv[-(1:hh),] # last n-h rows of hnk\_inv

+ u11=row\_h[,1:hh] # U11

+ u12=row\_h[,-(1:hh)] # U12

+ u12\_t=row\_n\_h[,1:hh] # t(U12)

+ un\_h=row\_n\_h[,-(1:hh)] # D matrix

+ hn\_h\_inv=un\_h-u12\_t %\*% solve(u11) %\*% inverse of D in eq(12)

+ # remove the oldest h and add new k ones

+ traindata=rbind(traindata[-(1:hh),],newdata[((i-1)\*kk+1):(kk\*i),])

+ h5\_inv=hn\_h\_inv

+

+ runtime[i]=proc.time()-start\_time

+ cat(i,"iterations","runtime=",runtime[i],"seconds","\n")

+ }

+ return(runtime)

+ }

> h1=k(trainx[1,],trainx[1,],sigma)+1/(2\*C)

> h1\_inv=solve(h1) # the inverse of H1

> h5\_inv=HNK\_inv2(h1\_inv,trainx[1,],trainx[2:5,])#add 4 new samples

> runtime1=ite\_runtime(h5\_inv,traind[1:5,],traind[6:30,])

1 iterations runtime= 0.15 seconds

2 iterations runtime= 0.33 seconds

3 iterations runtime= 0.5 seconds

4 iterations runtime= 0.7 seconds

5 iterations runtime= 0.92 seconds

Similarly, for the direct method, we use ‘Hn’ function to compute each new inverse of Hn directly through a ‘for’ loop as below.

> #direct method

> N=5 # the number of iterative

> kk=5 # each adding 5 samples

> runtime2=rep(0,N) # record the runtime for direct method

> start\_time2 = proc.time()

>

> for (i in 1:N){

+ hn\_inv=solve(Hn(traind[1:(kk\*i),]))

+ runtime2[i]=proc.time()-start\_time2

+ cat(i,"iterations","runtime=",runtime2[i],"seconds","\n")

+ }

1 iterations runtime= 0.12 seconds

2 iterations runtime= 0.43 seconds

3 iterations runtime= 1.04 seconds

4 iterations runtime= 2.01 seconds

5 iterations runtime= 3.41 seconds

Finally, we compare and plot the results of runtimes in both methods as below.

> #plot the runtimes of two methods

> x <- rep(1:N)

> plot(x,runtime2,type="l",col="blue",xlab="the number of iterations",

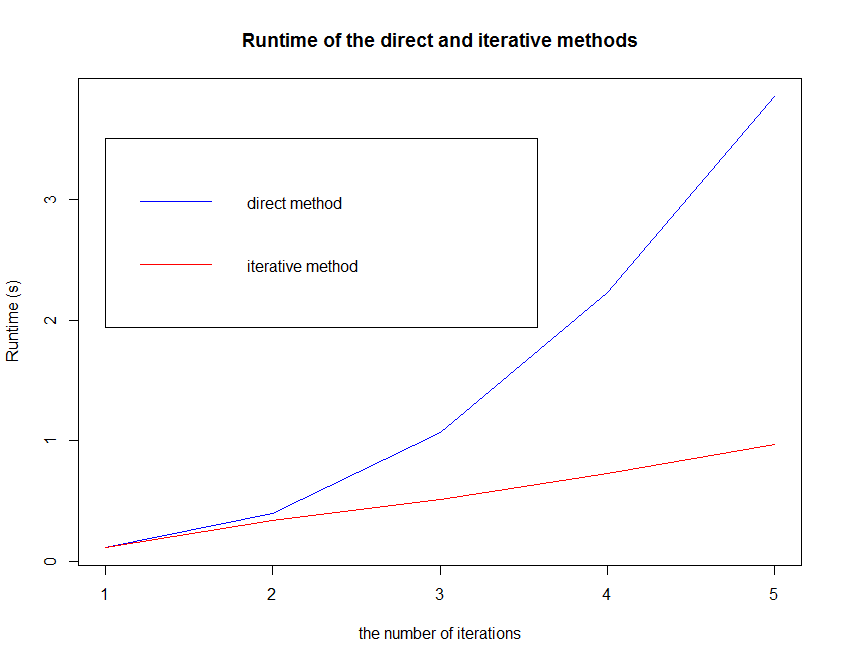
+ ylab="Runtime (s)", main="Runtime of the direct and iterative methods")

> #Add more data to the plot

> lines(x,runtime1,col="red") #add runtimes of iterative method

> legend(x=1,y=3,c("direct method"," iterative method"),cex=1.,

+ col=c("blue","red"),lty=c(1,1)) #add legend



For the training and test accuracy in iterative method, we use the inverse of H11 to compute the alpha vector and current radius R2, then use a ‘for’ loop to obtain the training and test accuracy as below.

> #compute the training accuracy in iterative method

> alpha\_n=alpha1(H11\_inv,trainx[10:20,]) # iterative alpha\_n

> ap3=ap(alpha\_n,trainx[10:20,]) # the constant term in R2 and dz

> r2=R2(alpha\_n,trainx[10:20,],ap3) # trained radius R2

>

> ta=0 # set the counter of prediction

> for (i in 1:dim(trainx[10:20,])[1]){

+ dz=DZ(alpha\_n,trainx[10:20,],(trainx[10:20,])[i,],ap3)

+ if (dz<=r2){

+ ta=ta+1

+ cat("n=",i,"It's a target","\n")}

+ else{

+ cat("n=",i,"It's an outlier","\n")}

+ }

n= 1 It's a target

n= 2 It's an outlier

n= 3 It's an outlier

n= 4 It's an outlier

n= 5 It's a target

n= 6 It's an outlier

n= 7 It's an outlier

n= 8 It's an outlier

n= 9 It's an outlier

n= 10 It's a target

n= 11 It's an outlier

> train\_accuracy=ta/dim(trainx[10:20,])[1]

> cat(" Training accuracy(%):",train\_accuracy\*100,"\n")

Training accuracy(%): 27.27273

Similarly, we can obtain the test accuracy of iterative method as below by a ‘for’ loop.

> # compute the test accuracy in iterative method

> tt=0 # set the counter of prediction

> for (i in 1:dim(testx)[1]){

+ dz=DZ(alpha\_n,trainx[10:20,],testx[i,],ap3)

+ if (dz<=r2){

+ tt=tt+1

+ cat("n=",i,"It's a target","\n")}

+ else{

+ cat("n=",i,"It's an outlier","\n")}

+ }

n= 1 It's an outlier

n= 2 It's an outlier

n= 3 It's an outlier

n= 4 It's an outlier

n= 5 It's an outlier

n= 6 It's an outlier

n= 7 It's an outlier

n= 8 It's an outlier

n= 9 It's an outlier

n= 10 It's an outlier

> test\_accuracy=(dim(testx)[1]-tt)/dim(testx)[1]

> cat("Testing accuracy(%):",test\_accuracy\*100,"\n")

Testing accuracy(%): 100

For the training and test accuracy in direct method, we use the inverse of H20 to compute the alpha vector and current radius R2, then use a ‘for’ loop to obtain the training and test accuracy as below.

> ##compute the training accuracy in direct method

>

> N=4 # the number of iterative with 20 training samples

> kk=5 # each adding 5 samples

> for (i in 1:N){

+ hn\_inv=solve(Hn(traind[1:(kk\*i),]))

+ } #output the inverse of H20 finally

> #compute the training accuracy in direct method

> alpha\_n=alpha1(hn\_inv,trainx) # iterative alpha\_n

> ap3=ap(alpha\_n,trainx)

> r2=R2(alpha\_n,trainx,ap3) # trained radius R2

> ta=0 # set the counter of prediction

> for (i in 1:dim(trainx)[1]){

+ dz=DZ(alpha\_n,trainx,trainx[i,],ap3)

+ if (dz<=r2){

+ ta=ta+1

+ cat("n=",i,"It's a target","\n")}

+ else{

+ cat("n=",i,"It's an outlier","\n")}

+ }

n= 1 It's a target

n= 2 It's an outlier

n= 3 It's a target

n= 4 It's an outlier

n= 5 It's an outlier

n= 6 It's an outlier

n= 7 It's a target

n= 8 It's an outlier

n= 9 It's a target

n= 10 It's a target

n= 11 It's a target

n= 12 It's an outlier

n= 13 It's an outlier

n= 14 It's a target

n= 15 It's a target

n= 16 It's an outlier

n= 17 It's an outlier

n= 18 It's an outlier

n= 19 It's a target

n= 20 It's an outlier

> train\_accuracy=ta/dim(trainx)[1]

> cat(" Training accuracy(%):",train\_accuracy\*100,"\n")

Training accuracy(%): 45

Similarly, we can obtain the test accuracy of direct method as below by a ‘for’ loop.

> # compute the test accuracy in direct method

> tt=0 # set the counter of prediction

> for (i in 1:dim(testx)[1]){

+ dz=DZ(alpha\_n,trainx,testx[i,],ap3)

+ if (dz<=r2){

+ tt=tt+1

+ cat("n=",i,"It's a target","\n")}

+ else{

+ cat("n=",i,"It's an outlier","\n")}

+ }

n= 1 It's an outlier

n= 2 It's an outlier

n= 3 It's an outlier

n= 4 It's an outlier

n= 5 It's an outlier

n= 6 It's an outlier

n= 7 It's an outlier

n= 8 It's an outlier

n= 9 It's an outlier

n= 10 It's an outlier

> test\_accuracy=(dim(testx)[1]-tt)/dim(testx)[1]

> cat("Testing accuracy(%):",test\_accuracy\*100,"\n")

Testing accuracy(%): 100

d. Repeat question (c) but now by using the data set corresponding to your group. For this case, use *k* = 15 and *h* = 10.

Here we just modify the function ‘ite\_runtime’ by kk=15 and hh=10 instead of kk=5 and hh=3 in question (c).

> # iterative method

> ite\_runtime=function(h15\_inv,traindata,newdata){

+ kk=15 # each adding 15 samples

+ hh=10 # each remove 10 samples

># the number of iterations=5 if use 30 samples

+ d=round(dim(newdata)[1]/kk)

+ runtime=rep(0,d)

+

+ start\_time=proc.time() # recording runtime starts

+ for (i in 1:d){

+ hnk\_inv=HNK\_inv2(h15\_inv,traindata,newdata[((i-1)\*kk+1):(kk\*i),])

+ row\_h=hnk\_inv[1:hh,] # h rows of hnk\_inv

+ row\_n\_h=hnk\_inv[-(1:hh),] # last n-h rows of hnk\_inv

+ u11=row\_h[,1:hh] # U11

+ u12=row\_h[,-(1:hh)] # U12

+ u12\_t=row\_n\_h[,1:hh] # t(U12)

+ un\_h=row\_n\_h[,-(1:hh)] # D matrix

+ hn\_h\_inv=un\_h-u12\_t %\*% solve(u11) %\*% u12 # in eq(12)

+ # remove the oldest h and add new k ones

+ traindata=rbind(traindata[-(1:hh),],newdata[((i-1)\*kk+1):(kk\*i),])

+ h15\_inv= hn\_h\_inv

+ runtime[i]=proc.time() - start\_time

+ cat(i,"iterations","runtime=",runtime[i],"seconds","\n")

+ }

+ return(runtime)

+ }

Then, we can record the runtimes of 10 iterations as below.

> h1=k(data\_pos\_x[1,],data\_pos\_x[1,],sigma)+1/(2\*C)

> h1\_inv=solve(h1) # the inverse of H1

> h15\_inv=HNK\_inv2(h1\_inv,data\_pos\_x[1,],data\_pos\_x[2:15,])

> runtime1=ite\_runtime(h15\_inv,data\_pos\_x[1:15,],data\_pos\_x[16:165,])

1 iterations runtime= 1.88 seconds

2 iterations runtime= 4.13 seconds

3 iterations runtime= 6.8 seconds

4 iterations runtime= 9.79 seconds

5 iterations runtime= 12.69 seconds

6 iterations runtime= 14.96 seconds

7 iterations runtime= 17.39 seconds

8 iterations runtime= 21.13 seconds

9 iterations runtime= 25.75 seconds

10 iterations runtime= 30.73 seconds

Similarly, for the direct method, we use ‘Hn’ function to compute each new inverse of Hn directly through a ‘for’ loop as below.

> #direct method

> kk=15 # each adding 5 samples

> N=10 # the number of iterative=(165-16+1)/kk=10

> runtime2=rep(0,N) # record the runtime for direct method

> start\_time2 = proc.time()

>

> for (i in 1:N){

+ hn\_inv=solve(Hn(traind[1:(kk\*i),]))

+ runtime2[i]=proc.time()-start\_time2

+ cat(i,"iterations","runtime=",runtime2[i],"seconds","\n")

+ }

1 iterations runtime= 0.74 seconds

2 iterations runtime= 3.23 seconds

3 iterations runtime= 8.66 seconds

4 iterations runtime= 17.82 seconds

5 iterations runtime= 32.23 seconds

6 iterations runtime= 51.95 seconds

7 iterations runtime= 80.37 seconds

8 iterations runtime= 116.83 seconds

9 iterations runtime= 163.1 seconds

10 iterations runtime= 217.53 seconds

Finally, we compare and plot the results of runtimes in both methods as below.

#plot the runtimes of two methods

> x <- rep(1:N)

> plot(x,runtime2,type="l",col="blue",xlab="the number of iterations",

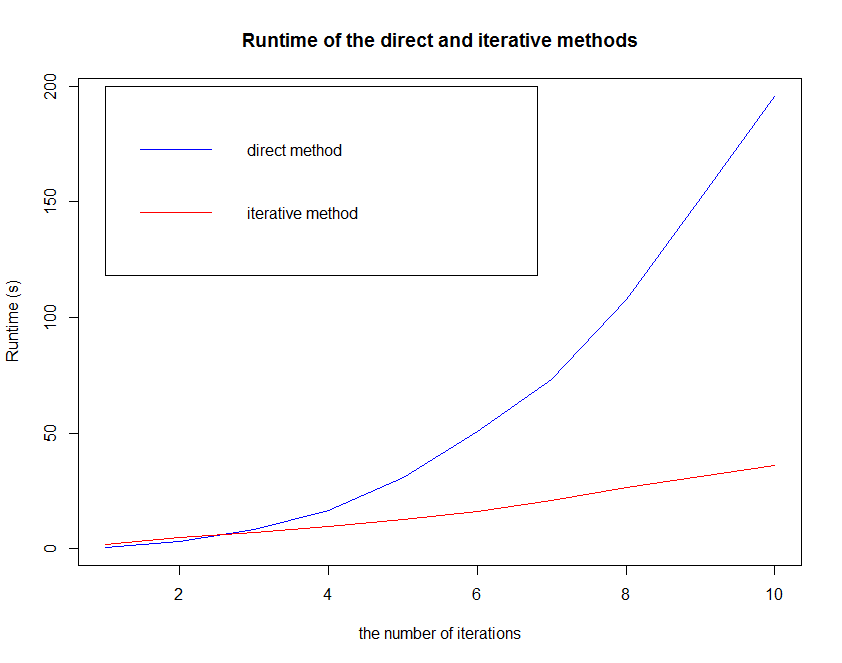
+ ylab="Runtime (s)", main="Runtime of the direct and iterative methods")

> #Add more data to the plot

> lines(x,runtime1,col="red") #add runtimes of iterative method

> legend(x=1,y=200,c("direct method","iterative method"),cex=1.,

+ col=c("blue","red"),lty=c(1,1)) #add legend



For the training and test accuracy in iterative method, we define a function ‘D\_inv1’ to get the inverse of H65, then use the inverse of H65 to compute the alpha vector and current radius R2, then use a ‘for’ loop to obtain the training and test accuracy as below. Note that the training set here contain 165 samples and the test set contains 31 samples.

Firstly, we define the function ‘D\_inv1’ based on ‘D\_inv’ and they are different in ‘kk’ and ‘hh’. The function ‘D\_inv1’ may help us output the final inverse of H65 after 10 iterations of adding ‘kk=15’ and removing ‘hh=10’ samples.

> D\_inv1=function(h5\_inv,traindata,newdata){

+ kk=15 # each adding 5 samples

+ hh=10 # each remove 3 samples

+ d=round(dim(newdata)[1]/kk) # the number of iterations

+ for (i in 1:d){

+ hnk\_inv=HNK\_inv2(h5\_inv,traindata,newdata[((i-1)\*kk+1):(kk\*i),])

+ row\_h=hnk\_inv[1:hh,] # h rows of hnk\_inv

+ row\_n\_h=hnk\_inv[-(1:hh),] # last n-h rows of hnk\_inv

+ u11=row\_h[,1:hh] # U11

+ u12=row\_h[,-(1:hh)] # U12

+ u12\_t=row\_n\_h[,1:hh] # t(U12)

+ un\_h=row\_n\_h[,-(1:hh)] # D matrix

+ hn\_h\_inv=un\_h-u12\_t %\*% solve(u11) %\*% u12 # inv(D) in eq(12)

+ # remove the oldest h and add new k ones

+ traindata=rbind(traindata[-(1:hh),],newdata[((i-1)\*kk+1):(kk\*i),])

+ h5\_inv=hn\_h\_inv

+ }

+ return(h5\_inv)

+ }

Use the inverse of H65 to compute the alpha vector and current radius R2, then use a ‘for’ loop to obtain the training accuracy as below.

> h1=k(data\_pos\_x[1,],data\_pos\_x[1,],sigma)+1/(2\*C) # H1

> h1\_inv=solve(h1) # the inverse of H1

> h15\_inv=HNK\_inv2(h1\_inv,data\_pos\_x[1,],data\_pos\_x[2:15,]) # H15

> hnk\_inv=D\_inv1(h15\_inv,data\_pos\_x[1:15,],data\_pos\_x[16:165,]) #H65

> alpha\_n=alpha1(hnk\_inv,data\_pos\_x[101:165,]) # iterative alpha\_n

> ap3=ap(alpha\_n,data\_pos\_x[101:165,]) # the constant term in R2 and dz

> r2=R2(alpha\_n,data\_pos\_x[101:165,],ap3) # trained radius R2

> ta=0 # set the counter of prediction

> for (i in 1:dim(data\_pos\_x[101:165,])[1]){

+ dz=DZ(alpha\_n,data\_pos\_x[101:165,],(data\_pos\_x[101:165,])[i,],ap3)

+ if (dz<=r2){

+ ta=ta+1

+ cat("n=",i,"It's a target","\n")}

+ else{

+ cat("n=",i,"It's an outlier","\n")}

+ }

n= 1 It's an outlier

n= 2 It's an outlier

……………..

n= 64 It's an outlier

n= 65 It's an outlier

> train\_accuracy=ta/dim(data\_pos\_x[101:165,])[1]

> cat(" Training accuracy(%):",train\_accuracy\*100,"\n")

Training accuracy(%): 47.69231

Similarly, we can obtain the test accuracy of iterative method as below by a ‘for’ loop.

> # compute the test accuracy in iterative method

> tt=0 # set the counter of prediction

> for (i in 1:dim(data\_neg\_x[1:31,])[1]){

+ dz=DZ(alpha\_n,data\_pos\_x[101:165,],(data\_neg\_x[1:31,])[i,],ap3)

+ if (dz<=r2){

+ tt=tt+1

+ cat("n=",i,"It's a target","\n")}

+ else{

+ cat("n=",i,"It's an outlier","\n")}

+ }

n= 1 It's an outlier

n= 2 It's an outlier

…………………….

n= 27 It's an outlier

n= 28 It's an outlier

n= 29 It's an outlier

n= 30 It's an outlier

n= 31 It's an outlier

> test\_accuracy=(dim(data\_neg\_x[1:31,])[1]-tt)/dim(data\_neg\_x[1:31,])[1]

> cat("Testing accuracy(%):",test\_accuracy\*100,"\n")

Testing accuracy(%): 100

For the training and test accuracy in direct method, we use the inverse of H165 to compute the alpha vector and current radius R2, then use a ‘for’ loop to obtain the training and test accuracy as below.

> ##compute the training accuracy in direct method

>

> hn\_inv=solve(Hn(data\_pos\_x[1:165,]))

> alpha\_n=alpha1(hn\_inv,data\_pos\_x[1:165,]) # iterative alpha\_n

> ap3=ap(alpha\_n,data\_pos\_x[1:165,])

> r2=R2(alpha\_n,data\_pos\_x[1:165,],ap3) # trained radius R2

> ta=0 # set the counter of prediction

> for (i in 1:dim(data\_pos\_x[1:165,])[1]){

+ dz=DZ(alpha\_n,data\_pos\_x[1:165,],(data\_pos\_x[1:165,])[i,],ap3)

+ if (dz<=r2){

+ ta=ta+1

+ cat("n=",i,"It's a target","\n")}

+ else{

+ cat("n=",i,"It's an outlier","\n")}

+ }

n= 1 It's a target

n= 2 It's an outlier

n= 3 It's a target

……

n= 163 It's an outlier

n= 164 It's an outlier

n= 165 It's an outlier

> train\_accuracy=ta/dim(data\_pos\_x[1:165,])[1]

> cat(" Training accuracy(%):",train\_accuracy\*100,"\n")

Training accuracy(%): 55.75758

Similarly, we can obtain the test accuracy of direct method as below by a ‘for’ loop.

> # compute the test accuracy in direct method

> tt=0 # set the counter of prediction

> for (I in 1:dim(data\_neg\_x[1:31,])[1]){

+ dz=DZ(alpha\_n,data\_pos\_x[1:165,],(data\_neg\_x[1:31,])[I,],ap3)

+ if (dz<=r2){

+ tt=tt+1

+ cat(“n=”,I,”It’s a target”,”\n”)}

+ else{

+ cat(“n=”,I,”It’s an outlier”,”\n”)}

+ }

n= 1 It’s an outlier

n= 2 It’s an outlier

n= 3 It’s an outlier

……..

n= 29 It’s an outlier

n= 30 It’s an outlier

n= 31 It’s an outlier

> test\_accuracy=(dim(data\_neg\_x[1:31,])[1]-tt)/dim(data\_neg\_x[1:31,])[1]

> cat(“Testing accuracy(%):”,test\_accuracy\*100,”\n”)

Testing accuracy(%): 100