## Problem 1

We have, ,

and

The join distribution of and -

Let and

and

where,

Jacobian matrix

So, the join distribution of and can be written as-

The marginal distribution of is-

Let

So, if and are two standard normal *pdf* then follows standard Cauchy distribution.

Now, to show that follows Cauchy Distribution, the following steps have been used.

1. Two random samples ( and have been generated from standard normal distribution using Box-Muller Algorithm. Detail procedure will be discussed later in this part.
2. Using those standard normal samples, Cauchy random sample has been generated using .
3. Histogram with Cauchy density curve has been presented.

**Step 1. Box-Muller Algorithm:**

The rnormalBX function has been used to generate two standard normal random sample. We can use this function to generate normal random sample with any mean and variance for any size of the sample. This function takes four parameters-

*= sample size of the standard normal random sample (Default 1)*

*mean = mean of the standard normal random sample (Default 0)*

*mean = variance of the standard normal random sample (Default 1)*

*seed = random seed (Default NULL)*

rnormalBX <- function(n=1, mean = 0, var = 1, seed = NULL){

At first, this function will check whether any value for random seed (seed) is given or not. If there is any value for seed (i.e. example seed = 123), it will set that value as random seed.

if(!is.null(seed)) set.seed(seed) # set seed

Then, it will generate two random sample () from uniform distribution between 0 and 1. Using that and , we have calculated and theta using the following formula.

and

Finally using following formula two random standard normal samples have been generated and combined those two column vector using cbind function.

and

u = runif(n, 0, 1)  
 v = runif(n, 0, 1)  
 r = sqrt(-2\*log(v))  
 theta = 2\*pi\*u  
 norm = cbind(r\*cos(theta), r\*sin(theta))

It is seen that, if the sample size is small, the mean and variance of the two standard normal sample may not equal to 0 and 1. That is way, I have standardized those samples using the following formula so that it becomes normal with mean 0 and variance 1.

if(n != 1){norm = (norm-colMeans(norm))/sqrt(diag(var(norm)))}

Then, if the parameters (mean, var) those are supplied in the function are not 0 or 1 respectively, it will use the following function to generate normal random sample with specified mean and variance.

Finally, it will return two normal random samples removing variables names by using unname function.

if (mean != 0 || var != 1){  
 norm = sqrt(var)\*norm + mean  
   
 }  
 return(unname(norm, force = TRUE))  
 }

Example for calling this function (rnormalBX) is given below-

x = rnormalBX(n = 1000, mean = 5, var = 10, seed = 123)

**Step 2. Cauchy Random Sample**

The function rcauchydist has been written to generate random sample from standard Cauchy distribution with size (say ). This function has two parameters-

*= sample size of the standard Cauchy random sample (Default 1)*

*seed = random seed (Default NULL).*

Within this function, I have used rnormalBX function that generate two standard normal random samples using Box-Muller algorithm. This function has been discussed in step 1. Using these two standard normal random samples, the Cauchy random sample has been generated using the following formula.

rcauchydist <- function(n=1, seed = NULL){  
 x = rnormalBX(n = n, mean = 0, var = 1, seed = seed)  
 return(x[,1]/x[,2])  
}

Example for calling this function (rcauchydist) is given below where a standard Cauchy random sample of size 1000 has been generated-

x = rcauchydist(n = 1000, seed = 4578)

I have written a function densityCurve to generate histogram with Cauchy density curve. I have also used this function in other part of this assignment. In this function, there are two parameters-

*data = sample data for which we want to generate histogram*

nbreaks *= number of class in histogram (Default NULL)*

densityCurve <- function(data , nbreaks = NULL){

At first, this function will check nbreaks parameter. If the parameter is “NULL”, it will generate best possible number of classes for the histogram using pretty function. The pretty function compute a sequence of about n+1 equally spaced ‘round’ values which cover the range of the values in x (given data). Then, a histogram has been drawn using the data supplied in this function with given or generated (if NULL) number of classes (nbreaks). In this hist function, I have used freq = FALSE so that the histogram shows relative frequency (proportion) in the axis.

if(is.null(nbreaks)){ nbreaks = pretty(data)}  
 title = "Histogram with Cauchy Curve"  
 his <- hist(data, breaks = nbreaks, freq = FALSE, col="red",

xlab="x", main=title, xlim = c(-30,30),

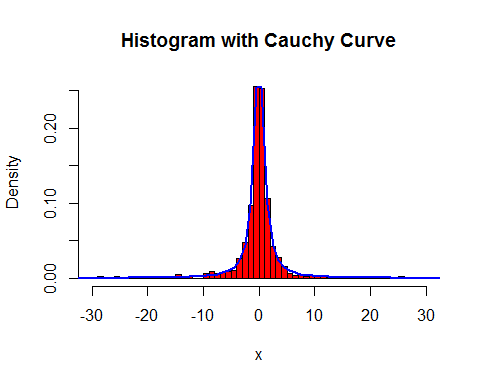
ylim = c(0,.4))

Now, to draw Cauchy density curve on this histogram, we need to generate a sequence of values from to which is the support of Cauchy random variable. To do so, we have used mid points of histogram and then calculated the density for those mid points from Cauchy distribution with location 0 and scale 1 using dcauchy function. Finally, we have added the density line using lines function.

xfit <- his$mids  
 yfit <- dcauchy(xfit, location = 0, scale = 1)  
 lines(xfit, yfit, col="blue", lwd=2)  
 }

Now, we have generated histogram with Cauchy density curve using following function.

densityCurve(x, nbreaks = 600)



From the histogram with Cauchy density curve, it seems that the random sample follows standard Cauchy distribution.

1. **To study the evaluation of the empirical average of Cauchy distribution I have written the cauchyMeanForEachSample function. This function has two parameters-**

*samSize = series of sample size (for this problem 1 to ) (Default 1 to 100)*

*seed = random seed (Default NULL).*

For this problem, I have followed the following setps.

1. Generate a random sample from standard Cauchy distribution of size (say )
2. Find the mean of that sample
3. Repeat step (i) and (ii) for all (say )

Instead of using do loop, I have used apply function because it more efficient compare to do loop with respect to time. The apply function returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

# problem 1 part 2  
  
cauchyMeanForEachSample <- function(samSize = 1:100, seed = NULL){

At first, this function will check whether any value for random seed (seed) is given or not. If there is any value for seed (i.e. example seed = 123), it will set that value as random seed.

if(!is.null(seed)) set.seed(seed)

Then, it will generate random sample of size using rcauchydist function that I wrote in part 1 of this problem using Box-Muller algorithm and calculate mean of that sample. It will follow the same process for all sample sizes supplied by samSize parameter and it done by apply function which is an alternative of do loop. Finally, it will return the mean vector for all sample sizes.

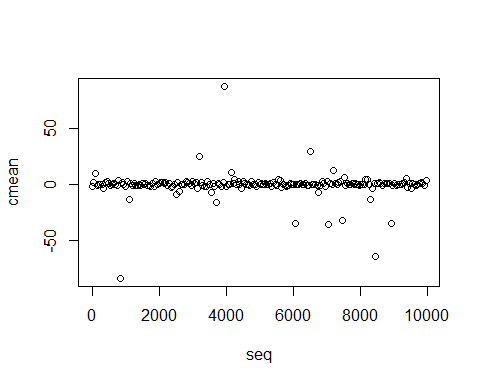
return(apply(as.array(samSize), 1, function(x)

mean(rcauchydist(x))))

}

To call this function cauchyMeanForEachSample, we need to create a sequence of integer number that will represent , the sample size. I have used seq.int function to generate a sequence from 1 to with step size 50. Then, I called cauchyMeanForEachSample function to generate mean of standard Cauchy random variable with all those sample size. I have also draw a scatter plot for those means by sample size.

seq = seq.int(1,10^4, 50)  
cmean = cauchyMeanForEachSample(seq, seed = 245)  
plot(seq, cmean)



Form this scatter diagram, we can see that the mean of the Cauchy distribution is not stable, not even for large sample size. So, even if the sample size is large, the mean of Cauchy distribution is unstable that leads us to understand why the mean of Cauchy distribution is undefined.

1. **For Monte Carlo Experiment we have followed the following algorithm-**
   1. Generate a random sample from standard Cauchy distribution of size (say )
   2. Find the mean of that sample
   3. Repeat step (i) and (ii) for (say 2000) times

I have written cauchyMeanSim function for above algorithm. This function has three parameters

*samSize = sample size (Default 100)*

*distSize = replication size (size of the distribution of mean) (default 1000)*

*seed = random seed (Default NULL).*

By following above three steps, we can generate the distribution of sample mean of standard Cauchy distribution. Instead of using do loop, I have used replicate function because it more efficient compare to do loop with respect to time. The function replicate is a wrapper for the common use of sapply for repeated evaluation of an expression (which will usually involve random number generation).

cauchyMeanSim <- function(samSize = 100, distSize = 1000 ,

seed = NULL){

At first, this function will check whether any value for random seed (seed) is given or not. If there is any value for seed (i.e. example seed = 123), it will set that value as random seed.

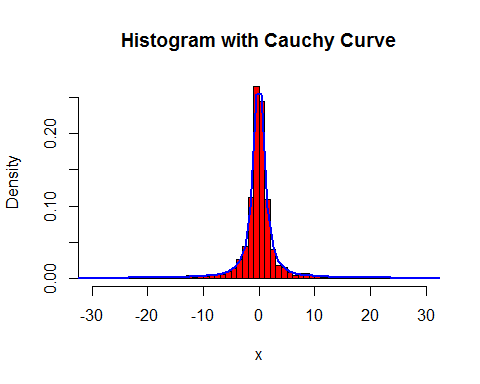
if(!is.null(seed)) set.seed(seed)

Then, it will generate random sample of size (say ) using rcauchydist function (discussed in part 1) and calculate mean of that sample. It will replicate this process for distSize (say 2000) times. For replicating the process, I have used replicate function that is an alternative of do loop. Finally, it will return the distribution of mean for .

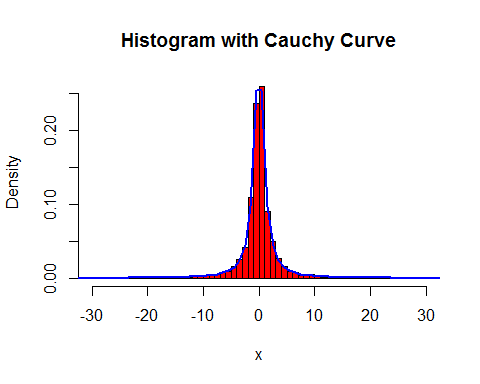
return (replicate(distSize, mean(rcauchydist(samSize))))  
}

To generate distribution of Cauchy empirical sample mean, I have called the function cauchyMeanSim with parameter samSize = 40, distSize = 2000 and seed = 365. After generating distribution of Cauchy empirical sample mean, histogram with density curve has been drawn for the distribution using densityCurve function that we wrote in part 1.

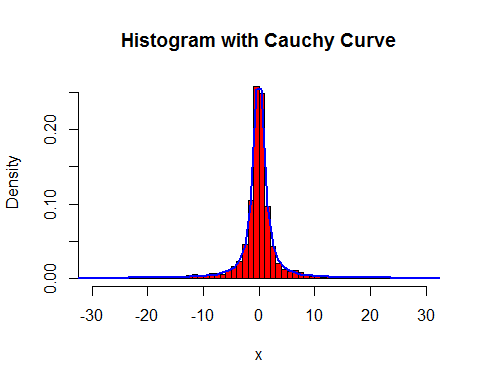
cmean = cauchyMeanSim(samSize = 40, distSize = 2000, seed = 365)  
densityCurve(cmean, nbreaks = 800)



cmean = cauchyMeanSim(100, distSize = 2000, seed = 365)  
densityCurve(cmean, nbreaks = 1400)



cmean = cauchyMeanSim(500, distSize = 2000, seed = 4125)  
densityCurve(cmean, nbreaks = 1500)



From the above three different distribution of Cauchy empirical sample mean, it seems that the distribution follows standard Cauchy distribution. Although CLT states that as sample size increases the distribution of mean follows normal distribution. However, for Cauchy distribution, the distribution of mean does not follow CLT. It follows Cauchy distribution.

## Problem 2

1. **First I will discuss the SVM code.**

For solving this problem we need “quadprog” package that is loaded by the following code.

require('quadprog')

Data import and changing of values from 2 to -1 and separating and have been done in the following code.

data <- read.table("C:/Users/ka746940/Desktop/UCF/STA 6106 - Statistical Computing/Assignments/Midterm/pb2.txt")  
data[,1][data[,1]==2] <- -1

X = data[,2:5]

mmean = colMeans(X)  
cvar = diag(var(X))  
Y = data[,1]  
X1 = scale(X) # standardized features matrix

Following algorithm has been used to this problem.

1. Created a function (rbf\_kernel) for calculating radial basis kernel
2. Created a function (poli\_kernel) for the Polynomial kernel (We will use it in part (d))
3. Created a function (kcalculator) for the kernel matrix of the data
4. Created a function (bcalculator) for calculating
5. Function (svmtrain) for fitting a SVM model
6. Function (svmpredict) for predicting new observation using the fitted SVM model

Now, we will discuss all this steps.

1. **Function (rbf\_kernel) for calculating radial basis kernel**

The form of radial basis kernel is

where, and are vectors.

In this function rbf\_kernel, there are three parameters

*x1 = first vector (for example* ***)***

*x2 = second vector (for example* ***)***

*ker\_per = value for (Default 1).*

Within this function it will convert and as a matrix and after calculating kernel value for those vectors, it will return the value.

rbf\_kernel <- function(x1,x2,ker\_par = 1){  
 x1 = as.matrix(x1)  
 x2 = as.matrix(x2)  
 return (exp(-(1/(2\*ker\_par^2))\*t(x1-x2)%\*%(x1-x2)))  
}

1. **Function (poli\_kernel) for calculating radial basis kernel**

The form of Polynomial kernel is

where, and are vectors.

In this function poli\_kernel, there are four parameters

*x1 = first vector (for example* ***)***

*x2 = second vector (for example* ***)***

*c = value for (Default 0).*

*d = value for (Default 1).*

Within this function it will calculate polynomial kernel value for , and then it will return the value.

poli\_kernel <- function(x1, x2, c = 0, d = 1){  
 return((t(as.matrix(x1)) %\*% as.matrix(x2) + c)^d)  
}

1. **Function (kcalculator) for calculating kernel matrix**

The form of Polynomial kernel is

where, and are vectors.

In this function kcalculator, there are three parameters

*X = data matrix for which we need to calculate*

*kernel = name of the kernel (Default “Gaussian”).*

ker\_par *= kernel parameter (if it is “Gaussian”, required only value of (Default 1). If*

*it is “Polynomial”, required two values for c and d).*

Within this function it will calculate based on the kernel type. First it will convert into a matrix and create a null matrix () of size 62 by 62 for this problem. Based on kernel type, I have calculated the matrix using nested loop. Finally, it will return the matrix.

kcalculator <- function(X, kernel = “Gaussian”, ker\_par = 1){  
 X=as.matrix(X)  
 N<-dim(X)[1]  
 K<-matrix(0,N,N)  
 if (toupper(kernel)== "GAUSSIAN"){  
 for(i in 1:N){  
 for(j in 1:N){  
 K[i,j]<-rbf\_kernel(X[i,],X[j,],ker\_par)  
 }  
 }  
 }  
 if (toupper(kernel)== "POLYNOMIAL"){  
 for(i in 1:N){  
 for(j in 1:N){  
 K[i,j]<-poli\_kernel(X[i,],X[j,],ker\_par[1], ker\_par[2])  
 }  
 }  
 }  
 return(t(K))  
}

1. **Function (bcalculator) for calculating value**

The form of Polynomial kernel is

We can write this equation as following in matrix form

where, indicates row wise scaler multiplication. In this function bcalculator, there are five parameters

*Y = Target vector*

*X = data matrix for which we need to calculate*

*alpha = estimated obtained from solution of dual problem*

*kernel = name of the kernel*

ker\_par *= kernel parameter (if it is “Gaussian”, required only value of . If*

*it is “Polynomial”, required two values for c and d).*

Within this function it will calculate based on the kernel type. Using above formula it will calculate .

bcalculator <- function(Y, X, alpha, kernel, ker\_par){  
 N<-length(Y)  
 K = kcalculator(X, kernel, ker\_par)  
 w01=rowSums(t((alpha\*Y)\*K))  
 return(mean(Y-w01))  
}

All the above function will be used in the following functions.

1. **Function (svmtrain) for fitting a SVM model**

In the following svmtrain function, I have trained the data using SVM model. Function contain six parameters.

*X = Feature matrix*

*Y = Target vector*

*C = upper limit of constrains*

*kernel = name of the kernel*

ker\_par *= kernel parameter (if it is “Gaussian”, required only value of . If*

*it is “Polynomial”, required two values for c and d)*

*esp = small number used for adding with Dm matrix if it is singular and selecting for .*

In this function, I have used all most the same steps as it was given. I will just discuss those steps that I have changed.

svmtrain <- function(X, Y, C=Inf, kernel = "Gaussian",

ker\_par =1.5, esp=1e-2){

N<-length(Y)  
 X<-as.matrix(X)  
 Y<-as.vector(Y)

In the following steps, I have used kcalculator function that I discuss in step (iii) to calculated matrix for the given data using given kernel and parameters. And then it will calculate the following matrix (***Dm***).

which can be written as

where indicates element wise multiplication.

K = kcalculator(X, kernel, ker\_par)  
 Dm = (Y %\*% t(Y))\*K

From here to until finding , I did not change anything in the code.

Dm<-Dm+diag(N)\*1e-8 # adding a very small number to the diag,   
 dv<-t(rep(1,N))  
 meq<-1  
 Am<-cbind(matrix(Y,N),diag(N))

# the 1 is for the sum(alpha)==0, others for each alpha\_i >= 0  
 bv<-rep(0,1+N)   
 if(C!=Inf){  
 # an upper bound is given  
 Am<-cbind(Am,-1\*diag(N))  
 bv<-c(cbind(matrix(bv,1),matrix(rep(-C,N),1)))  
 }  
 alpha\_org<-solve.QP(Dm,dv,Am,bvec=bv, meq=meq)$solution  
 indx<-which(alpha\_org>esp,arr.ind=TRUE)  
 alpha<-alpha\_org[indx]  
 nSV<-length(indx)  
 if(length(indx)==0){  
 throw("QP is not able to give a solution for these data points")  
 }  
 Xv<-X[indx,]  
 Yv<-as.vector(Y[indx])  
 w<-unname(t(Xv)%\*%(alpha\*Yv), force = TRUE)  
 # choose one of the support vector to compute b. for safety reason,  
 # select the one with max alpha

Here, I have calculated that is explained in step (iv).

b = bcalculator(Yv, Xv, alpha, kernel, ker\_par)

Finally, it will return all results as a list.

return(list(alpha=alpha, wstar=w, b=b, nSV=nSV, Xv=Xv, Yv=Yv,

kernel = kernel ,ker\_par=ker\_par))

}

1. **Function (svmpredict) for predicting new observation using the fitted SVM model**

the following svmpredict function has been used to predict observation. Function contain two parameters.

*x = new vector / matrix of feature*

*model = estimated model obtained from svmtrain* *from* *fitted SVM*

This function can predict for a single observation of features or a set of observations of features. It will first extract all parameters obtained from fitted model and then within loop for all observations, it will calculate predicted value using the following formula-

svmpredict <- function(x, model){  
 x = as.matrix(x)  
 kernel = model$kernel  
 ker\_per = model$ker\_par  
 alpha<-model$alpha  
 b<-model$b  
 Yv<-model$Yv  
 Xv<-model$Xv  
 ker\_par<-model$ker\_par  
 # wstar<-model$wstar

result = as.vector(rep(0,dim(x)[1]))

for (k in 1:dim(x)[1]){  
 sum = 0  
 if (toupper(kernel)== "GAUSSIAN"){  
 for (i in 1 : length(alpha)){  
 sum = sum + alpha[i] \* Yv[i] \* rbf\_kernel(Xv[i,],x[k,],ker\_per)  
 }  
 result[k]<-sign(sum + b)  
 }  
   
 if (toupper(kernel)== "POLYNOMIAL"){  
 for (i in 1 : length(alpha)){  
 sum = sum + alpha[i] \* Yv[i] \* poli\_kernel(Xv[i,],x[k,],

ker\_per[1], ker\_per[2])

}  
 result[k]<-sign(sum + b)  
 }  
 }  
 return(result)  
}

Following code is used to call svmtrain function to fit the model using Gaussian kernel with parameter 1.5 for sigma. It is found that the model contains 62 support vector for the data. For Gaussian kernel, I did not standardize the data.

model =svmtrain(X, Y, kernel = "Gaussian", ker\_par = 1.5)  
model

## $alpha  
## [1] 1.05015 0.95028 0.89218 1.05019 1.05009 1.04974 0.7327245  
## [8] 1.05016 1.05016 0.83702 1.04863 1.06079 0.15644 0.7565102  
## [15] 0.70499 0.73640 1.05030 0.98065 1.04300 0.99209 1.0560719  
## [22] 1.06243 1.05014 1.03911 0.91115 0.95395 0.77806 0.9744232  
## [29] 0.97079 1.05017 0.78069 0.94982 0.94947 0.93265 0.9498211  
## [36] 0.94983 0.94981 0.94968 0.93274 0.94969 0.93294 0.9497462  
## [43] 0.94983 0.94985 0.94929 0.89190 0.96229 0.95306 0.9497318  
## [50] 0.75674 0.94983 0.81435 0.94784 0.95307 0.93468 0.9498302  
## [57] 0.81176 0.94996 0.94981 0.94983 0.94983 0.94983  
##   
## $wstar  
## [,1]  
## [1,] 101.601256  
## [2,] 56.720330  
## [3,] 286.630550  
## [4,] 8.467797  
##   
## $b  
## [1] -0.05016975  
##   
## $nSV  
## [1] 62  
##   
##   
## $kernel  
## [1] "Gaussian"  
##   
## $ker\_par  
## [1] 1.5

Following code is used to predict and calculate training Error for the SVM model with Polynomial and Gaussian kernel. It can be seen that the training error for this data is 0 for both model.

p1 = svmpredict(X1, model1)  
cat("Training Error", (length(Y)-sum(Y==p1))/length(Y)\*100, "%")

## Training Error 0 %

p2 = svmpredict(X, model)  
cat("Training Error", (length(Y)-sum(Y==p2))/length(Y)\*100, "%")

## Training Error 0 %

1. **Impact of**

Following code is used for different to see impact of number of support vectors. First, I have generated a sequence of gamma values and then a null matrix of size length of sigma by three where I will store training error, number of support vector and sigma.

# part b problem 2  
  
sigm = seq(.5, 100, length = 1000)  
itsig = matrix(0, length(sigm), 3)

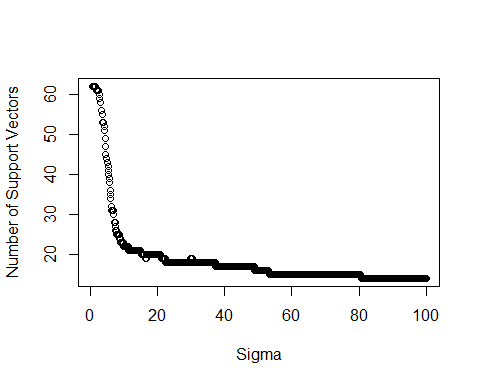
Then I have used a loop to fit a model for each of the of the gamma value. And for each of the fitted model I calculated training error and number of support vector and then stored these values in a matrix.

for (i in 1:length(sigm)){  
 fit = svmtrain(X, Y, kernel = "Gaussian", ker\_par = sigm[i])  
 pred = svmpredict(X, fit)  
 error = (length(Y)-sum(Y==pred))/length(Y)\*100  
 nSV = fit$nSV  
 itsig[i,] = c(error, nSV, sigm[i])  
}

I plotted number of support vector by sigma and the following figure shows that as the sigma increases the number of support vector decreases. On the other hand, we can say for smaller sigma value it is over fitting the data. Since the number of support vector increases for lower sigma, the model complicity increases.

plot(itsig[,3], itsig[,2], xlab = "Sigma",

ylab = "Number of Support Vectors")



1. **Classification of Observation**

For , I have used svmpredict function for both kernel and it is predicted as 1 for both cases.

a = matrix(c(18,17,33,26), 1,4,byrow = TRUE)  
a1 = (a-mmean)/cvar  
p1 = svmpredict(a1, model1)  
cat("Prediction", p1)

## Prediction 1

p2 = svmpredict(a, model)  
cat("Prediction", p2)

## Prediction 1

1. **SVM model using Polynomial kernel**

The SVM code that I discussed in part (a) also included polynomial kernel. Here I will just include polynomial kernel function and prediction function.

* **Function (poli\_kernel) for calculating radial basis kernel**

The form of Polynomial kernel is

where, and are vectors.

In this function poli\_kernel, there are four parameters

*x1 = first vector (for example* ***)***

*x2 = second vector (for example* ***)***

*c = value for (Default 0).*

*d = value for (Default 1).*

Within this function it will calculate polynomial kernel value for , and then it will return the value.

poli\_kernel <- function(x1, x2, c = 0, d = 1){  
 return((t(as.matrix(x1)) %\*% as.matrix(x2) + c)^d)  
}

* **Function (svmpredict) for predicting new observation using the fitted SVM model**

the following svmpredict function has been used to predict observation. Function contain two parameters.

*x = new vector / matrix of feature*

*model = estimated model obtained from svmtrain* *from* *fitted SVM*

This function can predict for a single observation of features or a set of observations of features. It will first extract all parameters obtained from fitted model and then within loop for all observations, it will calculate predicted value using the following formula-

svmpredict <- function(x, model){

Converting input feature vector as matrix.

x = as.matrix(x)

Assigning variable for different parameters obtained from the model.

kernel = model$kernel  
 ker\_per = model$ker\_par  
 alpha<-model$alpha  
 b<-model$b  
 Yv<-model$Yv  
 Xv<-model$Xv  
 ker\_par<-model$ker\_par

Initializing null vector of size equal length of the input feature.

result = as.vector(rep(0,dim(x)[1]))

Loop for each of the input feature vector. Within the loop it will assign a null sum variable so that we can use it for iterative sum of the above equation. After that it will check which kernel is being used when it was fitted the model. Based on the kernel it will start another loop for calculation sum of the above equation. Finally, it find the sign of the equation and store it in result vector.

for (k in 1:dim(x)[1]){  
 sum = 0  
 if (toupper(kernel)== "GAUSSIAN"){  
 for (i in 1 : length(alpha)){  
 sum = sum + alpha[i] \* Yv[i] \* rbf\_kernel(Xv[i,],x[k,],ker\_per)  
 }  
 result[k]<-sign(sum + b)  
 }  
   
 if (toupper(kernel)== "POLYNOMIAL"){  
 for (i in 1 : length(alpha)){  
 sum = sum + alpha[i] \* Yv[i] \* poli\_kernel(Xv[i,],x[k,],

ker\_per[1], ker\_per[2])

}  
 result[k]<-sign(sum + b)  
 }  
 }  
 return(result)  
}

Following code is used to call svmtrain function to fit the model using Polynomial kernel with parameter 23 and 2 for and respectively. It is found that the model contains only 15 support vector for the data. For Polynomial kernel, I have standardized the data, otherwise the kernel matrix becomes singular.

model1 = svmtrain(X1, Y, kernel = "Polynomial", ker\_par = c(23,2))  
model1

## $alpha  
## [1] 0.94426 12.759816 2.977131 0.495124 6.5785068 4.6269065  
## [7] 17.05089 13.167360 1.661623 9.151191 1.9643374 14.0507365  
## [13] 29.10916 5.499686 0.48651  
##   
## $wstar  
## [,1]  
## [1,] 0.13176131  
## [2,] -0.05720521  
## [3,] 0.23166994  
## [4,] -0.19237407  
##   
## $b  
## [1] -2.113031  
##   
## $nSV  
## [1] 15  
##   
## $Xv  
## V2 V3 V4 V5  
## [1,] 0.26995486 0.50746227 0.279433139 -1.78899530  
## [2,] -0.34994149 -0.70264007 -1.615472836 -1.36238873  
## [3,] -0.34994149 -2.39678334 0.008732286 -0.08256901  
## [4,] -0.03999331 -1.91274240 1.091535700 -1.14908544  
## [5,] -1.27978602 -0.21859913 -0.397318995 -1.14908544  
## [6,] 1.19979939 1.47554414 1.091535700 1.41055399  
## [7,] 0.57990304 0.26544180 -0.803370275 -1.36238873  
## [8,] -0.96983784 0.02342134 0.414783566 0.13073427  
## [9,] 0.88985122 1.23352367 -0.126618141 -0.29587230  
## [10,] -0.34994149 0.26544180 -1.886173690 -1.78899530  
## [11,] -0.03999331 -1.67072194 -1.209421556 0.55734084  
## [12,] 0.57990304 -0.46061960 -0.803370275 -1.78899530  
## [13,] -0.03999331 0.50746227 0.279433139 0.13073427  
## [14,] -2.20963055 -1.91274240 -0.397318995 -0.93578216  
## [15,] -2.51957873 -2.39678334 -2.156874544 -2.00229859  
##   
## $Yv  
## [1] 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1  
##   
## $kernel  
## [1] "Polynomial"  
##   
## $ker\_par  
## [1] 23 2

## Problem 3

1. **We have**

(1)

The Lagrangian of this problem can be written as-

(2)

for and

1. **Dual Problem**

Differentiate (2) with respect to , and and set it to 0, we have

(3)

(4)

(5)

Plugging (3), (4) and (5) into (2), we have

So the dual problem is-

(6)

1. **KKT Conditions**

The standard KKT conditions contain three conditions:

* Dual feasibility
* Primal feasibility
* Complementary slackness

Let be the solutions of dual problem, the optimal vector is

from (3) (7)

(8)

(9)

Equation (9) implies that, if

So, the for which are the support vectors. Since (from 7), it follows that and this implies that .

Again, if then , implies that and .

Similarly, if , then , so that by (9) and .

So,

1. **Strong Duality:**

To check strong duality, Slater’s condition states that

Alternatively, a weak version of Slater’s condition can be used. This is satisfied if the primal problem is convex and are affine.

In this case, the primal is

which is convex and the constraints is affine. So, Slater’s conditions are met and strong duality holds.

1. **Solution of primal using Dual solution:**

Suppose, we have solution of dual problem. Since, the strong duality holds for this problem, we can write

So, the margin of the optimal hyperplane can be written as

1. **Code for solving Dual and Primal:**

In this problem, I have generated using normal distribution. We have that has fifteen +1 values and ten -1 values. For the matrix, I have used the following distribution

For

For

For this data set, I have used the following code to generate the data.

require('quadprog')  
X = as.list(numeric())  
for (j in 1:15){  
 x2 = rnorm(4, 2, 1)  
for (i in 1:4){  
 x2 = cbind(x2,rnorm(4, 2, 1))  
}  
 x2=unname(as.matrix(x2), force = TRUE)  
 X[j]=list(x2)  
}  
for (j in 16:25){  
 x2 = rnorm(4, 0, 1)  
 for (i in 1:4){  
 x2 = cbind(x2,rnorm(4, 0, 1))  
 }  
 x2=unname(as.matrix(x2), force = TRUE)  
 X[j]=list(x2)  
}  
Y = c(rep(1,15), rep(-1,10))

The Gaussian kernel for this problem can be written as

In this case, I input two matrixes in the function rbf\_kernel and took sum of diagonal elements of the matrix multiplication of two matrixes.

## Defining the Gaussian kernel  
rbf\_kernel <- function(x1,x2,gamma){  
 x1 = as.matrix(x1)  
 x2 = as.matrix(x2)  
 return(exp(-(1/gamma^2)\*sum(diag(t(x1 - x2) %\*% (x1 - x2)))))  
}

**The rest of the code for this problem is exactly the same as the SVM code that I discussed earlier (Problem 2 part 1.**

kcalculator <- function(X, ker\_par){  
 X=as.matrix(X)  
 N<-dim(X)[1]  
 K<-matrix(0,N,N)  
 for(i in 1:N){  
 for(j in 1:N){  
 K[i,j]<-rbf\_kernel(X[i,][[1]],X[j,][[1]],ker\_par)  
 }  
 }  
 return(K)  
}  
  
  
bcalculator <- function(Y, X, alpha, ker\_par){  
 N<-length(Y)  
 K = kcalculator(X, ker\_par)  
 w01=rowSums((alpha\*Y)\*K)  
 w0 = mean(Y-w01)  
   
}  
  
  
svmtrain <- function(X, Y, C=Inf, ker\_par =1.5, esp=1e-2){  
 N<-length(Y)  
 X<-as.matrix(X)  
 Y<-as.vector(Y)  
   
 K = kcalculator(X, ker\_par)  
 Dm = (Y %\*% t(Y))\*K  
 Dm<-Dm+diag(N)\*1e-8 # adding a very small number to the diag, some trick  
 dv<-t(rep(1,N))  
 meq<-1  
 Am<-cbind(matrix(Y,N),diag(N))  
 bv<-rep(0,1+N) # the 1 is for the sum(alpha)==0, others for each alpha\_i >= 0  
 if(C!=Inf){  
 # an upper bound is given  
 Am<-cbind(Am,-1\*diag(N))  
 bv<-c(cbind(matrix(bv,1),matrix(rep(-C,N),1)))  
 }  
 alpha\_org<-solve.QP(Dm,dv,Am,bvec=bv, meq=meq)$solution  
 indx<-which(alpha\_org>esp,arr.ind=TRUE)  
 alpha<-alpha\_org[indx]  
 nSV<-length(indx)  
 if(length(indx)==0){  
 throw("QP is not able to give a solution for these data points")  
 }  
 Xv<-X[indx,]  
 Yv<-as.vector(Y[indx])  
 # choose one of the support vector to compute b. for safety reason,  
 # select the one with max alpha  
   
 b = bcalculator(Yv, Xv, alpha, ker\_par)  
   
 return(list(alpha=alpha, b=b, nSV=nSV, Xv=Xv, Yv=Yv,

ker\_par=ker\_par))

}

The SMM model has been fitted here. We found that the model contains 8 support vector.

model1 = svmtrain(X, Y, ker\_par = 34)  
model1

## $alpha  
## [1] 2.83081 9.54869 3.703201 0.475271 2.058594 2.807872  
## [7] 1.72966 4.97916 10.032167 4.676705  
##   
## $b  
## [1] 0.07056687  
##   
## $nSV  
## [1] 10  
##   
## $Xv  
## $Xv[[1]]  
## [,1] [,2] [,3] [,4] [,5]  
## [1,] 1.2766464 2.307676 4.1086038 1.5102182 1.8509717  
## [2,] 2.5317520 1.707989 2.5577176 1.9055271 2.3988965  
## [3,] 0.6136538 2.700019 -0.2695060 0.7065571 0.3818654  
## [4,] 3.3966935 2.196149 0.3507233 1.7059178 2.6703605  
##   
## $Xv[[2]]  
## [,1] [,2] [,3] [,4] [,5]  
## [1,] 2.548059 1.594153 2.734204 2.3776449 1.3360105  
## [2,] 1.145736 1.139862 1.429038 0.4529139 1.1068188  
## [3,] 3.606087 3.374920 1.494517 1.2060917 1.2396867  
## [4,] 1.146121 2.731299 1.818344 1.1037879 0.4944776  
##   
## $Xv[[3]]  
## [,1] [,2] [,3] [,4] [,5]  
## [1,] 1.259820 1.482540 1.3345772 0.7837194 1.8026087  
## [2,] 2.846331 2.111328 0.8638959 2.5455845 0.9488197  
## [3,] 1.672740 1.015993 1.0385627 2.9192863 3.2443124  
## [4,] 3.883639 2.760249 1.6311673 1.3756332 1.5976370  
##   
## $Xv[[4]]  
## [,1] [,2] [,3] [,4] [,5]  
## [1,] 4.1771549 0.7644139 2.125775 0.4092327 -0.4025023  
## [2,] 0.5261820 2.5675718 1.805588 2.6635420 1.8849839  
## [3,] 0.5141084 1.2837233 2.670735 1.3729260 2.1790900  
## [4,] 3.5100643 2.7053427 3.162000 0.5145675 1.7085410  
##   
## $Xv[[5]]  
## [,1] [,2] [,3] [,4] [,5]  
## [1,] -0.02327673 2.061803 2.0332523 2.2556539 4.086404  
## [2,] 2.38537636 2.485626 0.7441299 0.8790429 2.693246  
## [3,] 1.73229441 2.306513 2.0555231 0.4803739 3.177325  
## [4,] 0.89529552 3.069487 2.8120416 -0.3876563 1.379278  
##   
## $Xv[[6]]  
## [,1] [,2] [,3] [,4] [,5]  
## [1,] 1.724797 3.759377 -0.3284305 1.8243546 1.557033  
## [2,] 1.965682 2.300920 2.0182555 0.8052183 1.399350  
## [3,] 1.279355 1.679359 0.8176136 3.4631539 4.104422  
## [4,] 2.637948 1.160669 2.3701292 0.9671931 1.332345  
##   
## $Xv[[7]]  
## [,1] [,2] [,3] [,4] [,5]  
## [1,] -0.6825388 0.3508535 0.35382368 0.1739108 1.26609274  
## [2,] 0.8660750 -2.6451010 -0.41199194 -0.1135327 3.14512108  
## [3,] 0.1911760 -0.2251995 0.44475163 -0.1442088 -0.08616853  
## [4,] -1.1984620 1.1777434 0.06905578 0.5021829 -1.38045626  
##   
## $Xv[[8]]  
## [,1] [,2] [,3] [,4] [,5]  
## [1,] -0.09891229 -0.3906270 -0.2242652 1.1506056 1.02774047  
## [2,] 1.71779791 1.4420060 -0.1485043 -1.3639470 0.44152061  
## [3,] -1.18050249 0.2212948 -0.7403536 -0.5461495 -0.07605167  
## [4,] 1.24413209 1.0441486 1.4833411 0.6674507 -1.82642915  
##   
## $Xv[[9]]  
## [,1] [,2] [,3] [,4] [,5]  
## [1,] 0.1616069 -0.04411519 1.5318480 0.24459638 0.5834051  
## [2,] -1.2121039 0.27060355 0.3856968 0.03750932 0.4570691  
## [3,] 0.2247229 0.13958473 -1.2683299 0.99605694 0.8666696  
## [4,] 1.3962083 0.46092650 0.4458176 -0.42601609 -1.5199806  
##   
## $Xv[[10]]  
## [,1] [,2] [,3] [,4] [,5]  
## [1,] -0.14511286 -1.3291719 0.1768102 0.4227011 2.0788844  
## [2,] -0.12430348 2.1225674 0.8222449 0.1153580 -0.6609476  
## [3,] 0.04622189 0.9462414 -1.2129077 1.1752454 1.7207594  
## [4,] -0.88521430 1.2174459 -1.6514694 -0.3970321 -0.1979674  
##   
##   
## $Yv  
## [1] 1 1 1 1 1 1 -1 -1 -1 -1  
##   
## $ker\_par  
## [1] 34

##   
## $margin  
## [1] 0.1527794  
##   
## $primal  
## [1] 42.84209

Using the following formula we can calculate the primal problem as –

I found that the solution for the primal is 42.84. and the margin of optimal hyperplane is 0.1527794.

1. **Prediction code:**

The prediction code is exactly the same as before that I explained in part c of problem 2 except I just converted the list of matrix into matrix within matrix.

svmpredict <- function(x,model){  
 x = as.matrix(x)  
 kernel = model$kernel  
 ker\_per = model$ker\_par  
 alpha<-model$alpha  
 b<-model$b  
 Yv<-model$Yv

Here, I converted the list of matrix into matrix within matrix.

Xv<-as.matrix(model$Xv)  
 ker\_par<-model$ker\_par  
 # wstar<-model$wstar  
 result = as.vector(rep(0,dim(x)[1]))  
 for (k in 1:dim(x)[1]){  
 sum = 0  
 for (i in 1 : length(alpha)){  
 sum = sum + alpha[i] \* Yv[i] \* rbf\_kernel(Xv[i,][[1]],

x[k,][[1]],ker\_per)

}  
 result[k]<-sign(sum + b)  
 }  
 margin = sum(alpha)  
 return(list(result = result, margin = sqrt(1/margin), primal = margin))  
}  
  
  
svmpredict(X, model1)

## $result  
## [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

It is found that for this model the training error is 0% for this data.