## Problem 1

1. To simulate random variable form the following probability distribution using Metropolis-Hastings algorithm, two python functions “norm\_mix” and “rand\_norm\_mix” have been written. Function “norm\_mix” calculates the probability for the random variables of the given distribution and “rand\_norm\_mix” generates the random number for the following mixture distribution using Metropolis-Hastings algorithm.

(1)

In following codes were used to import some python library.

# -\*- coding: utf-8 -\*-

"""

Created on Wed Nov 23 21:08:43 2016

@author: Kanak

"""

import numpy as np

import matplotlib.pyplot as plt

from scipy.stats import norm

“norm\_mix” function calculate the probability for the normal location mixture distribution using equation (1).

def norm\_mix(data, mu, sigma, p):

a = p\*norm(mu[0], sigma[0]).pdf(data)+(1-p)\*norm(mu[1],

sigma[1]).pdf(data)

return(a)

“rand\_norm\_mix” function generate random sample for the normal location mixture distribution (Equation 1) using Metropolis-Hastings algorithm with proposal distribution . The algorithm is given bellow.

**Algorithm:**

After getting initial random number, I have repeated the following procedure.

1. Proposal distribution
2. The initial is considered that is given in question.
3. Repeated the following steps times
   1. Generate from
   2. Generate from
   3. If , we have accepted Y for and append to *X*; otherwise is considered for .

The python code is given bellow.

def rand\_norm\_mix(n, mu, sigma, p, pro\_par, x1 = None):

if not x1 == None:

x = []

x.append(x1)

else:

x = []

x.append(norm(0, pro\_par).rvs(1)[0])

u = np.random.uniform(size = n)

for i in range(1, n):

xt = x[i-1]

y = norm(xt, pro\_par).rvs(1)[0]

num = norm\_mix(y, mu, sigma, p)\*norm(y, pro\_par).pdf(xt)

den = norm\_mix(xt, mu, sigma, p)\*norm(xt, pro\_par).pdf(y)

if (u[i] <= num / den):

x.append(y)

else:

x.append(xt)

return (np.array(x))

Now, For each of the initial value, I have generated 10000 observation from normal location mixture distribution (Equation 1) by calling the function “rand\_norm\_mix” using given parameter with given proposal distribution. Then, for each of the initial value, I have plotted to see the sample path.

n = 10000

mu = [7, 10]

sigma = [.5, .5]

p = 0.7

pro\_par = 0.01

x0 = [0,7,15]

b=1000

indexlow = b

indexup = n

x = []

for i in range(0, len(x0)):

x.append(rand\_norm\_mix(n, mu, sigma, p, pro\_par, x0[i]))

x = np.array(x)

fig ,axs=plt.subplots(len(x0),1, figsize=(10, 50))

for i in range(0,len(x0)):

axs[i].plot(range(indexlow, indexup), x[i,indexlow:indexup])

plt.show()

plt.clf()

plt.cla()

plt.close()

fig ,axs=plt.subplots(len(x0),1, figsize=(10, 50))

for i in range(0,len(x0)):

prob = norm\_mix(x[i,indexlow:indexup], mu, sigma, p)

myHist = axs[i].hist(x[i, indexlow:indexup], 40, normed=True)

xx = np.arange(min(x[i, indexlow:indexup])-2, max(x[i,indexlow:indexup])+2,0.001)

prob = norm\_mix(xx, mu, sigma, p)

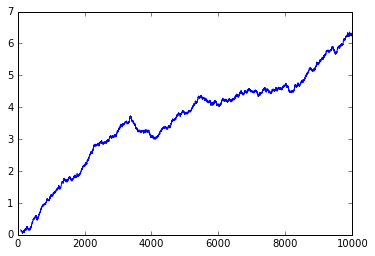
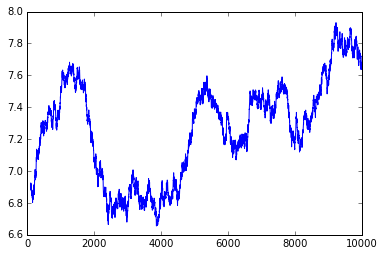
h = axs[i].plot(xx, prob, lw=2)

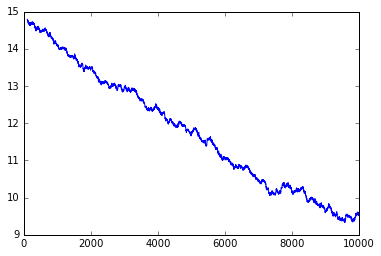
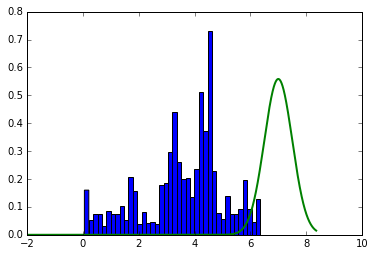
plt.show()

plt.clf()

plt.cla()

plt.close()

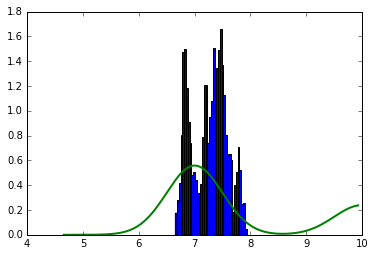
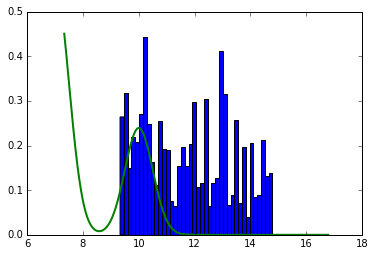
 

Figure 1: Sample path of generated sample (first three plots) and histogram with probability density plot (last three plots)

From the above figures, we can see that the sample path for the generated samples show some trends or pattern which indicate that the generated sample is not converging. The histogram with probability plot also shows that the generated samples are not sowing the same distribution pattern as the true probability distribution. Which also indicates that the samples generated with the given proposal distribution is not conversing. **However, if we can select best proposal distribution we can have the better sample. In the next part I have generated samples with the given initial values but with different proposal distribution.**

1. Here, I have generated three samples with the given initial values. However, since it is shown that for the given proposal distribution, the sample path does not converge, that is way I have used as the proposal distribution to show the histogram with true probability density curve.

n = 10000

p = 0.7

pro\_par = 1

x0 = [0,7,15]

b=1000

indexlow = b

indexup = n

x = []

for i in range(0, len(x0)):

x.append(rand\_norm\_mix(n, mu, sigma, p, pro\_par, x0[i]))

x = np.array(x)

fig ,axs=plt.subplots(len(x0),1, figsize=(10, 50))

for i in range(0,len(x0)):

axs[i].plot(range(indexlow, indexup), x[i,indexlow:indexup])

plt.show()

plt.clf()

plt.cla()

plt.close()

fig ,axs=plt.subplots(len(x0),1, figsize=(10, 50))

for i in range(0,len(x0)):

prob = norm\_mix(x[i,indexlow:indexup], mu, sigma, p)

myHist = axs[i].hist(x[i, indexlow:indexup], 40, normed=True)

xx = np.arange(min(x[i, indexlow:indexup])-2, max(x[i,indexlow:indexup])+2,0.001)

prob = norm\_mix(xx, mu, sigma, p)

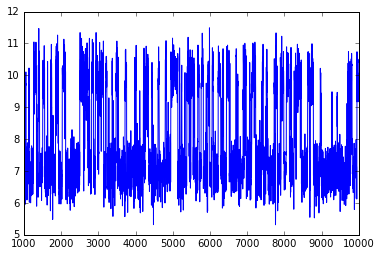
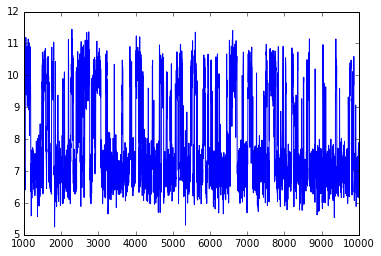
h = axs[i].plot(xx, prob, lw=2)

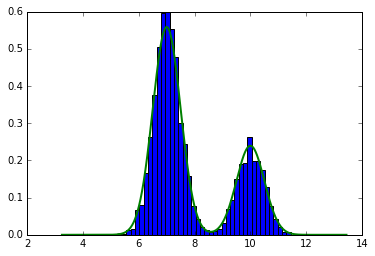
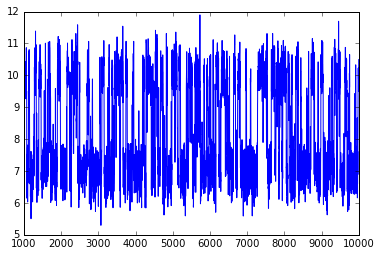
plt.show()

plt.clf()

plt.cla()

plt.close()





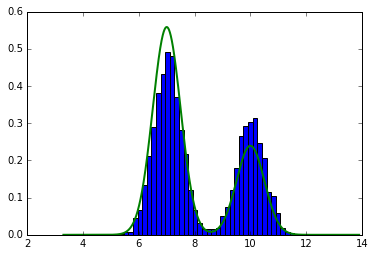
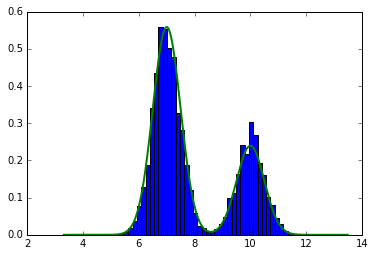


Figure 2: Sample path of generated sample (first three plots) and histogram with probability density plot (last three plots)

From the above plots, it is seen that the sample path plots are not showing any pattern which indicates that the samples generated using proposal distribution is converging. The histograms with density plots also show that the samples are fitting with the true probability distribution.

So, we can say that to generate a random sample using Metropolis-Hastings algorithm, it important to select an appropriate proposal distribution. And also, initial value of the chain is also important. As we can see from figure 1 that the distribution is different for different initial values. For initial value 7, it shows better probability distribution compare to other two initial values. Because, initial value 7 is one of the location of the mixture normal distribution that is the center of distribution.

1. “gelman\_rubin” function has been used to calculate given in Section 9.4 by Rizzo.

def gelman\_rubin(x):

if not isinstance(x, np.ndarray):

try:

x = np.array(x)

except:

print ("x is not possible to convert as array")

return

n = x.shape[1]

x\_means = np.mean(x, axis = 1)

b = n \* np.var(x\_means)

x\_w = np.var(x, axis = 1)

W = np.mean(x\_w)

v\_hat = W\* (n-1) / n + (b/n)

r\_hat = np.sqrt(v\_hat / W)

return(r\_hat)

In the following python code, I have generated 10000 random sample with given three initial value and proposal distribution . I have considered this proposal because it is shown in part (a) of this question that for the given proposal distribution the generated chain does not converge**. In following part of this question I will show the G-R convergence path for different proposal distribution.**

n = 10000

mu = [7, 10]

sigma = [.5, .5]

p = 0.7

pro\_par = 1

x0 = [0,7,15]

b=100

indexlow = b

indexup = n

x = []

for i in range(0, len(x0)):

x.append(rand\_norm\_mix(n, mu, sigma, p, pro\_par, x0[i]))

cumsumx = np.cumsum(x, axis = 1)

ncol = np.array(range(1, x.shape[1]+1))

cumsumx = cumsumx / ncol

fig ,axs=plt.subplots(len(x0),1, figsize=(10, 50))

for i in range(0, len(x0)):

axs[i].plot(range(0, n-b), cumsumx[i, b:n])

plt.show()

plt.clf()

plt.cla()

plt.close()

rhat = []

xrange = np.array(range(b, n))

for i in xrange:

rhat.append(gelman\_rubin(cumsumx[:,0:i]))

rhat = np.array(rhat)

h = 1.2

condi = xrange[rhat<1.2]

if (sum(condi) == 0):

lst = n

else:

lst = min(condi)

plt.plot(xrange,rhat)

plt.axvline(x=lst, c="red", linewidth=0.5,zorder=0)

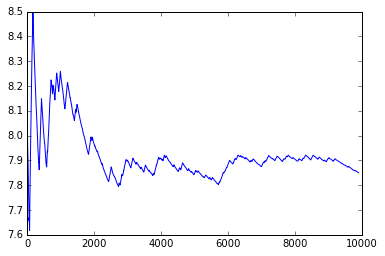
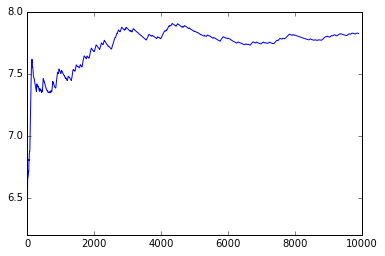
plt.axhline(y=h, c="red", linewidth=0.5,zorder=0)

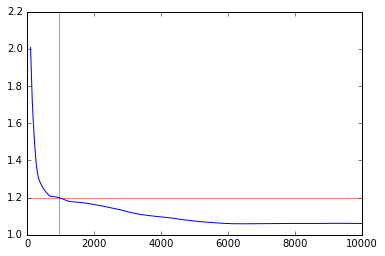
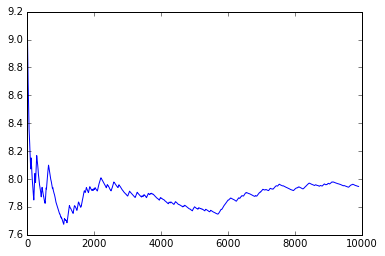
plt.show()

plt.clf()

plt.cla()

plt.close()





Burning numbers 882

Figure 3: Sample chain path of generated samples (first three plots) and G-R convergence chart (last plot)

**Comment**: The line graph shows that at the first few iteration there is a trend and very high volatility in the sequence. We consider those iteration as a burning sequence.

From the Figure 3, it is seen that different initial value shows different pattern of the distribution. So, it depends on the initial value. If we cannot consider appropriate initial value it might not converge. However, for the proposal distribution , the chain converges at (G-R statistics less than 1.2) at 882 iteration.

It is seen that the convergence depends on the parameter of the proposal distribution as well as the initial value. I have investigated this in the following. I have considered the given three initial value and [0.1, .5, 1, 2, 5] for standard deviation of the proposal distribution.

n = 10000

mu = [7, 10]

sigma = [.5, .5]

p = 0.7

pro\_par = [0.1, .5, 1, 2, 5]

x0 = [0,7,15]

b=200

indexlow = b

indexup = n

listn = []

for j in range(0, len(pro\_par)):

x = []

for i in range(0, len(x0)):

x.append(rand\_norm\_mix(n, mu, sigma, p, pro\_par[i], x0[i]))

x = np.array(x)

cumsumx = np.cumsum(x, axis = 1)

ncol = np.array(range(1, x.shape[1]+1))

cumsumx = cumsumx / ncol

rhat = []

xrange = np.array(range(b, n))

for i in xrange:

rhat.append(gelman\_rubin(cumsumx[:,0:i]))

rhat = np.array(rhat)

h = 1.2

condi = xrange[rhat<1.2]

if (sum(condi) == 0):

lst = n

else:

lst = min(condi)

listn.append(lst)

plt.plot(xrange,rhat)

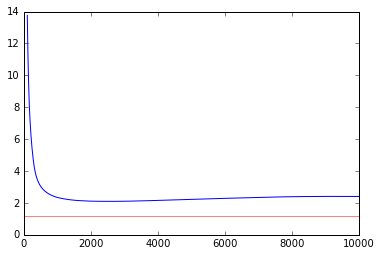
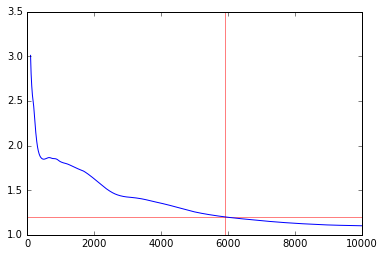
plt.axvline(x=lst, c="red", linewidth=0.5,zorder=0)

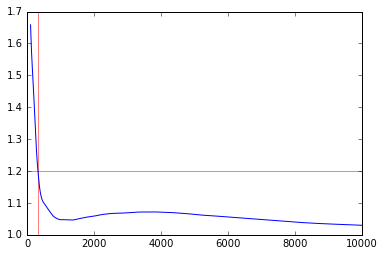
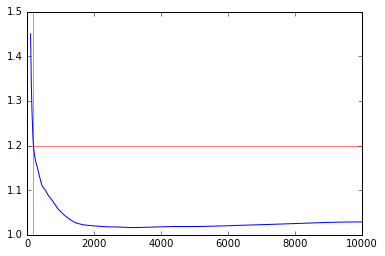
plt.axhline(y=h, c="red", linewidth=0.5,zorder=0)

plt.show()

print("Burning numbers", lst)

print(listn)

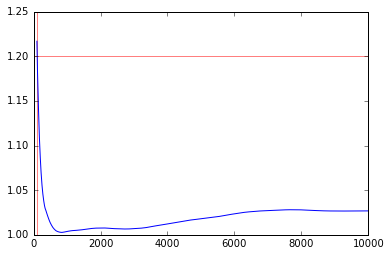


Figure 3: G-R convergence charts for different proposal distribution

# output

# Standard deviations [0.1, .5, 1, 2, 5]

# [10000, 5917, 322, 186, 108]

The burning sample number for the different standard deviation of the proposal distribution are given above. As we can see, the Standard deviation increases the burning number decreases. It does not mean if we increase more we will get less burning number of samples. From the above figures, we can see that for proposal distribution the chain does not converge even with 10000 samples. However, if we increase standard deviation the chain converges and the burning number converges. So we can say that the appropriate proposal distribution is important to generate sample using Metropolis-Hastings algorithm.

## Problem 2

1. For the MLE parameter estimation of logistic regression, the probability function, Likelihood function, Gradient function and Hessian function are given below.

Let, be the input matrix, be the column vector of , be the N-vector of fitted probabilities with element , be an diagonal matrix of weights with the ith diagonal element element .

Probability:

(1)

Likelihood function:

(2)

Gradient function:

(3)

Hessian matrix:

(4)

import numpy as np

import pandas as pd

from numpy.linalg import inv

# Data Import

dt = pd.read\_csv("C:/Users/ka746940/Desktop/UCF/STA 6106 - Statistical Computing/Assignments/Final/Final\_chao/challenge.csv");

dt = np.array(dt)

The following function “probability” has been used to calculate Probability using equation (1).

# Probability Calculation

def probability(theta, x):

prob = (1.0/(1.0+np.exp(-np.dot(x, theta))))

return prob

The following function “log\_likelihood” has been used to calculate likelihood value using equation (2).

#Get Log Likelihood of Dataset

def log\_likelihood(Y,p):

loglikelihood = Y\*np.log(p) + (1-Y)\*np.log(1-p)

return -1\*loglikelihood.sum()

The following function “gradient\_hessian” has been used to calculate gradient function and hessian matrix using equation (3) and (4).

# Get Gradient and Hessian Matrix

def gradient\_hessian(X,error, W):

gred = (np.dot(error,X))

hessi = np.dot(X.T,np.dot(W,X))

return {"gradient": gred, "hessian": hessi}

The following function “newton\_raphson” has been used to optimize the parameter of the logistic equation using following algorithm.

**Algorithm:**

1. Consider initial beta as
   1. Compute by using equation (1) with function “probability”
   2. Compute the diagonal matrix . The ith diagonal element is
   3. Compute error as
   4. Compute gradient and hessian matrix using equation (3) and (4) respectively with function “gradient\_hessian”
   5. Compute “” as matrix multiplication of and
   6. Update as
2. Continue step (a) to (f) until maximum iteration “maxite” or absolute maximum “” exceed the given threshold.
3. After the iteration it will check whether it exceed the maximum iteration or not. If it exceeds maximum iteration, it will print a message “Failed to converge” and exit.
4. If the maximum iteration does not exceed and parameter estimates converge with the given threshold it will return following results.

Output Returned:

1. Parameter estimates,
2. Log-Likelihood for the estimated parameters,
3. Probability for given using estimated parameters,
4. Estimated error,
5. Gradient matrix,
6. Hessian Matrix
7. Number of iteration it took to converge

# Newton-Raphson Optimization

def newton\_raphson(Y, X, theta0 = None, tol = 1e-6, maxite = 200):

if not isinstance(X, np.ndarray) or not isinstance(Y, np.ndarray):

try:

X = np.array(X)

Y = np.array(Y)

except:

print ("X is not possible to convert as array")

return

ncol = X.shape[1]

if (theta0 == None):

theta = np.repeat(0, ncol)

else:

theta = theta0

itter = 0

conv = 999999999

while (itter < maxite and conv> tol):

itter = itter + 1

p = probability(theta, X)

W = np.diag(p)

error = Y - p

gre\_heis = gradient\_hessian(X, error, W)

update = np.dot(inv(gre\_heis["hessian"]), gre\_heis["gradient"])

theta = theta + update

conv = max(np.absolute(update))

if (itter == maxite):

print("Failed to converge")

return

else:

p = probability(theta, X)

likeli = log\_likelihood(Y, p)

error = Y - p

W = np.diag(p)

gr\_hes = gradient\_hessian(X, error, W)

return({"prem": theta, "gradient": gr\_hes["gradient"], \

"Hessian": gr\_hes["hessian"], "error": error, \

"iteration": itter, "likelihood": likeli, "probability": p})

The following function “feature\_mat” has been used to construct feature matrix using following procedure.

1. It will take the whole data set and . variable should be in the first column.
2. Then it will extract and from the data and it will add a new column of 1’s at the first column of feature matrix .

def feature\_mat(dt):

X = []

for i in range(len(dt)):

X.append([1,dt[i,1]])

Y = dt[:,0]

return(np.array(X), np.array(Y))

X, Y = feature\_mat(dt)

After running the “newton\_raphson” for the given data set, I have got the following results.

model = newton\_raphson(Y, X, maxite = 2000)

**Output:**

{'error': array([ 0.060752, 0.140683, 0.171155, 0.397319, -0.430493,

-0.37472423, -0.37472423, -0.37472423, -0.32209402, -0.27362104,

-0.22996825, -0.22996825, 0.77003175, 0.77003175, -0.15804912,

-0.12954604, -0.08554358, 0.91445642, -0.0690441 , -0.0690441 ,

-0.04454057, -0.03564143, -0.0227033 ]),

'gradient': array([ 4.71051168e-07, 2.66590121e-05]),

'prem': array([ 15.04289723, -0.23216268]),

'probability': array([ 0.939247, 0.859316, 0.828844, 0.602680, 0.430493,

0.37472423, 0.37472423, 0.37472423, 0.32209402, 0.27362104,

0.22996825, 0.22996825, 0.22996825, 0.22996825, 0.15804912,

0.12954604, 0.08554358, 0.08554358, 0.0690441 , 0.0690441 ,

0.04454057, 0.03564143, 0.0227033 ]),

'Hessian': array([[ 6.99999953e+00, 4.45999973e+02],

[ 4.45999973e+02, 2.87436492e+04]]),

'likelihood': 10.157596343933617,

'iteration': 78}

So, the MLE estimate of and using Newton-Raphson algorithm with ‘Iterative Reweighted Least Squares’ are 15.04289723 and -0.23216268 respectively.

1. To find the predicted probability of O-ring failure, follwong “pred\_prob” python function is used. The algorithm to calculate the predicted probability is given below.
2. It will take new observations and the estimated model obtained from “newton\_raphson” as imput
3. It will convert new observation as numpy array if it is not a numpy array.
4. It will extract estimated parameter from the model that is given as input
5. Using “probability” function described above (equation 1), it will calculate probability and return the predicted probability

def pred\_prob(x, model):

if not isinstance(x, np.ndarray):

try:

x = np.array(x)

except:

print ("x is not possible to convert as array")

return

prem = model["prem"]

pred = probability(prem,x)

return({"pred": pred})

Also, I have plotted the scatter diagram with fitted line and error.

plt.scatter(X[:,1], Y, label='Scatter Plot')

plt.plot(X[:,1], model["probability"], 'k', label='Fitted Line', lw = 2, color = 'r')

for i in range(len(Y)):

if(i == 0):

plt.plot([X[i,1], X[i,1]], [model["probability"][i], Y[i]], 'k', label = "Error", color = 'g', lw = 2)

else:

plt.plot([X[i,1], X[i,1]], [model["probability"][i], Y[i]], 'k', color = 'g', lw = 2)

legend = plt.legend(loc='best')

plt.show()

plt.clf()

plt.cla()

plt.close()

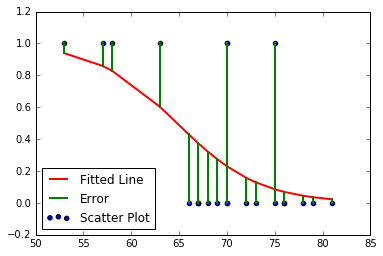


Figure 1: Scatter diagram with fitted line and error

To find the predicted probability of O-ring failure at 310 F, we can use the following code.

pred\_prob([1, 31], model)

# {'pred': 0.99960878191994684}

**Comment:**

It can be seen from the estimated value that the average estimated probability of O-ring failure is almost 1 which indicates that it is almost certain that at 310 F temperature, O-ring will definite fail.

## Problem 3

## Part 1

1. To minimize the given function using Newton’s method, I have used objective function, Gradient-Hessian and Newton’s python functions.

Objective function:

(1)

Gradient function:

(2)

Hessian matrix:

(3)

The following function “gxy” has been used to calculate objective function using equation (1).

def gxy(x):

return(4\*x[0]\*x[1]+(x[0]+x[1]\*\*2)\*\*2)

The following function “gradient\_hessian” has been used to calculate gradient and hessian matrix using equation (2) and (3).

def gradient\_hessian(X):

"""Calculate gradient and Hessian"""

gred1 = 4\*X[1]+2\*(X[0]+X[1]\*\*2)

gred2 = 4\*X[0]+4\*X[1]\*(X[0]+X[1]\*\*2)

gred = np.array([gred1, gred2])

hessi11 = 2.0

hessi12 = 4\*X[1]+4

hessi21 = 4\*X[1]+4

hessi22 = 4\*X[0]+12\*X[1]\*\*2

hessi = np.array([[hessi11, hessi12], [hessi21, hessi22]])

return {"gradient": gred, "hessian": hessi}

The following function “newton\_raphson” has been used to optimize the parameter of the objective function using following algorithm.

Algorithm:

1. Consider initial value of and as which is vector
   1. Compute gradient and hessian matrix using equation (2) and (3) respectively with function “gradient\_hessian”
   2. Compute “” as matrix multiplication of and
   3. Update as
2. Continue step (a) to (c) until maximum iteration “maxite” or absolute maximum “” exceed the given threshold.
3. After the iteration it will check whether it exceed the maximum iteration or not. If it exceeds maximum iteration, it will print a message “Failed to converge” and exit.
4. If the maximum iteration does not exceed and parameter estimates converge with the given threshold it will return following results.

Output Returned by the function:

1. Parameter estimates, which contains two value for and
2. Value of for the estimated value of
3. Gradient matrix, at
4. Hessian Matrix at
5. Number of iteration it took to converge

def newton\_raphson(func, f\_gred, X = None, tol = 1e-6, maxite = 200):

if (X == None):

X = np.repeat(0, 2)

if not isinstance(X, np.ndarray):

try:

X = np.array(X, dtype='f')

except:

print ("X is not possible to convert as array")

return

update = 999999

conv = update

itter = 0

while (itter < maxite and conv> tol):

itter = itter + 1

gre\_heis = f\_gred(X)

update = np.dot(inv(gre\_heis["hessian"]), gre\_heis["gradient"])

X = X - update

conv = max(np.absolute(update))

if (itter == maxite):

print("Failed to converge")

return

else:

g = func(X)

gr\_hes = f\_gred(X)

return({"prem": X, "gradient": gr\_hes["gradient"], \

"Hessian": gr\_hes["hessian"], "iteration": itter, "objective": g})

After running the “newton\_raphson” for the given initial value, I have got the following results.

newton\_raphson(gxy, gradient\_hessian, [1,1])

**Output:**

{'Hessian': array([[ 2. , 1.33333333],

[ 1.33333333, 8.88888889]]),

'gradient': array([ 0., 0.]),

'iteration': 7,

'objective': -0.592592592592593,

'prem': array([ 0.88888889, -0.66666667])}

1. To minimize the given function using steepest decent method with golden section search, I have used objective function, Gradient-Hessian and Newton’s python functions.

Objective function:

(1)

Gradient function:

(2)

Parameter Update function:

The following function “gradient” has been used to calculate gradient function using equation (2).

def gradient(X):

"""Calculate gradient"""

gred1 = 4\*X[1]+2\*(X[0]+X[1]\*\*2)

gred2 = 4\*X[0]+4\*X[1]\*(X[0]+X[1]\*\*2)

gred = np.array([gred1, gred2])

return gred

The following function “steep\_des\_GS” has been used to optimize the parameter of the objective function using following algorithm. **It is important to mention that for golden section search, I have defined a range for the search area and it will consider the same search area at every iteration.**

Algorithm:

1. Within this “steep\_des\_GS” function, I first use a function “update” to calculate parameters using equation
2. Copy search range into so that we can use it in every iteration
3. Consider initial value of and as which is vector
4. Compute gradient matrix using equation (2) with function “gradient”
5. Compute and , where
6. Compute and
   1. If
      1. Then and
   2. Else if
      1. Then and
   3. Continue step (i) and (ii) until
7. Compute “” as
8. Update as
9. Reset and as and , so that for the next iteration it can use the given search area again
10. Continue step (4) to (9) until maximum iteration “maxite” or absolute maximum “” exceed the given threshold.
11. After the iteration it will check whether it exceed the maximum iteration or not. If it exceeds maximum iteration, it will print a message “Failed to converge” and exit.
12. If the maximum iteration does not exceed and parameter estimates converge with the given threshold it will return following results.

Output Returned by the function:

1. Parameter estimates,
2. Value of for the estimated value of
3. Gradient matrix,
4. Number of iteration it took to converge

def steep\_des\_GS(func, f\_gred, x, a, b, tol = 1e-6, maxite = 200):

def update(X, gred, alpha):

upd = X - alpha \* gred

return(upd)

if not isinstance(x, np.ndarray):

try:

x = np.array(x, dtype='f')

except:

print ("X is not possible to convert as array")

return

bb = b

aa = a

upd = (a+b)/2\*f\_gred(x)

tau = (np.sqrt(5)-1.0)/2

itter = 0

while (np.sqrt(np.dot(upd, upd)) > tol and itter < maxite):

a = aa

b = bb

gred = f\_gred(x)

x1 = a + (1.0-tau)\*(b-a)

x2 = a + tau \* (b-a)

f1 = func(update(x, gred, x1))

f2 = func(update(x, gred, x2))

while((b-a) > tol):

if (f1 > f2):

a = x1

x1 = x2

f1 = f2

x2 = a + tau \* (b-a)

f2 = func(update(x, gred, x2))

else:

b = x2

x2 = x1

f2 = f1

x1 = a + (1.0-tau) \* (b-a)

f1 = func(update(x, gred, x1))

upd = ((a+b)/2.0)\*gred

x = x - upd

itter += 1

if (itter == maxite):

print("Failed to converge")

return

else:

g = func(x)

gr\_hes = f\_gred(x)

return({"prem": x, "gradient": gr\_hes, "iteration": itter, \

"objective": g})

After running the “steep\_des\_GS” for the given initial value, I have got the following results.

steep\_des\_GS(gxy, gradient, np.array([1,1]), 0, 10)

**Output:**

{'gradient': array([ -7.26853731e-07, 4.79966659e-07]),

'iteration': 23,

'objective': -0.59259259259240249,

'prem': array([ 0.88888845, -0.66666655])}

**Comment**: As we can see that both method obtained the same parameter estimates with the same initial values. However, steepest descent method with golden section search takes 23 iteration while Newton’s method takes 7 iteration. The advantage of using steepest descent method with golden section search is that it does not need any Hessian function of the objective function.

## Problem 3

## Part 2

1. To maximize the given function using Newton’s method, I have used objective function, Gradient-Hessian and Newton’s python functions.

Objective function:

(1)

Gradient function:

(2)

Hessian matrix:

(3)

The following function “gxy” has been used to calculate objective function using equation (1).

def gxy(x):

return(np.log(x)/(1+x))

The following function “gradient\_hessian” has been used to calculate gradient and hessian function using equation (2) and (3).

def gradient\_hessian(X):

"""Calculate gradient and Hessian"""

gred = np.array(((1+X)/X-np.log(X))/((1+X)\*\*2))

hessi = np.array(((-X\*\*(-2) - X\*\*(-1)) \* (1 + X)\*\*2 - 2 \* (1 + X) \* ((1 + X) / X - np.log(X))) / ((1 + X)\*\*4))

return {"gradient": gred, "hessian": hessi}

The following function “newton\_raphson” has been used to optimize the parameter of the objective function using following algorithm.

**Algorithm:**

1. Consider initial value of as
   1. Compute gradient and hessian using equation (2) and (3) respectively with function “gradient\_hessian”
   2. Compute “” as matrix multiplication of and
   3. Update as
2. Continue step (a) to (c) until maximum iteration “maxite” or absolute maximum “” exceed the given threshold.
3. After the iteration it will check whether it exceed the maximum iteration or not. If it exceeds maximum iteration, it will print a message “Failed to converge” and exit.
4. If the maximum iteration does not exceed and parameter estimates converge with the given threshold it will return following results.

Output Returned by the function:

1. Parameter estimates,
2. Value of for the estimated value of
3. Gradient matrix, at
4. Hessian at
5. Number of iteration it took to converge

def newton\_raphson(func, f\_gred, X, tol = 1e-6, maxite = 200):

if not isinstance(X, np.ndarray):

try:

X = np.array(X, dtype='f')

except:

print ("X is not possible to convert as array")

return

update = 999999

conv = update

itter = 0

while (itter < maxite and conv> tol):

itter = itter + 1

gre\_heis = f\_gred(X)

update = np.dot((1./gre\_heis["hessian"]), gre\_heis["gradient"])

X = X - update

conv = np.absolute(update)

if (itter == maxite):

print("Failed to converge")

return

else:

g = func(X)

gr\_hes = f\_gred(X)

return({"prem": X, "gradient": gr\_hes["gradient"], \

"Hessian": gr\_hes["hessian"], "iteration": itter, "objective": -g})

After running the “newton\_raphson” for the given initial value, I have got the following results.

newton\_raphson(gxy, gradient\_hessian, [1])

**Output:**

{'Hessian': array([-0.01688967], dtype=float32),

'gradient': array([ 0.], dtype=float32),

'iteration': 9,

'objective': array([-0.27846453], dtype=float32),

'prem': array([ 3.59112144], dtype=float32)}

1. To maximize the given function using secant method, I have used objective function, Gradient and Newton’s python functions.

Objective function:

(1)

Gradient function:

(2)

Hessian function:

(3)

The following function “gradient” has been used to calculate gradient function using equation (2).

def gradient(X):

"""Calculate gradient."""

gred = np.array(((1 + X) / X - np.log(X)) / ((1 + X)\*\*2))

return (gred)

The following function “secant\_opt” has been used to optimize the parameter of the objective function using following algorithm. **It is important that optimization using secant method needs two initial values.**

**Algorithm:**

1. Consider initial values of as which is vector
   1. Compute gradient using equation (2) with function “gradient” for each value of and store it to “gre\_heis0” and “gre\_heis1”
   2. Compute hessian using equation (3) as
   3. Compute “” as matrix multiplication of and
   4. Store
   5. Update as
2. Continue step (a) to (e) until maximum iteration “maxite” or absolute maximum “” exceed the given threshold.
3. After the iteration it will check whether it exceed the maximum iteration or not. If it exceeds maximum iteration, it will print a message “Failed to converge” and exit.
4. If the maximum iteration does not exceed and parameter estimates converge with the given threshold it will return following results.

Output Returned by the function:

1. Parameter estimates,
2. Value of for the estimated value of
3. Gradient matrix, at
4. Number of iteration it took to converge

def secant\_opt(func, f\_gred, X, tol = 1e-6, maxite = 200):

if not isinstance(X, np.ndarray):

try:

X = np.array(X, dtype='f')

except:

print ("X is not possible to convert as array")

return

update = 999999999

itter = 0

conv = update

while (itter < maxite and conv> tol):

itter = itter + 1

gre\_heis0 = f\_gred(X[0])

gre\_heis1 = f\_gred(X[1])

hessi = ((X[1] - X[0])/(gre\_heis1-gre\_heis0))

update = hessi\*gre\_heis1

X[0] = X[1]

X[1] = X[0] - update

conv = np.absolute(update)

if (itter == maxite):

print("Failed to converge")

return

else:

g = func(X[1])

gre\_heis = f\_gred(X[1])

return({"prem": X[1], "gradient": gre\_heis, \

"iteration": itter, "objective": -g})

After running the “newton\_raphson” for the given initial value, I have got the following results.

secant\_opt(gxy, gradient, [3, 4])

**Output:**

{'gradient': array(-1.3401678529359155e-09),

'iteration': 6,

'objective': -0.27846453364463852,

'prem': 3.5911217}

**Comment**: As we can see that both method obtained the same parameter estimates. However, secant method takes 6 iteration while Newton’s method takes 9 iteration. The advantage of using secant method is that it does not need any Hessian function of the objective function. But, the initial parameter should be close to the optimal value. For all these optimization method, it is important that the function should be a log concave.

## Problem 4

1. **SVM With Gaussian, Polynomial, Marr, Marlet wavelet, Morlet-RBF kernel**

In this problem, I have used the same R code that I used in the midterm exam. I just change the kernel function so that we can use this code for any of this kernel. When we will call the “svmtrain” function, we need to mention the kernel name. it will use that kernel for the SVM model fit.

require('quadprog')

## Loading required package: quadprog

data <- read.table("C:/Users/ka746940/Desktop/UCF/STA 6106 - Statistical Computing/Assignments/Final/pb22.txt")  
#data <- read.table("D:/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 = as.data.frame(scale(X))

The function “kernel\_cal” contains all the kernel function. Rest of the code is same as the midterm code.

kernel\_cal <- function(x1,x2,ker\_par, kernel = "GAUSSIAN"){  
 if(toupper(kernel)=="GAUSSIAN"){   
 ## Defining the Gaussian kernel  
 K<-exp(-(1/(ker\_par^2))\*t(x1-x2)%\*%(x1-x2))  
 return(K)  
   
 } else if(toupper(kernel)=="POLYNOMIAL"){   
 ## Defining the Polinomial kernel  
 K<- (t(as.matrix(x1)) %\*% as.matrix(x2) + ker\_par[1])^ker\_par[2]  
 return(K)  
   
 } else if(toupper(kernel)=="MARR"){   
 ## Defining the Marr wavelet kernel  
 K<- prod((1-((x1-x2)/ker\_par)^2)\*exp(-1/2\*((x1-x2)/ker\_par)^2))  
 return(K)  
   
 } else if(toupper(kernel)=="MORLET"){   
 ## Defining the Morlet wavelet kernel  
 K<- prod((cos(1.75\*(x1-x2)/ker\_par))\*exp(-1/2\*((x1-x2)/ker\_par)^2))  
 return(K)  
   
 } else if(toupper(kernel)=="MORLET-RBF"){   
 ## Defining the Morlet-RBF wavelet kernel  
 k1 = prod((cos(1.75\*(x1-x2)/ker\_par[1]))\*exp(-1/2\*((x1-x2)/ker\_par[1])^2))  
 K<- exp(-ker\_par[2]\*(2-2\*k1))  
 return(K)  
   
 }  
}  
  
  
  
  
  
kcalculator <- function(X, kernel, 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]<-kernel\_cal(X[i,],X[j,],ker\_par, kernel = kernel)  
 }  
 }  
 return(K)  
}  
  
bcalculator <- function(Y, X, alpha, kernel, ker\_par){  
 N<-length(Y)  
 K = kcalculator(X, kernel, ker\_par)  
 w01=rowSums((alpha\*Y)\*K)  
 w0 = mean(Y-w01)  
   
}  
  
svmtrain <- function(X, Y, C=Inf, kernel = "Gaussian", ker\_par =1.5, esp=1e-2){  
 if(!(toupper(kernel) %in% c("GAUSSIAN","POLYNOMIAL", "MARR", "MORLET", "MORLET-RBF"))){  
 cat("There is no Kernel named ", kernel)  
 return()  
 }  
 N<-length(Y)  
 X<-as.matrix(X)  
 Y<-as.vector(Y)  
   
 K = kcalculator(X, kernel, ker\_par)  
 Dm = (Y %\*% t(Y))\*K  
 Dm<-Dm+diag(N)\*1e-5   
 dv<-t(rep(1,N))  
 meq<-1  
 Am<-cbind(matrix(Y,N),diag(N))  
 bv<-rep(0,1+N)   
 if(C!=Inf){  
 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){  
 stop("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)  
 b = bcalculator(Yv, Xv, alpha, kernel, ker\_par)  
   
 return(list(alpha=alpha, wstar=w, b=b, nSV=nSV, Xv=Xv, Yv=Yv, kernel = kernel ,ker\_par=ker\_par))  
}  
  
### Predict the class of an object X  
  
  
  
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  
 for (i in 1 : length(alpha)){  
 s1 = alpha[i] \* Yv[i] \* kernel\_cal(Xv[i,],x[k,],ker\_per, kernel)  
 sum = sum + s1  
 }  
 result[k]<-sign(sum + b)  
 }  
 return(result)  
}

Here, different SVM models are fitted with fixed parameter to predict .

model =svmtrain(X, Y, kernel = "gaussian", ker\_par = 2.5)  
model1 = svmtrain(X1, Y, kernel = "Polynomial", ker\_par = c(23,2))  
model2 = svmtrain(X, Y, kernel = "marr", ker\_par = 5)  
model3 = svmtrain(X, Y, kernel = "morlet", ker\_par = 10)  
model4 = svmtrain(X, Y, kernel = "MORLET-RBF", ker\_par = c(10,2))  
p = svmpredict(X, model)  
cat("Training Erro", (length(Y)-sum(Y==p))/length(Y)\*100, "%")

## Training Erro 0 %

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

## Training Erro 0 %

p2 = svmpredict(X, model2)  
cat("Training Erro", (length(Y)-sum(Y==p2))/length(Y)\*100, "%")

## Training Erro 0 %

p3 = svmpredict(X, model3)  
cat("Training Erro", (length(Y)-sum(Y==p3))/length(Y)\*100, "%")

## Training Erro 0 %

p4 = svmpredict(X, model4)  
cat("Training Erro", (length(Y)-sum(Y==p4))/length(Y)\*100, "%")

## Training Erro 0 %

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

## Prediction 1

pr1 = svmpredict(a1, model1)  
cat("Prediction", pr1)

## Prediction 1

pr2 = svmpredict(a, model2)  
cat("Prediction", pr2)

## Prediction 1

pr3 = svmpredict(a, model3)  
cat("Prediction", pr3)

## Prediction 1

pr4 = svmpredict(a, model4)  
cat("Prediction", pr4)

## Prediction 1

Here, different SVM models are fitted with fixed parameter to predict . From the above results, it can be seen that for each of the model the training error is 0. And the prediction is 1 for the given **.**

**Using Cross-Validation Greed Search For Finding C And Sigma With Minimum Support Vector**

**Algorithm:**

1. Divide sample into two parts training set and validation set using 75% and 25% rules.
   1. Generate set of combination of C and Sigma
      1. Fit SVM model on Training set using C and Sigma
      2. Predict class using validation set on the fitted model
      3. Compute validation and training error
   2. Repeat step (a) to (c) for all combination of C and Sigma generated in step (2) and store results
   3. Find minimum validation error and find minimum support vectors for that minimum validation error
   4. For that minimum support vector, find the corresponding C and Sigma
2. Repeat step (a) to (d) for each kernel

For the above algorithm, R code is given below.

kk = c("GAUSSIAN", "MARR", "MORLET")  
require(caTools)

## Loading required package: caTools

set.seed(101)   
sample = sample.split(Y, SplitRatio = .75)  
X\_tr = subset(X1, sample == TRUE)  
X\_ts = subset(X1, sample == FALSE)  
Y\_tr = subset(Y, sample == TRUE)  
Y\_ts = subset(Y, sample == FALSE)   
   
cseq = seq(2, 500, length = 10)  
kerseq = seq(.1, 200, length = 100)  
combbi = expand.grid(cseq, kerseq)  
kkmin = list(kernel = character(), NSVM = numeric(), C = numeric(), Sigma = numeric(), Tr\_e = numeric(), Ts\_e = numeric())  
for (j in 1:length(kk)){  
 min\_k = numeric()  
 t\_e = numeric()  
 ts\_e = numeric()  
 for (i in 1:dim(combbi)[1]){  
 min\_k1 = svmtrain(X=X\_tr, Y=Y\_tr, C=combbi[i,1], kernel = kk[j], ker\_par =combbi[i,2])  
 min\_k[i] = min\_k1$nSV  
 p = svmpredict(X\_tr, min\_k1)  
 t\_e[i] = (length(Y\_tr)-sum(Y\_tr==p))/length(Y)\*100  
 t\_p = svmpredict(X\_ts, min\_k1)  
 ts\_e[i] = (length(Y\_ts)-sum(Y\_ts==t\_p))/length(Y)\*100  
 }  
 pp = which(ts\_e == min(ts\_e))  
 pp1 = which(min\_k == min(min\_k[pp]))  
 pp2 = combbi[pp1,][which(combbi[pp1,1] == min(combbi[pp1,1])),]  
 kkmin$kernel[j] = kk[j]  
 kkmin$NSVM[j] = min\_k[pp1[1]]  
 kkmin$C[j] = pp2[1,1]  
 kkmin$Sigma[j] = pp2[1,2]  
 kkmin$Ts\_e = ts\_e[pp[1]]  
 kkmin$Tr\_e = t\_e[pp[1]]  
}  
kkmin

## $kernel  
## [1] "GAUSSIAN" "MARR" "MORLET"   
## $NSVM  
## [1] 13 26 25  
##   
## $C  
## [1] 112.66667 57.33333 2.00000  
##   
## $Sigma  
## [1] 4.138384 24.330303 4.138384  
## $Tr\_e  
## [1] 4.83871  
##   
## $Ts\_e  
## [1] 3.225806

for (i in 1:3){  
 trn = svmtrain(X=X1, Y=Y, C=kkmin$C[i], kernel = kkmin$kernel[i], ker\_par =kkmin$Sigma[i])  
 p = svmpredict(X1, trn)  
 cat("Training Erro", (length(Y)-sum(Y==p))/length(Y)\*100, "%")  
 a = matrix(c(18,17,33,26), 1,4,byrow = TRUE)  
 a1 = scale(a, center = mmean, scale = cvar)  
 cat("Prediction", svmpredict(a1, trn))  
}

## Training Error 3.225806 %

## Prediction 1

## Training Error 12.90323 %

## Prediction 1

## Training Error 9.677419 %

## Prediction 1

**Comment**: Using the above algorithm, we can see that the minimum support vectors are 13, 26, 25 for Gaussian, Marr and Morlet kernel respectively. Using these parameters I have fitted SVM model using each kernel and found that all the models predicted the given as +1. Also, for those values of C and sigma the training errors are 3.2%, 12.9% and 9.6% for Gaussian, Marr and Morlet kernel respectively. It is important to mention that due to time constrain, I did not consider larger number of combination of C and Sigma. That this way, above results might not the exactly correct for this problem.

1. **SMM With Gaussian, Marlet wavelet kernel**

In this problem, I have used the same R code that I used in the midterm exam. I just change the kernel function so that we can use this code for both kernels. When we will call the “smmtrain” function, we need to mention the kernel name, it will use that kernel for the SMM model fit.

require('quadprog')  
set.seed(0)  
X = as.list(numeric())  
for (j in 1:15){  
 x2 = rnorm(5, 2, 1)  
for (i in 1:4){  
 x1 = rnorm(5, 2, 1)  
 x2= cbind(x2, x1)  
}  
 x2=unname(as.matrix(x2), force = TRUE)  
 X[j]=list(x2)  
}  
for (j in 16:25){  
 x2 = rnorm(5, 0, 1)  
 for (i in 1:4){  
 x1 = rnorm(5, 0, 1)  
 x2= cbind(x2, x1)  
 }  
 x2=unname(as.matrix(x2), force = TRUE)  
 X[j]=list(x2)  
}  
Y = c(rep(1,15), rep(-1,10))

The function “kernel\_cal” contains all the kernel function. Rest of the code is same as the midterm code.

kernel\_cal <- function(x1,x2,ker\_par, kernel = "GAUSSIAN"){  
 if(toupper(kernel)=="GAUSSIAN"){   
 ## Defining the Gaussian kernel  
 return(exp(-(1/ker\_par^2)\*sum(diag(t(x1 - x2) %\*% (x1 - x2)))))  
 } else if(toupper(kernel)=="MORLET"){   
 ## Defining the Morlet-RBF wavelet kernel  
 K = prod((cos(1.75\*(x1-x2)/ker\_par))\*exp(-1/2\*((x1-x2)/ker\_par)^2))  
 return(K)  
 }  
}  
  
kcalculator <- function(X, ker\_par, kernel){  
 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]<-kernel\_cal(X[i,][[1]],X[j,][[1]],ker\_par=ker\_par, kernel=kernel)  
 }  
 }  
 return(K)  
}  
  
bcalculator <- function(Y, X, alpha, ker\_par, kernel){  
 N<-length(Y)  
 K = kcalculator(X, ker\_par, kernel)  
 w01=rowSums((alpha\*Y)\*K)  
 w0 = mean(Y-w01)  
}  
  
smmtrain <- function(X, Y, C=Inf, ker\_par =1.5, kernel = "GAUSSIAN", esp=1e-2){  
 N<-length(Y)  
 X<-as.matrix(X)  
 Y<-as.vector(Y)  
   
 K = kcalculator(X, ker\_par, kernel)  
 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, kernel)  
   
 return(list(alpha=alpha, b=b, nSV=nSV, Xv=Xv, Yv=Yv, ker\_par=ker\_par, kernel = kernel))  
}  
  
### Predict the class of an object X

Here, SMM model has been fitted with fixed parameter.

model1 = smmtrain(X, Y, ker\_par = 34, kernel = "MORLET")  
model1

## $alpha  
## [1] 0.3607885 2.0851569 2.3837827 0.2432445 1.8180380 0.8637219 0.5945789  
## [8] 2.0521292 0.4960765 3.2160802 1.2877765 0.1080913  
##   
## $b  
## [1] 0.03427105  
##   
## $nSV  
## [1] 12  
##   
##   
## $Yv  
## [1] 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1  
##   
## $ker\_par  
## [1] 34  
##   
## $kernel  
## [1] "MORLET"

smmpredict <- function(x,model){  
 x = as.matrix(x)  
 kernel = model$kernel  
 ker\_per = model$ker\_par  
 alpha<-model$alpha  
 b<-model$b  
 Yv<-model$Yv  
 Xv<-as.matrix(model$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] \* kernel\_cal(Xv[i,][[1]],x[k,][[1]],ker\_per, kernel)  
 }  
 result[k]<-sign(sum + b)  
 }  
 margin = sum(alpha)  
 return(list(pred = result, margin = sqrt(1/margin), primal = margin, kernel = kernel))  
}

Here, SMM model has been fitted with fixed parameter.

p = smmpredict(X, model1)

cat("Training Erro", (length(Y)-sum(Y==p$pred))/length(Y)\*100, "%")

## $pred  
## [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 -1 -1  
## [24] -1 -1  
##   
## $margin  
## [1] 0.2539227  
##   
## $primal  
## [1] 15.50947  
##   
## $kernel  
## [1] "MORLET"

## Training Error 0 %

**Using Cross-Validation Greed Search For Finding C And Sigma With Minimum Support Vector**

**Algorithm:**

1. Divide sample into two parts training set and validation set using 75% and 25% rules.
   1. Generate set of combination of C and Sigma
      1. Fit SMM model on Training set using C and Sigma
      2. Predict class using validation set on the fitted model
      3. Compute validation and training error
   2. Repeat step (a) to (c) for all combination of C and Sigma generated in step (2) and store results
   3. Find minimum validation error and find minimum support vectors for that minimum validation error
   4. For that minimum support vector, find the corresponding C and Sigma
2. Repeat step (a) to (d) for each kernel

For the above algorithm, R code is given below.

kk = c("GAUSSIAN", "MORLET")  
require(caTools)  
set.seed(101)   
sample = sample.split(Y, SplitRatio = .75)  
X\_tr = subset(X, sample == TRUE)  
X\_ts = subset(X, sample == FALSE)  
Y\_tr = subset(Y, sample == TRUE)  
Y\_ts = subset(Y, sample == FALSE)   
  
cseq = seq(2, 500, length = 5)  
kerseq = seq(.1, 200, length = 5)  
combbi = expand.grid(cseq, kerseq)  
kkmin = list(kernel = character(), NSVM = numeric(), C = numeric(), Sigma = numeric(), Tr\_e = numeric(), Ts\_e = numeric())  
for (j in 1:length(kk)){  
 min\_k = numeric()  
 t\_e = numeric()  
 ts\_e = numeric()  
 for (i in 1:dim(combbi)[1]){  
 min\_k1 = smmtrain(X=X\_tr, Y=Y\_tr, C=combbi[i,1], kernel = kk[j], ker\_par =combbi[i,2])  
 min\_k[i] = min\_k1$nSV  
 p = smmpredict(X\_tr, min\_k1)  
 t\_e[i] = (length(Y\_tr)-sum(Y\_tr==p$pred))/length(Y)\*100  
 t\_p = smmpredict(X\_ts, min\_k1)  
 ts\_e[i] = (length(Y\_ts)-sum(Y\_ts==t\_p$pred))/length(Y)\*100  
 }  
 pp = which(ts\_e == min(ts\_e))  
 pp1 = which(min\_k == min(min\_k[pp]))  
 pp2 = combbi[pp1,][which(combbi[pp1,1] == min(combbi[pp1,1])),]  
 kkmin$kernel[j] = kk[j]  
 kkmin$NSVM[j] = min\_k[pp1[1]]  
 kkmin$C[j] = pp2[1,1]  
 kkmin$Sigma[j] = pp2[1,2]  
 kkmin$Ts\_e = ts\_e[pp[1]]  
 kkmin$Tr\_e = t\_e[pp[1]]  
}  
  
kkmin

## $kernel  
## [1] "GAUSSIAN" "MORLET"   
##   
## $NSVM  
## [1] 9 10  
##   
## $C  
## [1] 251.0 126.5  
##   
## $Sigma  
## [1] 200.000 50.075  
##   
## $Tr\_e  
## [1] 0  
##   
## $Ts\_e  
## [1] 0

for (i in 1:2){  
 trn = smmtrain(X=X, Y=Y, C=kkmin$C[i], kernel = kkmin$kernel[i], ker\_par =kkmin$Sigma[i])  
 p = smmpredict(X, trn)  
 cat("Training Erro", (length(Y)-sum(Y==p$pred))/length(Y)\*100, "%")  
 a = matrix(c(18,17,33,26), 1,4,byrow = TRUE)  
 a1 = as.matrix(list(X[[1]]))  
 cat("Prediction", smmpredict(a1, trn)$pred)  
   
}

## Training Error 0 %

## Prediction 1

## Training Error 0 %

## Prediction 1

**Comment**: Using the above algorithm, we can see that the minimum support vectors are 9, 10 for Gaussian, and Morlet kernel respectively. Using these parameters, I have fitted SMM model using each kernel and found that all the models predicted the given as +1. Also, for those values of C and sigma the training errors are 0% for both kernel. It is important to mention that due to time constrain, I did not consider larger number of combination of C and Sigma. That this way, above results might not the exactly correct for this problem.