Midterm - ENPM 808A

Date: 11 Nov, 2020

GOVIND AJITH KUMAR
VID: 166994 88

Exercise 3.10

Ans:

(a) We are given that

$$e_n(w) = \max(0, -y_n w^T x_n)$$

Let us randomly consider a point.

Now if $0 > -y_n w^T x_n$,

then $e_n(w) = 0$

But for this to happen: $y_n w^T x > 0$ (This means correctly classified.)

must be observed. Hence

$$\nabla e_n(w) = 0$$

Now, if $e_n(w) = -y_n w^T x$,

then: $0 < -y_n w^T x$,

$\Rightarrow y_n w^T x < 0$ (This means misclassification)

Now, by looking at the definition of stochastic gradient descent (SGD):

$$w(t+1) = w(t) - \eta \nabla e_n(w)$$

As we saw above, 2 separate cases exist: $\eta = 1$ and,

$$\nabla e_1(w) = 0 \quad \& \quad \nabla e_n(w) = -y_n x_n$$

If we substitute both of these, we get:

Equation ①: $w(t+1) = w(t) - 0$

Equation ②: $w(t+1) = w(t) + y_n x_n$

This is basically what PLA looks like as well, thus proving that PLA can be viewed as SGD on e_n with learning rate $\eta = 1$.

(b) To prove that:
for logistic regression with a very large w, argue that minimizing E_n using SGD is similar to PLA

From the equation on page 98, we know that:

$$\nabla e_n(w) = -\frac{y_n x_n}{1 + e^{y_n w^T x_n}}$$

Now, as per the problem w is large. Here again we have 2 cases.

Case: 1 y_n and $w^T x_n$ agrees.

If this is true, then $e^{y_n w^T x_n}$ also has to be large, since they are proportional.

If this is true then $\frac{-y_n x_n}{1 + e^{y_n w^T x_n}}$ is

Very small (approaching zero).

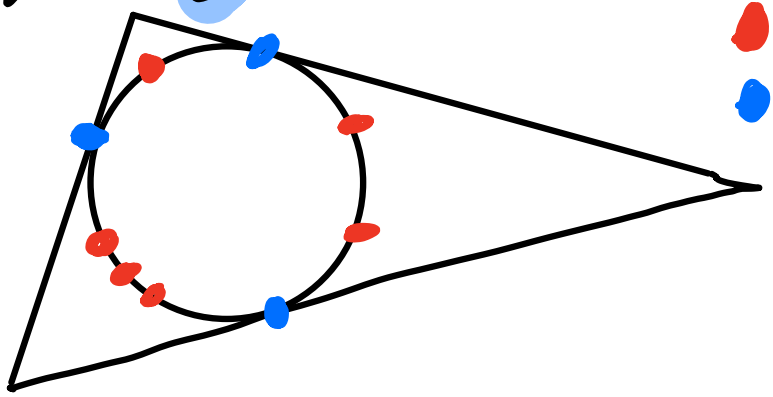
Case:2 y_n and $W^T X_n$ disagrees.

If this is true, then $e^{y_n W^T X_n}$ is small (almost too small compared to 1), hence $\frac{-y_n x_n}{1+e^{y_n W^T x_n}}$ will be almost approximately $\frac{-y_n x_n}{1} = -y_n x_n$.

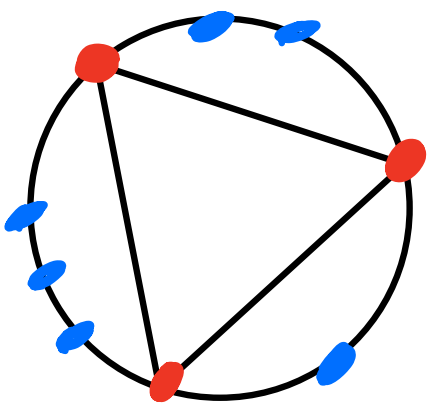
Hence, we have two possibilites, one approaching 0, and other $-y_n x_n$. This is just like PLA, hence the statement mentioned in the question is true.

(2) Consider a convex polygon with d-vertices (Also called as convex d-gons). Lets place 2d-1 points on a circle. Now, we have 2 specific cases:

→ Number of positive labels greater than Number of negative labels.



→ Number of negative labels greater than Number of positive labels.



You can see that, when the negative points are in majority, there are at most 'd' positive points and they are contained by the convex polygon by joining the red positive points (as shown).

In another case, consider the tangents at the negative points (highlighted in blue colour).

At lower limit, a set of 2d+1 points can be shattered. Hence for a polygon of 'd' vertices VC dimension can be seen to be 2d+1.

Also, placing the points on a circle maximises the number of sets required to shatter the set.

Hence, we can conclude that since a set with $2d+1$ points can be shattered, the VC dimension of the set of convex polygons with at-most d vertices is atleast $2d+1$, validating what was given in the question.

① For a linearly seperable data it is known that we can have a w^* so that:

$$y_n(w^*)^T x_n > 0 \text{ where}$$

$$n = 1 \ldots\ldots N$$

But if we swap out w^* with w, we get that

$$y_n(w^*)^T x_n \geq 1$$

Now we can cwrite the linearly seporable data as:

$$\begin{cases} \max_w c^T w \\ Aw \geq b \end{cases}$$

Because our range lies from $(-1, 1)$

We can write that:

$$b = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \Rightarrow b^T = (1, \ldots, 1)$$

A has to be:

$$\begin{pmatrix} — & y_1 x_1^T & — \\ — & y_2 x_2^T & — \\ & \vdots & \\ — & y_N x_N^T & — \end{pmatrix}$$

and $c = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \Rightarrow c^T = (1, 1 \ldots . 1)$

Problem_3_3

November 11, 2020

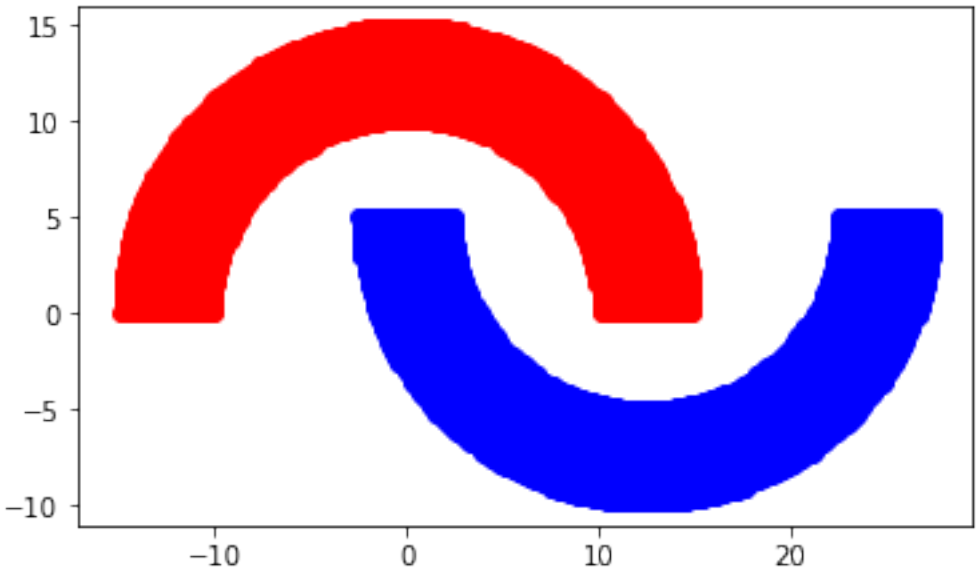
1 Problem 3.3 Learning From Data

[11]:
```
from matplotlib import pyplot as plt
import numpy as np
import random
from random import seed
```

2 Visualizing the dataset

[12]:
```
thk = 5
sep = -5
rad = 10

xs_red = []
ys_red = []

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_aspect('equal')

for x_coord in np.arange(-(rad+thk),rad+thk,0.15):
    for y_coord in np.arange(0 ,rad+thk,0.15):
        if rad**2 <= (x_coord - 0)**2 + (y_coord - 0)**2 <= (rad+thk)**2:
            xs_red.append(x_coord)
            ys_red.append(y_coord)

xs_blue = []
ys_blue = []

for x_coord in np.arange(-(thk/2),(thk/2 + (2*rad) + thk),0.15):
    for y_coord in np.arange(-sep ,-(rad++sep+thk),-0.15):
        if rad**2 <= (x_coord - ((thk/2) + rad))**2 + (y_coord - (-sep))**2 <=␣
↪(rad+thk)**2:
            xs_blue.append(x_coord)
            ys_blue.append(y_coord)
```

```
plt.scatter(xs_red, ys_red,color = 'red')
plt.scatter(xs_blue, ys_blue,color = 'blue')
plt.show()
```



3 PLA Algorithm

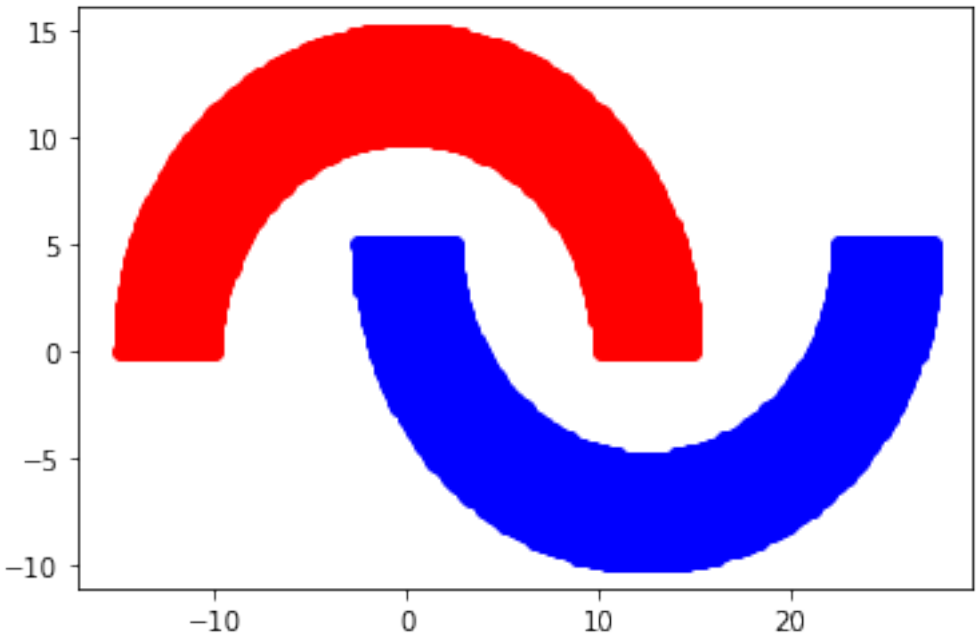
[13]:
```
"""
A function for prediction of Y
"""
def Y_predict(x_vector,w):
    x_new = [1]
    for i in x_vector:
        x_new.append(i)
    x_new = np.array((x_new))
    res = (np.dot(x_new,w))
    if res > 0:
        Y = 1
        return Y
    elif res < 0:
        Y = -1
        return Y
    elif res ==0:
        Y = 0
        return Y
```

```
"""
The main training function for the data, with the
Attributes
----------

X - The data set
iterations - the number of times the weights are iterated
eta - the learning rate
"""
def train(X,iterations,eta):
    global count
    global w
    global all_combined_targets
    for y_idx in range (len(X)):
        ran_num = random.randint(0,len(X)-1)
        x_train = X[ran_num]
        y_t = Y_predict(x_train,w)
        misrepresented_list = []
        for i,j in enumerate(all_combined_targets):
            if j!=y_t:
                misrepresented_list.append(i)
        if len(misrepresented_list)==0:
            print('Full accuracy achieved')
            break
        random_selection = random.randint(0,len(misrepresented_list)-1)
        random_index = misrepresented_list[random_selection]
        x_selected = X[random_index]
        y_selected = all_combined_targets[random_index]
        x_with1 = [1]
        for i in x_selected:
            x_with1.append(i)
        x_with1 = np.array((x_with1))
        s_t = np.matmul(w,x_with1)
        if (y_selected*s_t)<=1:
            w = w+(eta*(y_selected-s_t)*x_with1)
        if (count==iterations):
            print('maximum iterations reached in the training block')
            break
            count+=1
```

[14]:
```
xs_red = np.array(xs_red)
ys_red = np.array(ys_red)
xs_blue = np.array(xs_blue)
ys_blue = np.array(ys_blue)
points_1 = []
res1 = []
for i in range(len(xs_red)):
```

```
    points_1.append([xs_red[i],ys_red[i]])
    res1.append(-1)
points_1 = np.array(points_1)

points_2 = []
res2 = []
for i in range(len(xs_blue)):
    points_2.append([xs_blue[i],ys_blue[i]])
    res2.append(1)
points_2 = np.array(points_2)
all_input = np.concatenate((points_1, points_2)) #creating a combined dataset
all_d = np.concatenate((res2,res1))
```

[15]:
```
#Visualizing the linearly separable dataset
plt.scatter(xs_red, ys_red, color='red')
plt.scatter(xs_blue,ys_blue, color='blue')
length_dataset = len(xs_red)
d1 = -1 * (np.ones(int(length_dataset/2)))
d2 = np.ones(int(length_dataset/2))
all_combined_targets = np.concatenate((d2,d1))
```



4 WARNING! Running the below snippet of code, will not reach an end. This is how the PLA would react in this case, hence the next two blocks haven't been executed.

5 Answer (a) This showcases that the PLA will keep running for infinite time, without reaching an end.

[ ]:
```
#initializing all parameters
count = 0
# w0 = random.randint(1,4)
# w1 = random.randint(1,4)
# w2 = random.randint(1,4)
w0,w1,w2 = 0,0,0
w = np.array((w0,w1,w2))
weight= 0
iterations = 100
eta = 0.01
#calling the function
train(all_input,iterations,eta)
```

[ ]:
```
#Visualizing the linearly separable dataset after
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_aspect('equal')

plt.scatter(xs_red, ys_red, color='red')
plt.scatter(xs_blue,ys_blue, color='blue')
m = -(w[1]/w[2])
c = -(w[0]/w[2])
plt.plot( m*all_input + c,all_input ,'g--')
plt.xlim([-20, 30])
plt.ylim([-30, 20])
```

6 Answer (b) Pocket Algorithm

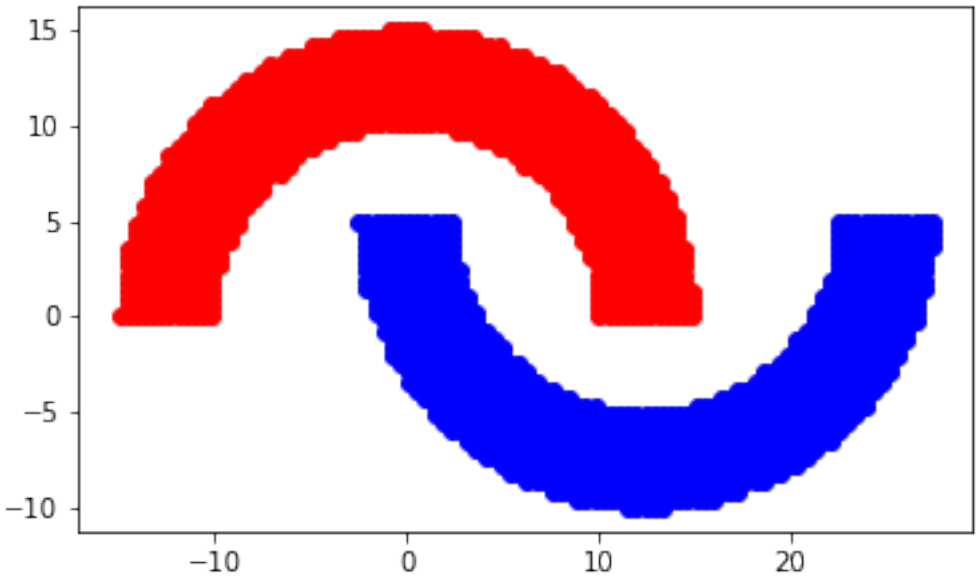
[16]:
```
#to reset all the varaible.s ONLY TO BE RUN IF YOU HAVE RUN THE CODES BEFORE␣
 ↪THIS
%reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

[17]:
```
from matplotlib import pyplot as plt
import numpy as np
import random
from sklearn import linear_model
from random import seed
```

```
np.random.seed(1)
```

[18]:
```
thk = 5
sep = -5
rad = 10

xs_red = []
ys_red = []

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_aspect('equal')

step= 0.44
for x_coord in np.arange(-(rad+thk),rad+thk,step):
    for y_coord in np.arange(0 ,rad+thk,step):
        if rad**2 <= (x_coord - 0)**2 + (y_coord - 0)**2 <= (rad+thk)**2:
            xs_red.append(x_coord)
            ys_red.append(y_coord)

xs_blue = []
ys_blue = []

for x_coord in np.arange(-(thk/2),(thk/2 + (2*rad) + thk),step):
    for y_coord in np.arange(-sep ,-(rad++sep+thk),-step):
        if rad**2 <= (x_coord - ((thk/2) + rad))**2 + (y_coord - (-sep))**2 <=␣
 ↪(rad+thk)**2:
            xs_blue.append(x_coord)
            ys_blue.append(y_coord)


plt.scatter(xs_red, ys_red,color = 'red')
plt.scatter(xs_blue, ys_blue,color = 'blue')
plt.show()
```



[19]:
```
# dataset preparation
x_combined = xs_red +xs_blue
y_combined = ys_red + ys_blue
y_train_1  = np.ones((len(xs_red), ), dtype=np.float)
y_train_2 = -1 * np.ones((len(xs_blue), ), dtype=np.float)
y_train = np.concatenate((y_train_1,y_train_2))
x_train = list(zip(x_combined, y_combined))
x_train = np.array(x_train)
n_train = len(x_train)
learningRate = 0.01
Y = y_train
oneVector = np.ones((x_train.shape[0], 1))
x_train = np.concatenate((oneVector, x_train), axis=1)
X_train = x_train
plotData = []
weights = np.random.rand(3, 1)
w_hat = weights
misClassifications = 1
minMisclassifications = 10000
iteration = 0
err_train_now = []
err_train_hat = []
train_err_now = 1
train_err_min = 1
```

[20]:
```
def evaluate_error(w, X, y):
    n = X.shape[0]
```

```
    pred = np.matmul(X, w)
    pred = np.sign(pred) - (pred == 0)
    pred = pred.reshape(-1)
    return np.count_nonzero(pred == y) / n
```

[21]:
```
while (misClassifications != 0 and (iteration<100000)):
    iteration += 1
    #for keeping track of the progress for 100000 iterations
    if (iteration%1000) == 0:
        print(iteration)
    misClassifications = 0
    for i in range(0, len(X_train)):
        currentX = X_train[i].reshape(-1, X_train.shape[1])
        currentY = Y[i]
        wTx = np.dot(currentX, weights)[0][0]
        if currentY == 1 and wTx < 0:
            misClassifications += 1
            weights = weights + learningRate * np.transpose(currentX)
        elif currentY == -1 and wTx > 0:
            misClassifications += 1
            weights = weights - learningRate * np.transpose(currentX)

        train_err_now = evaluate_error(weights, X_train, y_train)
        err_train_now.append(train_err_now)

        if train_err_now < train_err_min :
            train_err_min = train_err_now
            err_train_hat.append(train_err_min)
            w_hat = weights

    plotData.append(misClassifications)
    if misClassifications<minMisclassifications:
        minMisclassifications = misClassifications
print(weights.transpose())
print ("Best Case Accuracy of Pocket Learning Algorithm is: ",(((X_train.
 ↪shape[0]-minMisclassifications)/X_train.shape[0])*100),"%")
```

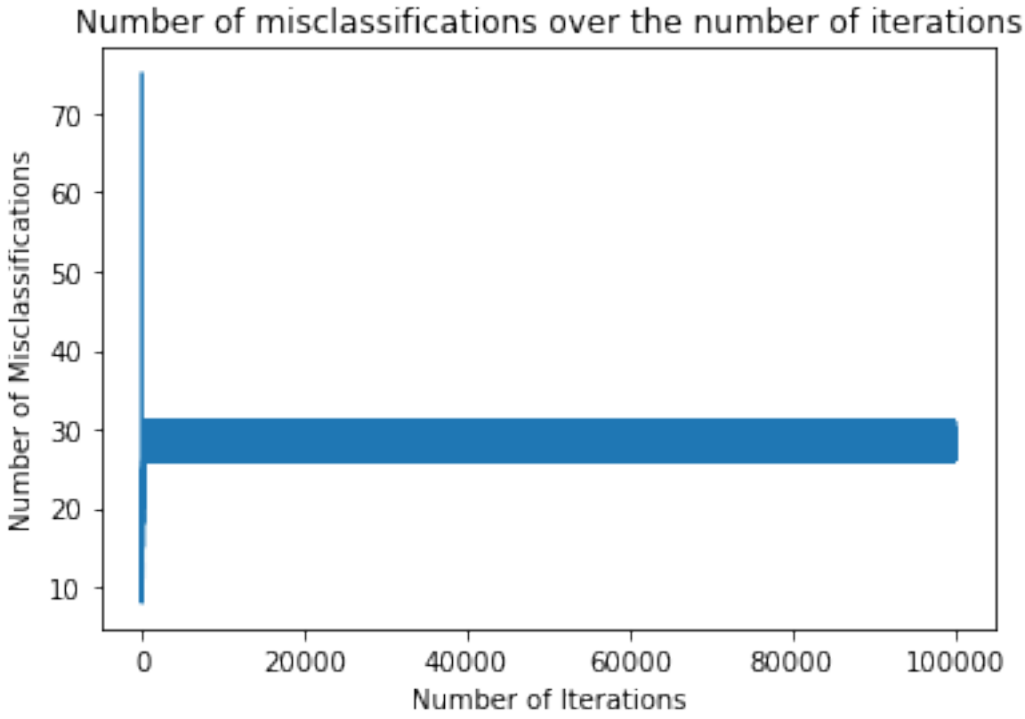
```
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
```

12000
13000
14000
15000
16000
17000
18000
19000
20000
21000
22000
23000
24000
25000
26000
27000
28000
29000
30000
31000
32000
33000
34000
35000
36000
37000
38000
39000
40000
41000
42000
43000
44000
45000
46000
47000
48000
49000
50000
51000
52000
53000
54000
55000
56000
57000
58000
59000

```
60000
61000
62000
63000
64000
65000
66000
67000
68000
69000
70000
71000
72000
73000
74000
75000
76000
77000
78000
79000
80000
81000
82000
83000
84000
85000
86000
87000
88000
89000
90000
91000
92000
93000
94000
95000
96000
97000
98000
99000
100000
[[-4.362978   -0.14447551  0.62491437]]
Best Case Accuracy of Pocket Learning Algorithm is:  99.6124031007752 %
```

[22]:
```
plt.title('Number of misclassifications over the number of iterations')
plt.plot(np.arange(0,iteration),plotData)
plt.xlabel("Number of Iterations")
```

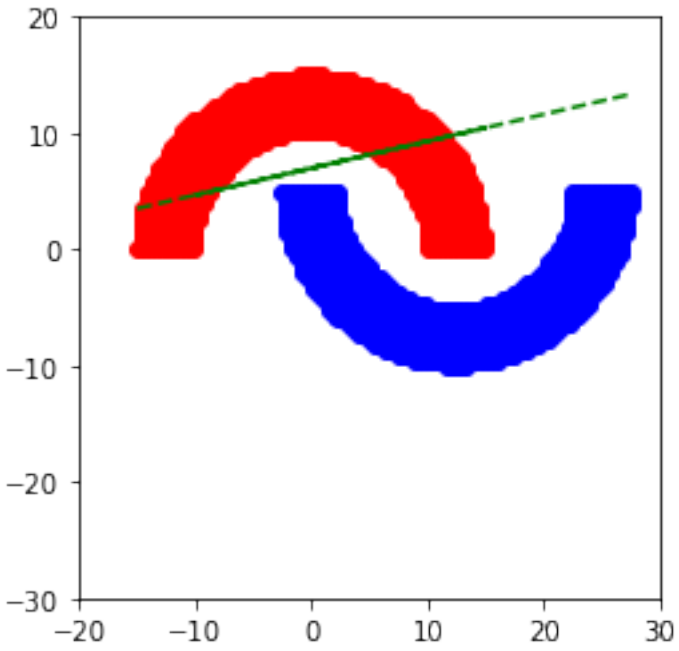
```
plt.ylabel("Number of Misclassifications")
plt.show()
```



7 Answer(c) The plot between the data and the final hypothesis

[23]:
```
#Visualizing the linearly separable dataset
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_aspect('equal')

plt.scatter(xs_red, ys_red, color='red')
plt.scatter(xs_blue,ys_blue, color='blue')
m = -(weights[1]/weights[2])
c = -(weights[0]/weights[2])
plt.plot( x_train, m*x_train + c ,'g--')
plt.xlim([-20, 30])
plt.ylim([-30, 20])
```

[23]:
```
(-30.0, 20.0)
```



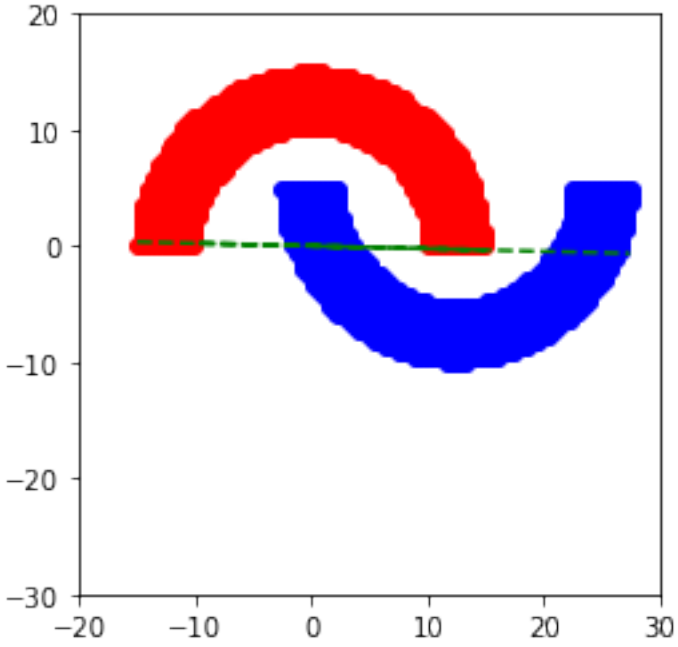
8 Answer (d) Linear regression

[24]:
```
reg = linear_model.LinearRegression()
reg.fit(x_train,y_train)
weights_linear_regression = reg.coef_
l = []
for i in weights_linear_regression:
    l.append(i)
print(weights_linear_regression)
```

```
[ 0.         -0.02335229  0.10002409]
```

[25]:
```
#Visualizing the linearly separable dataset
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_aspect('equal')

plt.scatter(xs_red, ys_red, color='red')
plt.scatter(xs_blue,ys_blue, color='blue')
m = weights_linear_regression[1]
c = weights_linear_regression[0]
plt.plot( x_train, m*x_train + c ,'g--')
plt.xlim([-20, 30])
plt.ylim([-30, 20])
```

[25]: (-30.0, 20.0)



9 Answer (d) It is seen that the linear regression offers better results as opposed to PLA, in terms of speed and fit

[ ]:

Problem_3_3_ThirdOrder

November 11, 2020

1 Problem 3.3 Learning From Data

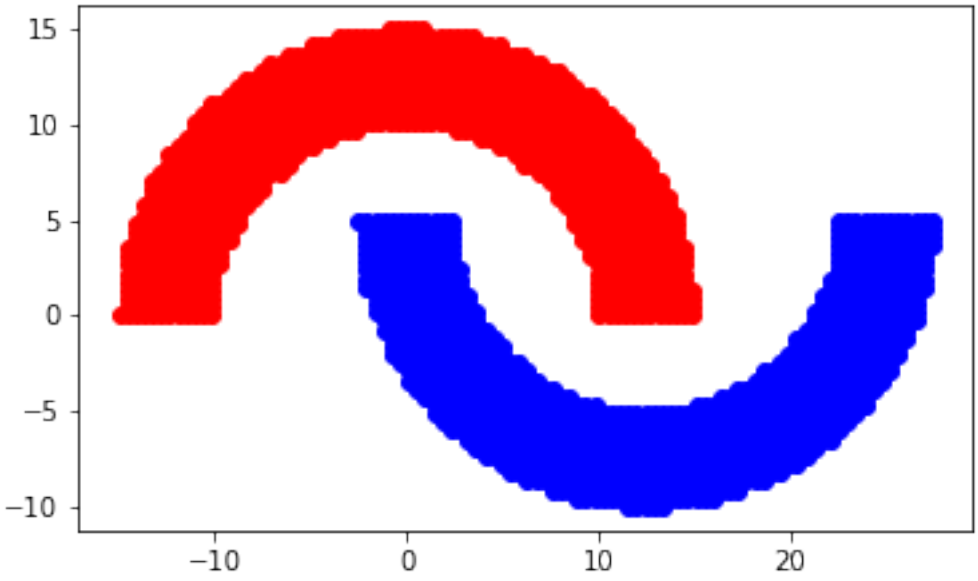
2 Answer d : For the 3rd order polynomial feature transform

[1]:
```
from matplotlib import pyplot as plt
import numpy as np
import random
from sklearn import linear_model
from random import seed
np.random.seed(1)
from sklearn.preprocessing import PolynomialFeatures
```

3 Visualizing the dataset

[2]:
```
thk = 5
sep = -5
rad = 10

xs_red = []
ys_red = []

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_aspect('equal')

step= 0.44
for x_coord in np.arange(-(rad+thk),rad+thk,step):
    for y_coord in np.arange(0 ,rad+thk,step):
        if rad**2 <= (x_coord - 0)**2 + (y_coord - 0)**2 <= (rad+thk)**2:
            xs_red.append(x_coord)
            ys_red.append(y_coord)

xs_blue = []
ys_blue = []

for x_coord in np.arange(-(thk/2),(thk/2 + (2*rad) + thk),step):
```

```
    for y_coord in np.arange(-sep ,-(rad++sep+thk),-step):
        if rad**2 <= (x_coord - ((thk/2) + rad))**2 + (y_coord - (-sep))**2 <=␣
↪(rad+thk)**2:
            xs_blue.append(x_coord)
            ys_blue.append(y_coord)


plt.scatter(xs_red, ys_red,color = 'red')
plt.scatter(xs_blue, ys_blue,color = 'blue')
plt.show()
```



4 Repeating Pocket Algorithm

[3]:
```
# dataset preparation
x_combined = xs_red +xs_blue
y_combined = ys_red + ys_blue
y_train_1  = np.ones((len(xs_red), ), dtype=np.float)
y_train_2 = -1 * np.ones((len(xs_blue), ), dtype=np.float)
y_train = np.concatenate((y_train_1,y_train_2))
x_train = list(zip(x_combined, y_combined))
x_for_third_degree = x_train
x_train = np.array(x_train)
n_train = len(x_train)
learningRate = 0.01
Y = y_train
oneVector = np.ones((x_train.shape[0], 1))
```

```
x_train = np.concatenate((oneVector, x_train), axis=1)
X_train = x_train
```

[4]:
```
def evaluate_error(w, X, y):
    n = X.shape[0]
    pred = np.matmul(X, w)
    pred = np.sign(pred) - (pred == 0)
    pred = pred.reshape(-1)
    return np.count_nonzero(pred == y) / n
```

5 Setting up the third order feature transform

[5]:
```
trans = PolynomialFeatures(degree=3)
x_for_third_degree = trans.fit_transform(x_for_third_degree)
x_for_third_degree = np.array(x_for_third_degree)
```

[6]:
```
x_train = np.concatenate((oneVector, x_for_third_degree), axis=1)
X_train = x_for_third_degree
plotData = []
weights = np.random.rand(10, 1)
w_hat = weights
misClassifications = 1
minMisclassifications = 10000
iteration = 0
err_train_now = []
err_train_hat = []
train_err_now = 1
train_err_min = 1
```

[7]:
```
while (misClassifications != 0 and (iteration<100000)):
    iteration += 1
    #for keeping track of the progress for 100000 iterations
    if (iteration%1000) == 0:
        print(iteration)
    misClassifications = 0
    for i in range(0, len(X_train)):
        currentX = X_train[i].reshape(-1, X_train.shape[1])
        currentY = Y[i]
        wTx = np.dot(currentX, weights)[0][0]
        if currentY == 1 and wTx < 0:
            misClassifications += 1
            weights = weights + learningRate * np.transpose(currentX)
        elif currentY == -1 and wTx > 0:
            misClassifications += 1
            weights = weights - learningRate * np.transpose(currentX)
```

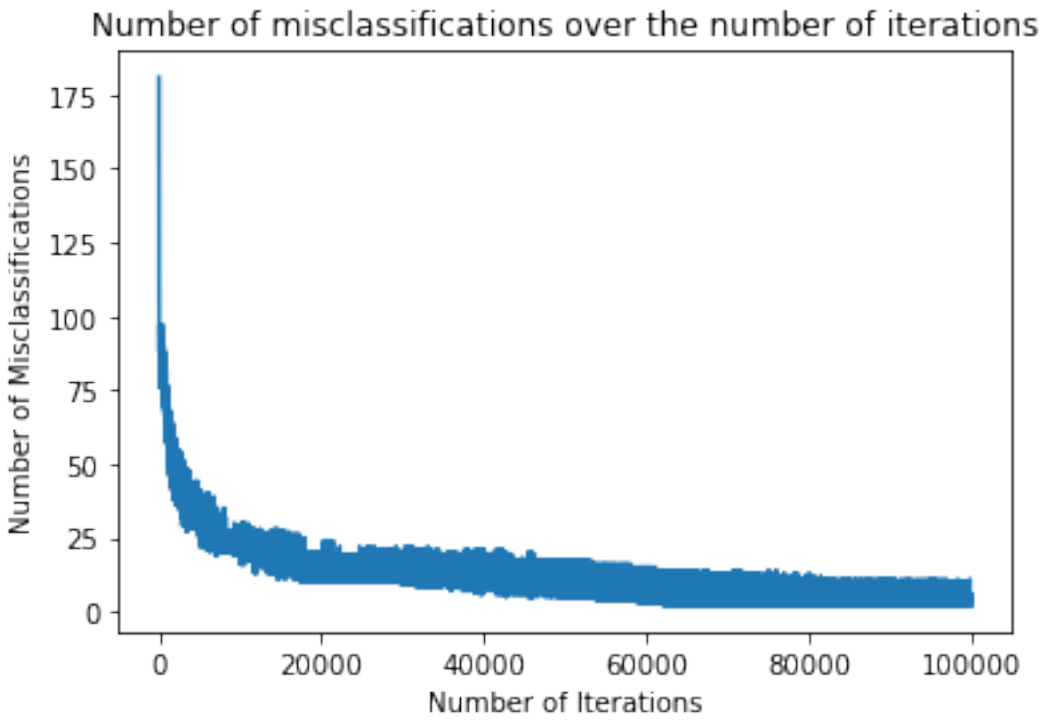
```
        train_err_now = evaluate_error(weights, X_train, y_train)
        err_train_now.append(train_err_now)

        if train_err_now < train_err_min :
            train_err_min = train_err_now
            err_train_hat.append(train_err_min)
            w_hat = weights

    plotData.append(misClassifications)
    if misClassifications<minMisclassifications:
        minMisclassifications = misClassifications
print(weights.transpose())
print ("Best Case Accuracy of Pocket Learning Algorithm is: ",(((X_train.
 ↪shape[0]-minMisclassifications)/X_train.shape[0])*100),"%")
```

```
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
14000
15000
16000
17000
18000
19000
20000
21000
22000
23000
24000
25000
26000
27000
28000
29000
30000
31000
32000
33000
```

34000
35000
36000
37000
38000
39000
40000
41000
42000
43000
44000
45000
46000
47000
48000
49000
50000
51000
52000
53000
54000
55000
56000
57000
58000
59000
60000
61000
62000
63000
64000
65000
66000
67000
68000
69000
70000
71000
72000
73000
74000
75000
76000
77000
78000
79000
80000
81000

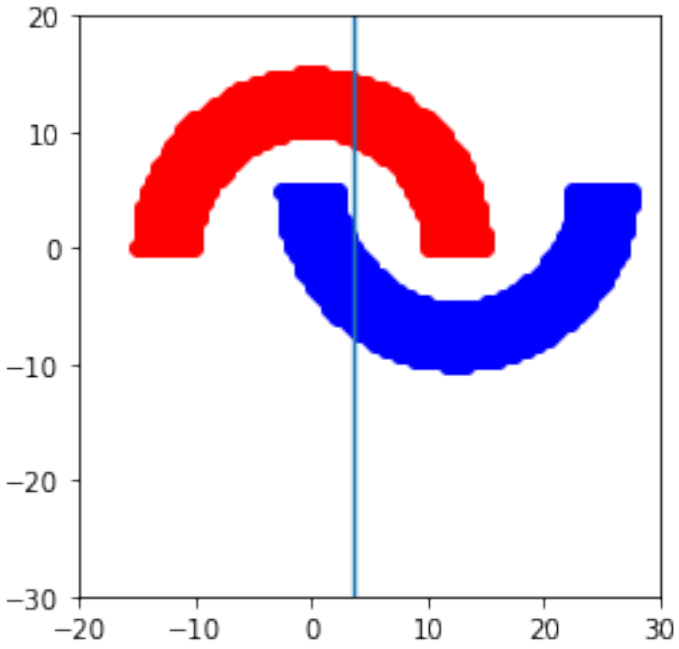
```
82000
83000
84000
85000
86000
87000
88000
89000
90000
91000
92000
93000
94000
95000
96000
97000
98000
99000
100000
[[-8132.28297813 -3545.39087547 -1415.45348564  2063.92219659
   1749.5152119  -3655.41146934  -120.39309532   -32.07075815
    -99.72529941   722.32982602]]
Best Case Accuracy of Pocket Learning Algorithm is:  99.90310077519379 %
```

[8]:
```
plt.title('Number of misclassifications over the number of iterations')
plt.plot(np.arange(0,iteration),plotData)
plt.xlabel("Number of Iterations")
plt.ylabel("Number of Misclassifications")
plt.show()
```



[9]:
```
#Visualizing the linearly separable dataset
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_aspect('equal')

plt.scatter(xs_red, ys_red, color='red')
plt.scatter(xs_blue,ys_blue, color='blue')
num_points = 200
x_poly_points = np.linspace(-20,30,num_points-1)
y_poly_points = np.linspace(-20,30,num_points-1)
f = []
for i in range(num_points-1):
    f.append((1*weights[0][0])+(weights[1][0]*x_poly_points[i]) +␣
 ↪(weights[2][0]*y_poly_points[i]) + (weights[3][0]*(x_poly_points[i]**2)) +
             (weights[4][0]*x_poly_points[i]*y_poly_points[i]) +␣
 ↪(weights[5][0]*(y_poly_points[i]**2)) + (weights[6][0]*(x_poly_points[i]**3))
             +(weights[7][0]*(x_poly_points[i]**2)*y_poly_points[i]) +␣
 ↪(weights[8][0]*x_poly_points[i]*(y_poly_points[i]**2))
             +(weights[9][0]*(y_poly_points[i]**3)))
plt.plot(x_poly_points,f)
plt.xlim([-20, 30])
plt.ylim([-30, 20])
```

[9]: (-30.0, 20.0)



6 Answer (d) Linear regression

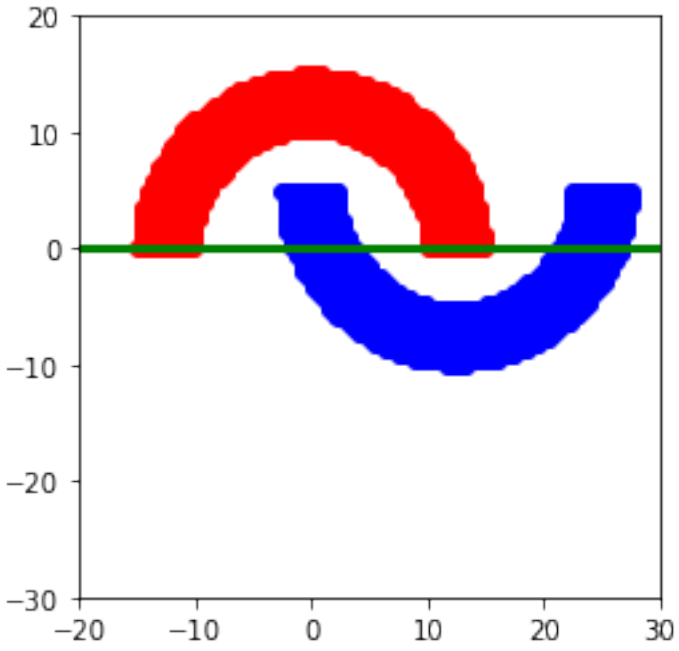
[10]:
```
reg = linear_model.LinearRegression()
reg.fit(x_train,y_train)
weights_linear_regression = reg.coef_
l = []
for i in weights_linear_regression:
    l.append(i)
print(weights_linear_regression)
```

```
[ 0.00000000e+00  5.29090660e-16  2.25069728e-02  1.30121222e-01
  8.65754742e-03  1.15922621e-02  1.39005514e-02 -3.92825494e-04
 -5.12882683e-04 -1.03413517e-03 -9.94036096e-04]
```

[11]:
```
#Visualizing the linearly separable dataset
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_aspect('equal')

plt.scatter(xs_red, ys_red, color='red')
plt.scatter(xs_blue,ys_blue, color='blue')
m = weights_linear_regression[1]
c = weights_linear_regression[0]
```

```
plt.plot( x_train, m*x_train + c ,'g--')
plt.xlim([-20, 30])
plt.ylim([-30, 20])
```

[11]: (-30.0, 20.0)



[ ]:

Problem_5 MidTerm

November 11, 2020

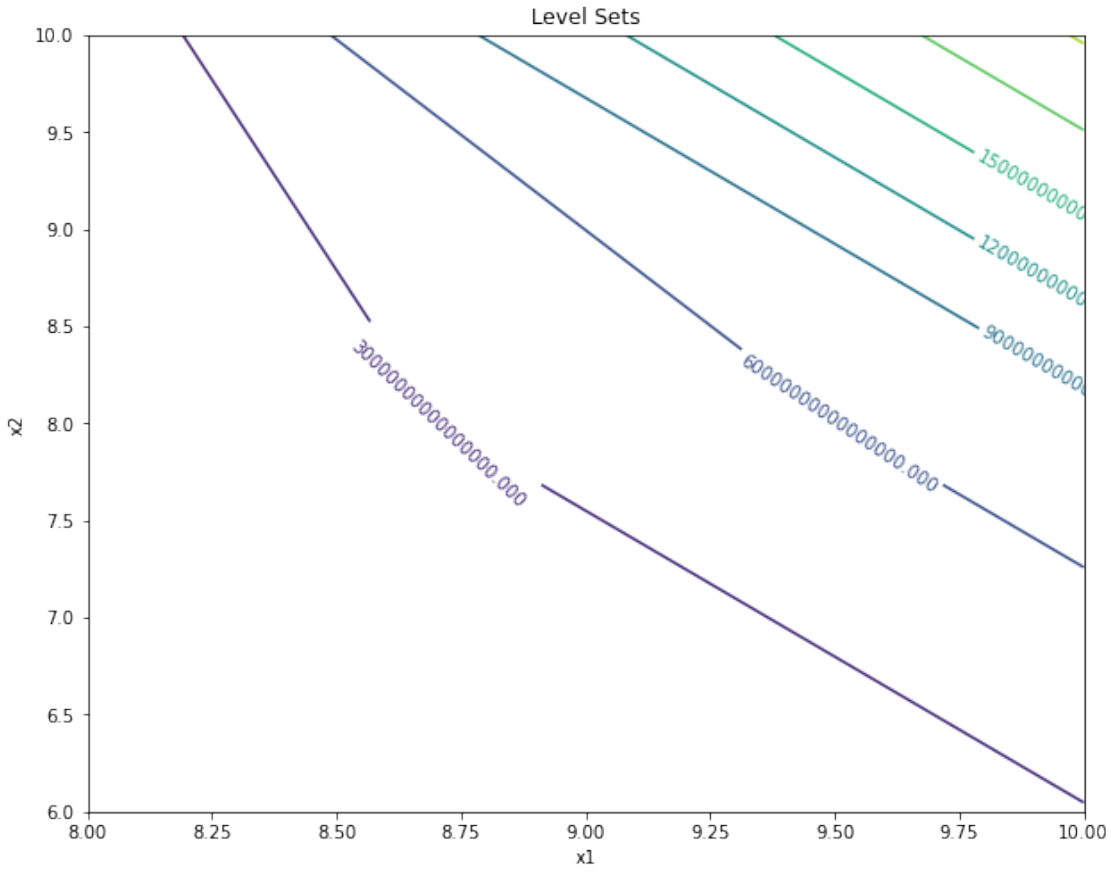
1 Problem_5 MidTerm

[1]:
```
# importing all necessary libraries
import numpy as np
import matplotlib.pyplot as plt
import math

#for plotting the level sets

import pylab

#for computing the gradient of the function

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

#for gradient descent

from sympy import *
```

2 Plotting the level sets

[2]:
```
# initializing an empty list for x1_points
x1_points     = []

# initializing an empty list for x2_points
x2_points     = []

# different values of x1 and x2
for val in range(-10, 12, 2):
    x1_points.append(val)
    x2_points.append(val)

# k values
k  = np.ndarray((11,11))

# for different k values as asked in the question
```

```
for x_1 in range(0, len(x1_points)):
    for x_2 in range(0, len(x2_points)):
        k[x_1][x_2] = math.exp(x_1 + (3*x_2) - 0.1) + math.exp(x_1 - (3*x_2) -␣
 ↪0.1) + math.exp(-x_1 - 0.1)

#setting the figure size
plt.figure(figsize=(10,8))

#run once and  alter the limits to get the graph
pylab.xlim([8, 10])
pylab.ylim([6, 10])


# Provide a title for the contour plot
plt.title('Level Sets')

#x1 axis label
plt.xlabel('x1')

#x2 axis label
plt.ylabel('x2')


# Plotting contours
contours = plt.contour(x1_points, x2_points, k)

# Display z values on contour lines
plt.clabel(contours, inline=1, fontsize=10)

# # Display the contour plot
plt.show()
```



3 Computing the gradient of the function

[3]:
```
def f(x_1,x_2):
    return math.exp(x_1 + (3*x_2) - 0.1) + math.exp(x_1 - (3*x_2) - 0.1) + math.
 ↪exp(-x_1 - 0.1)
```

[4]:
```
x_1 = x_2 = np.arange(-4, 4, 0.05)
X_1, X_2 = np.meshgrid(x_1, x_2)
zs = np.array([f(x_1,x_2) for x_1,x_2 in zip(np.ravel(X_1), np.ravel(X_2))])
Z = zs.reshape(X_1.shape)
g_x_1,g_x_2 = np.gradient(Z,0.05,0.05)
```

[5]:
```
# Gradient of the function with respect to the x_1 dimension
print('Gradient of the function with respect to the x_1 dimension :')
print(g_x_1)
```

```
Gradient of the function with respect to the x_1 dimension :
[[-7.51419827e+03 -7.89945946e+03 -8.30447340e+03 … -1.92794364e+07
  -2.02679142e+07 -2.13070724e+07]
```

```
 [-6.99086433e+03 -7.34929361e+03 -7.72609995e+03 … -1.79367005e+07
  -1.88563348e+07 -1.98231198e+07]
 [-6.01709269e+03 -6.32559563e+03 -6.64991585e+03 … -1.54382612e+07
  -1.62297978e+07 -1.70619173e+07]
 …
 [ 5.17895967e+03  5.44449062e+03  5.72363562e+03 …  1.32878346e+07
   1.39691164e+07  1.46853283e+07]
 [ 6.01709269e+03  6.32559563e+03  6.64991585e+03 …  1.54382612e+07
   1.62297978e+07  1.70619173e+07]
 [ 6.46753039e+03  6.79912776e+03  7.14772650e+03 …  1.65939647e+07
   1.74447554e+07  1.83391672e+07]]
```

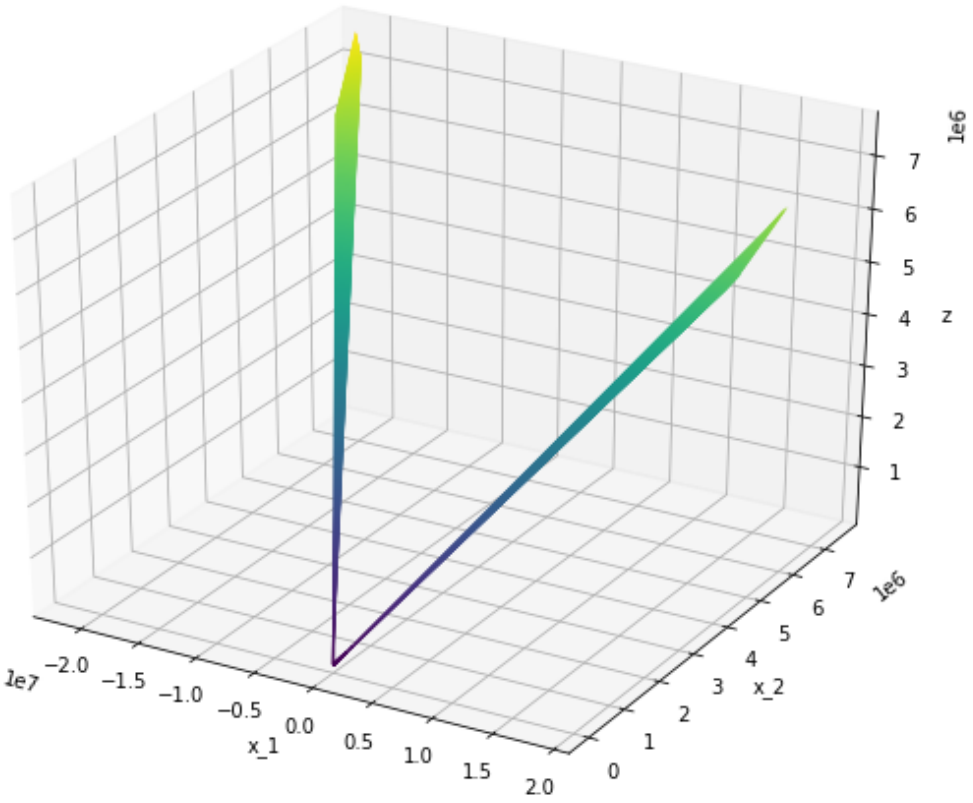
[6]:
```
# Gradient of the function with respect to the x_2 dimension
print('Gradient of the function with respect to the x_2 dimension :')
print(g_x_2)
```

```
Gradient of the function with respect to the x_2 dimension :
[[2.71766473e+03 2.78974394e+03 2.93748040e+03 … 6.92339372e+06
  7.27836371e+06 7.46028490e+06]
 [2.33240354e+03 2.39460638e+03 2.52208369e+03 … 5.95902020e+06
  6.26454570e+06 6.42112672e+06]
 [2.00080617e+03 2.05450832e+03 2.16454844e+03 … 5.12897621e+06
  5.39194445e+06 5.52671498e+06]
 …
 [1.71539766e+03 1.76178321e+03 1.85681499e+03 … 4.41455073e+06
  4.64088959e+06 4.75688767e+06]
 [2.00080617e+03 2.05450832e+03 2.16454844e+03 … 5.12897621e+06
  5.39194445e+06 5.52671498e+06]
 [2.33240354e+03 2.39460638e+03 2.52208369e+03 … 5.95902020e+06
  6.26454570e+06 6.42112672e+06]]
```

[7]:
```
#Visuzlizing the gradient
fig = plt.figure()
plt.figure(figsize=(10,8))
ax = plt.axes(projection='3d')
ax.contour3D(g_x_1, g_x_2, Z, 1000)
ax.set_xlabel('x_1')
ax.set_ylabel('x_2')
ax.set_zlabel('z')
```

[7]:
```
Text(0.5, 0, 'z')
```

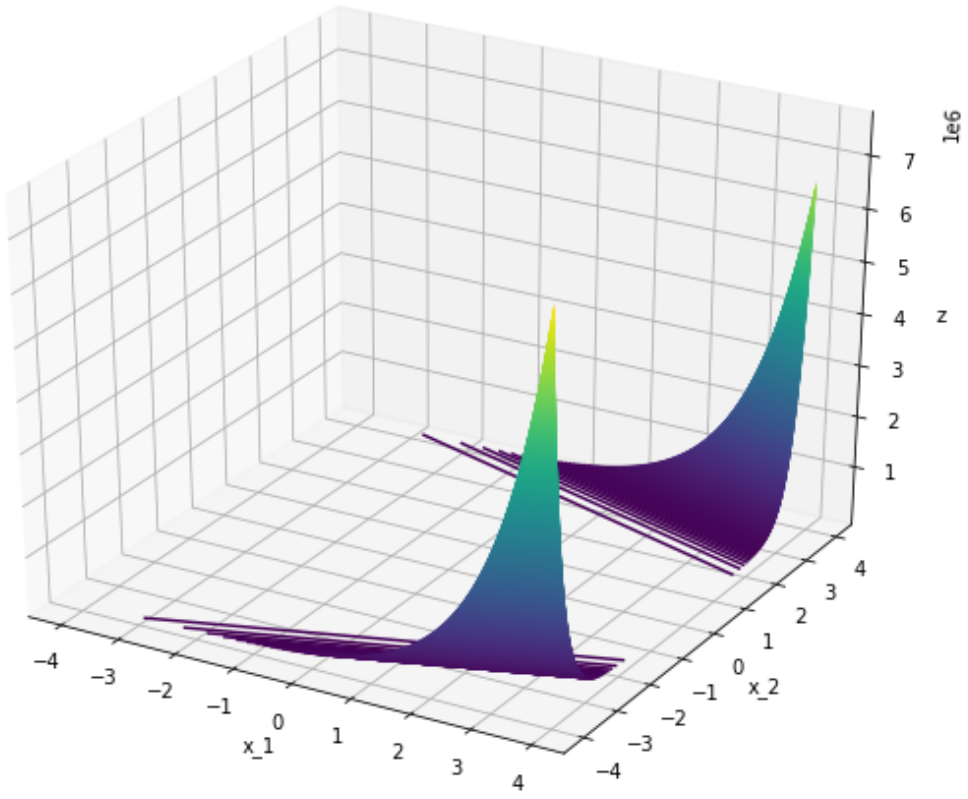
```
<Figure size 432x288 with 0 Axes>
```



[8]:
```
#Visuzlizing the 3D PLOT OF X_1 and X_2
fig = plt.figure()
plt.figure(figsize=(10,8))
ax = plt.axes(projection='3d')
ax.contour3D(X_1, X_2, Z, 1000)
ax.set_xlabel('x_1')
ax.set_ylabel('x_2')
ax.set_zlabel('z')
```

[8]:
```
Text(0.5, 0, 'z')
```

```
<Figure size 432x288 with 0 Axes>
```



4 Gradient Descent

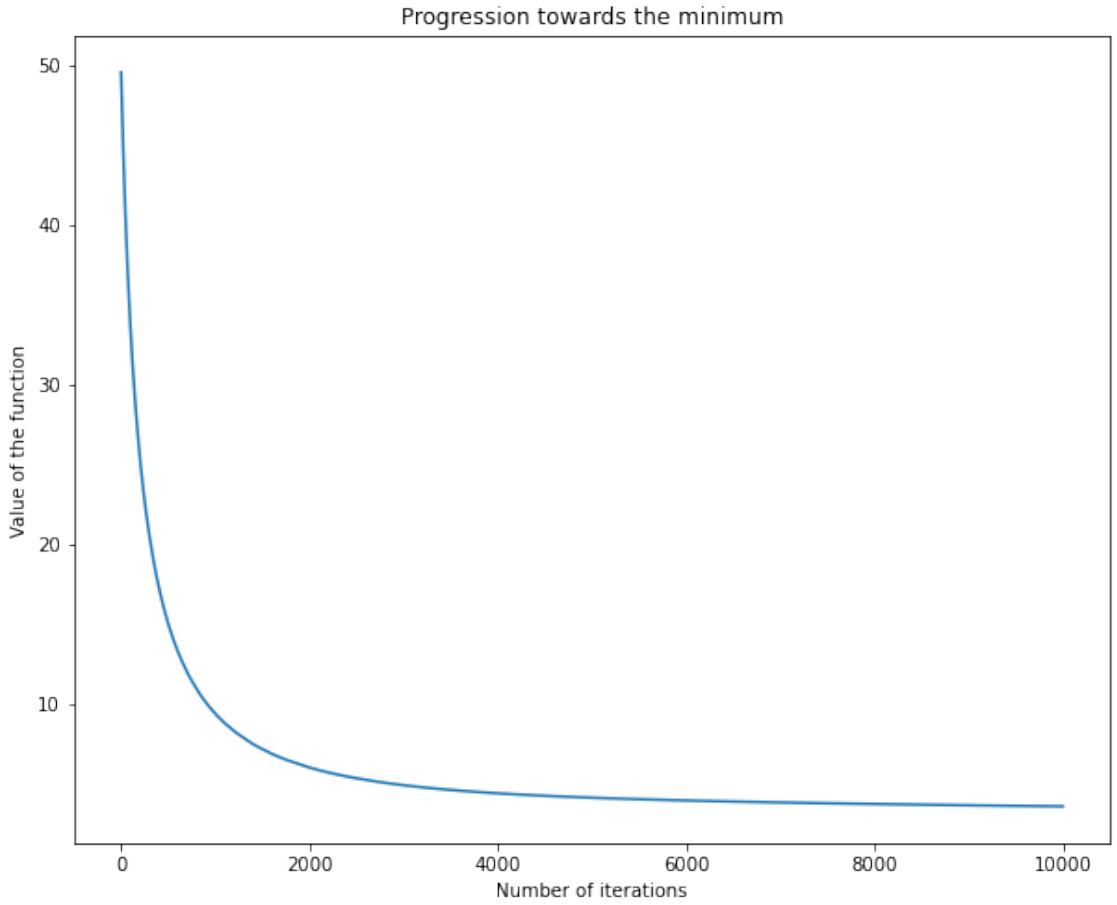
[9]:
```
x_1 = Symbol('x_1')
x_2 = Symbol('x_2')
all_func = [] #to plot the progression later
e = 2.718 #estimating the value of e
def func(x_1,x_2):
    calculated_func = e**(x_1 + (3*x_2) - 0.1) + e**(x_1 - (3*x_2) - 0.1) +␣
 ↪e**(-x_1 - 0.1)
    calculated_func = round(calculated_func,3)
    return (calculated_func)

f = e**(x_1 + (3*x_2) - 0.1) + e**(x_1 - (3*x_2) - 0.1) + e**(-x_1 - 0.1)

# First partial derivative of the function with respect to x_1
f_x_1 = f.diff(x_1)
# First partial derivative of the function with respect to x_2
f_x_2 = f.diff(x_2)
```

```
# Gradient
grad = [f_x_1,f_x_2]

theta_x_1 = 1  #x_1
theta_x_2 = 1 # x_2
alpha = .00001
iterations = 0
precision = 1/1000000
maxIterations = 10000

while True:
    curr_theta_x_1 = theta_x_1 - alpha*N(f_x_1.subs(x_1,theta_x_1).
 ↪subs(x_2,theta_x_2)).evalf()
    curr_theta_x_2 = theta_x_2 - alpha*N(f_x_2.subs(x_2,theta_x_2)).
 ↪subs(x_1,theta_x_1).evalf()

    iterations += 1
    if iterations > maxIterations:
        print("Too many iterations. Adjust alpha and make sure that the␣
 ↪function is convex!")
        print('The graph below shows the progression towards the minimum')
        plt.figure(0)
        plt.figure(figsize=(10,8))
        plt.title('Progression towards the minimum')
        plt.xlabel('Number of iterations')
        plt.ylabel('Value of the function')
        plt.plot(all_func)
        printData = False
        break

    if abs(curr_theta_x_1-theta_x_1) < precision and␣
 ↪abs(curr_theta_x_2-theta_x_2) < precision:
        break

    #adding to the all_function list
    all_func.append(func(curr_theta_x_1,curr_theta_x_2))

    #updating every value
    theta_x_1 = curr_theta_x_1
    theta_x_2 = curr_theta_x_2


#printing out the final results
print("The function "+str(function)+"has now converged to a minimum of : " ,␣
 ↪func(curr_theta_x_1,curr_theta_x_2))
print("Number of iterations it took was recorded to be :",iterations)
print("Minimum x_1 = ",curr_theta_x_1)
```

```
print("Minimum x_2 =",curr_theta_x_2)
```

```
Too many iterations. Adjust alpha and make sure that the function is convex!
The graph below shows the progression towards the minimum
The function <sympy.DeprecatedImportModule object at 0x000001CF3877BD60>has now
converged to a minimum of :  3.552
Number of iterations it took was recorded to be : 10001
Minimum x_1 =  0.504440656756166
Minimum x_2 = 0.0252579196980445
```

```
<Figure size 432x288 with 0 Axes>
```



[ ]: