Problem 3.1 (a) Run the PLA starting from w =0 until it converges. Plot the data and the final hypothesis.

In [ ]:

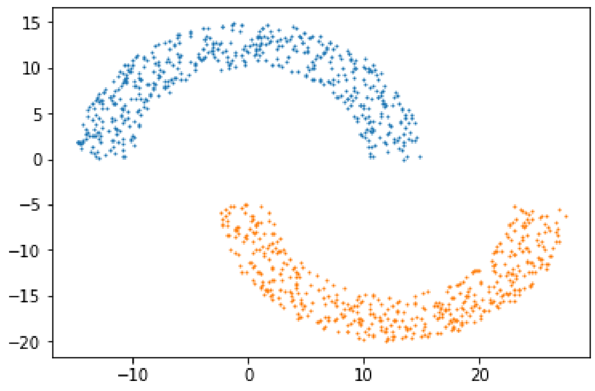
```
# Generate data below -
```

In [382]:

```
import numpy as np

#parameters
rad = 10
thk = 5
sep = 5

#n data points,(x1,y1) are the coordinates of the top semi-circle
def generatedata(rad,thk,sep,n,x1=0,y1=0):
    # center of the top semi-circle
    X1 = x1
    Y1 = y1

    # center of the bottom semi-circle
    X2 = X1 + rad + thk / 2
    Y2 = Y1 - sep

    # data points in the top semi-circle
    top = []
    # data points in the bottom semi-circle
    bottom = []

    # parameters
    r1 = rad + thk
    r2 = rad

    cnt = 1
    while(cnt <= n):
        #uniformed generated points
        x = np.random.uniform(-r1,r1)
        y = np.random.uniform(-r1,r1)

        d = x**2 + y**2
        if(d >= r2**2 and d <= r1**2):
            if (y > 0):
                top.append([X1 + x,Y1 + y])
                cnt += 1
            else:
                bottom.append([X2 + x,Y2 + y])
                cnt += 1
        else:
            continue

    return top,bottom
```

In [397]:

```
import matplotlib.pyplot as plt
top,bottom = generatedata(rad,thk,sep,1000)

X1 = [i[0] for i in top]
Y1 = [i[1] for i in top]

X2 = [i[0] for i in bottom]
Y2 = [i[1] for i in bottom]

plt.scatter(X1,Y1,s = 1)
plt.scatter(X2,Y2,s = 1)
plt.show()
```



In [398]:

```
# pre-processing the data for (a)
x1 = [[1] + i + [1] for i in top]
x2 = [[1] + i + [-1] for i in bottom]
data = x1 + x2

data = np.array(data)
np.random.shuffle(data)
```

In [399]:

```
X = data[:,0:3]
```

In [400]:

```
y = data[:,3]
```

In [405]:

```
#Algorith for PLA
def perceptron(X, Y):
#initialize the weights as 0
    w = [0,0,0]
#set learning rate to 1
    eta = 1
    iter = 0
    while True:
#get sign of the dot product of w transpose and X
        sig = np.sign(np.dot(X, np.transpose(w)))
# If the sign vector is same as the expected output vector y, then separation is complete
        correct = sig == Y
        if np.all(correct):
            print('total no of iterations',iter)
            return w
        else:
#Otherwise increase weight
            i = np.argmax(~correct)
            w = w + eta*X[i]*Y[i]
        iter = iter+1
```

In [406]:

```
weight = perceptron(X,y)
weight
```

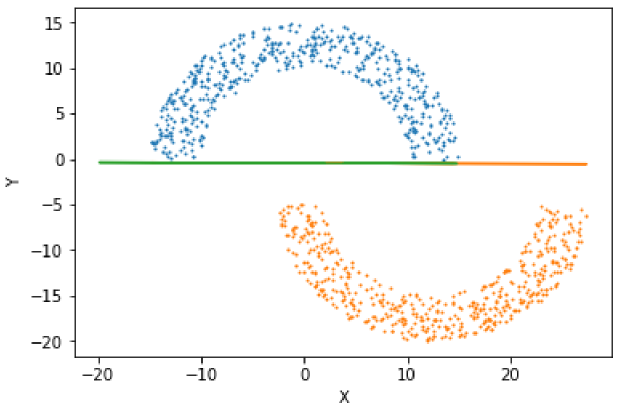
```
total no of iterations 19
```

Out[406]:

```
array([17.        ,  0.15345236, 39.08463044])
```

In [404]:

```
#Plot hypothesis on the separation of the data
def plot_hypothesis(x_axis, w):
    m = -w[1]/w[2] if w[2] != 0 else 0
    b = -w[0]/w[2] if w[2] != 0 else 0
    y_axis = m*x_axis + b
    plt.plot(x_axis, y_axis)
plot_hypothesis(X, weight)
plt.scatter(X1,Y1,s = 1)
plt.scatter(X2,Y2,s = 1)
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```



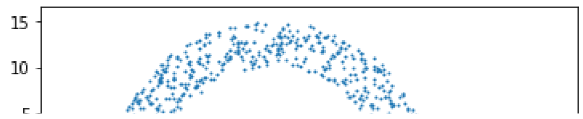
Solution 3.1(a) - The plot above represents the data and final hypothesis

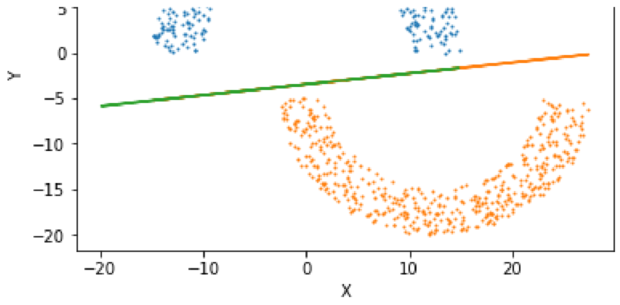
3.1(b) - Repeat Part a using Linear Regression and explain your observations

In [411]:

```
def perceptron(X, Y):
    w = [0,0,0]
    eta = 1
    iter = 0
# calculate inverse of (t(X).X) - where t(X) is transpose of X
    x1 = np.linalg.inv(np.dot(np.transpose(X),X))
# calculate Xcrose by taking x1.t(X)
    Xcross = np.dot(x1,np.transpose(X))
#One-step algorithm, weight is calculated by Xcross.Y
    w = np.dot(Xcross, Y)
    return w


def plot_hypothesis(x_axis, w):
    m = -w[1]/w[2] if w[2] != 0 else 0
    b = -w[0]/w[2] if w[2] != 0 else 0
    y_axis = m*x_axis + b
    plt.plot(x_axis, y_axis)

weight = perceptron(X,y)
print('WLin',weight)
plot_hypothesis(X, weight)
plt.scatter(X1,Y1,s = 1)
plt.scatter(X2,Y2,s = 1)
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

```
WLin [ 0.26671916 -0.00922369  0.07700174]
```





Solution 3.1(b) - As we can see in the plot above, linear regression can be used for the classification of the two data. Infact linear regression does quite a good job in classification. The weights calculated using linear regression is a one step algorithm to compute an optimal solution for weight.

3.3(a)For the double semi circle task in 3.1, set sep =-5 and generate 2000 examples. What will happen when PLA is run on those examples

Solution 3.3(a)- When I tried running the PLA for the case when sep = -5, the algorithm continued to run for a long time without resulting in a successful classification. This is because for sep =-5, there is no way of classifying the data points perfectly without noise. This can be understood by referring the plot below.

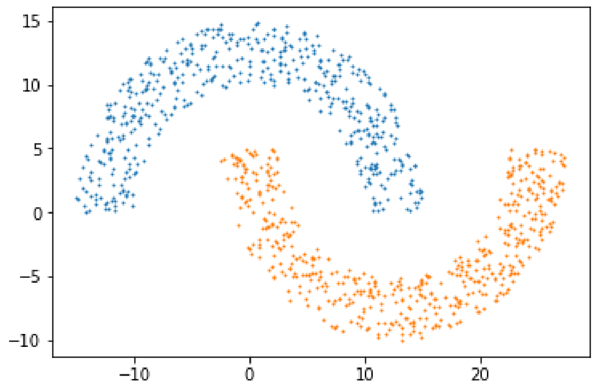
3.3(b) Run the pocket algorithm for 100,000 iterations an plot Ein verser iteration t

In [113]:

```
#parameters
rad = 10
thk = 5
sep = -5

#n data points,(x1,y1) are the coordinates of the top semi-circle
def generatedata(rad,thk,sep,n,x1=0,y1=0):
    # center of the top semi-circle
    X1 = x1
    Y1 = y1

    # center of the bottom semi-circle
    X2 = X1 + rad + thk / 2
    Y2 = Y1 - sep

    # data points in the top semi-circle
    top = []
    # data points in the bottom semi-circle
    bottom = []

    # parameters
    r1 = rad + thk
    r2 = rad

    cnt = 1
    while(cnt <= n):
        #uniformed generated points
        x = np.random.uniform(-r1,r1)
        y = np.random.uniform(-r1,r1)

        d = x**2 + y**2
        if(d >= r2**2 and d <= r1**2):
            if (y > 0):
                top.append([X1 + x,Y1 + y])
                cnt += 1
            else:
                bottom.append([X2 + x,Y2 + y])
                cnt += 1
        else:
            continue

    return top,bottom
```

In [121]:

```
import matplotlib.pyplot as plt
import numpy as np
top,bottom = generatedata(rad,thk,sep,1000)

X1 = [i[0] for i in top]
Y1 = [i[1] for i in top]

X2 = [i[0] for i in bottom]
Y2 = [i[1] for i in bottom]

plt.scatter(X1,Y1,s = 1)
plt.scatter(X2,Y2,s = 1)
plt.show()
```

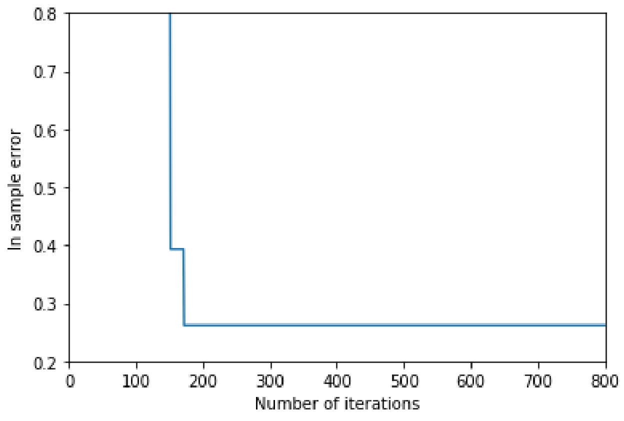


In [115]:

```
# pre-processing the data for (a)
x1 = [[1] + i + [1] for i in top]
x2 = [[1] + i + [-1] for i in bottom]
data = x1 + x2

data = np.array(data)
np.random.shuffle(data)

X = data[:,0:3]
y = data[:,3]
```

In [493]:

```
def perceptron(X, Y):
    w = [0,0,0]
    eta = 1
    sig = []
    iter = 0
    error =1000
    best_error = error
    err = []
    while True:
        sig = np.sign(np.dot(X, np.transpose(w)))
        correct = sig == Y
        if np.all(correct):
            print('total no of iterations',iter)
            return err
        else:
            i = np.argmax(~correct)
            s = np.dot(X,np.transpose(w))
            error = np.absolute((Y[i] -sum(s)/2000))
            if((error<best_error)):
                best_error = error
            w = w + eta*X[i]*Y[i]
        if(iter >= 100000):
            return err
        iter = iter+1
        err.append(best_error)

def plot_hypothesis(x_axis, w):
    y_axis = w
    plt.plot(x_axis, y_axis)

err1 = perceptron(X,y)
iter = np.arange(0,100000)
plot_hypothesis(iter,err1)
plt.xlim(0,800)
plt.ylim(0.2,0.8)
plt.xlabel("Number of iterations")
plt.ylabel("In sample error")
plt.show()
```



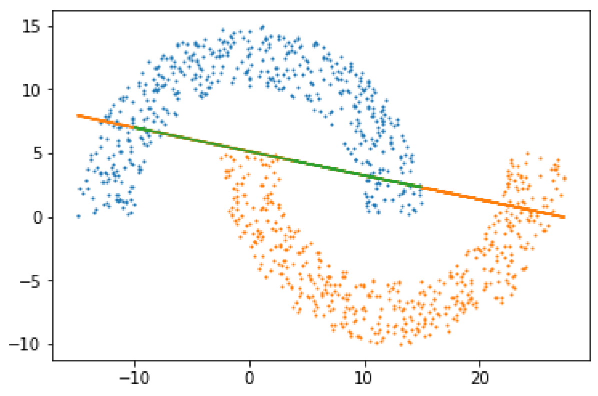
Solution 3.3(b)- Here we have run the pocket algorithm for 100000 iterations and as we can see from the plot above, the in sample error decreases monotonously as the number of iterartions increase.

3.3(c) Plot the data and the final hypothesis in part b.

In [631]:

```
import timeit
def perceptron(X, Y):
    start = timeit.default_timer()
    w = [0,0,0]
    eta = 1
    iter = 0
    error =0.0
    err1=0
    while True:
        sig = np.sign(np.dot(X, np.transpose(w)))
        correct = sig == Y
        if np.all(correct):
            print('total no of iterations',iter)
            return w
        else:
            i = np.argmax(~correct)
            s = np.dot(X,np.transpose(w))
            w = w + eta*X[i]*Y[i]
            error = np.mean(np.sqrt(((Y[i] - s)) **2))/2000
            err1 = error +err1
        if(iter == 100000):
            print('Computation time using Pocket algorithm: ',timeit.default_timer()-start)
            print('Computation time using Pocket algorithm: ',timeit.default_timer()-start)
            print(err1/100000)
            return w
        iter = iter+1
def plot_hypothesis(x_axis, w):
    m = -w[1]/w[2] if w[2] != 0 else 0
    b = -w[0]/w[2] if w[2] != 0 else 0
    y_axis = m*x_axis + b
    plt.plot(x_axis, y_axis)

weight = perceptron(X,y)
plot_hypothesis(X, weight)
plt.scatter(X1,Y1,s = 1)
plt.scatter(X2,Y2,s = 1)
plt.show()
```

```
Computation time using Pocket algorithm:  6.756924647660526
0.22712831640151848
```



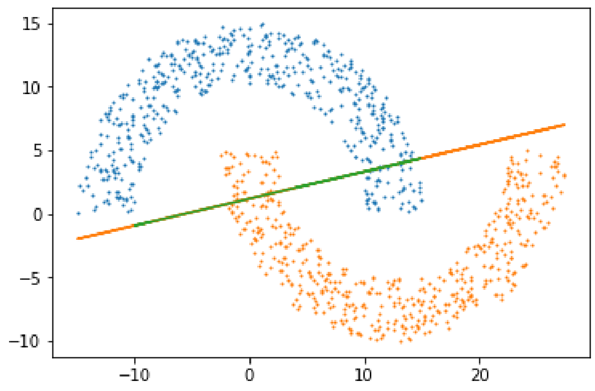
Solution 3.3(c) - The plot above shows the final hypothesis when run using pocket algorith for 100,000 iterations

3.3(d) Use the linear regression algorithm to obtain the weights and compare this result with the pocket algorithm in terms of computation time and quality of solution

In [632]:

```
def perceptron(X, Y):
    start = timeit.default_timer()
    w = [0,0,0]
    eta = 1
    iter = 0
# calculate inverse of (t(X).X) - where t(X) is transpose of X
    x1 = np.linalg.inv(np.dot(np.transpose(X),X))
# calculate Xcrosee by taking x1.t(X)
    Xcross = np.dot(x1,np.transpose(X))
#One-step algorithm, weight is calculated by Xcross.Y
    w = np.dot(Xcross, Y)
    print('Computation time using Linear Regression: ',timeit.default_timer()-start)
    s = np.dot(X,np.transpose(w))
    error = np.mean(np.square(Y-s))
    print('Error measure: ',error)
    return w
def plot_hypothesis(x_axis, w):
    m = -w[1]/w[2] if w[2] != 0 else 0
    b = -w[0]/w[2] if w[2] != 0 else 0
    y_axis = m*x_axis + b
    plt.plot(x_axis, y_axis)

weight = perceptron(X,y)
plot_hypothesis(X, weight)
plt.scatter(X1,Y1,s = 1)
plt.scatter(X2,Y2,s = 1)
plt.show()
```

```
Computation time using Linear Regression:  0.0008699467380210099
Error measure:  0.28213595762640886
```



Solution 3.3(d) From the above two plots, we can see that the computation time using Linear Regression: 0.00086 whearas the computation time using pocket algorithm is 6.7569. On the other hand the In sample error for linear regression is 0.2821 and the in-sample error for pocket algorithm is 0.2271. So time wise linear regression is much better and quality wise pocket algorithm is better

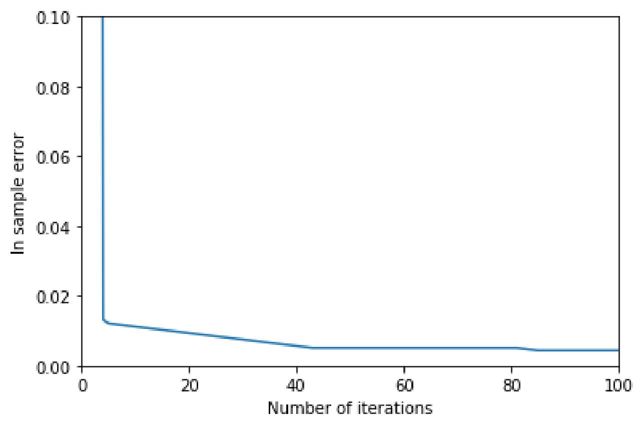
3.3(e) repeat b-e with 3rd order polynomial

In [11]:

```
import pandas as pd
new = pd.DataFrame(data =[X[:,0],X[:,1],X[:,2], X[:,1]**2, X[:,1]*X[:,2], X[:,2]**2 , X[:,1]**3, X[:
,1]**2 * X[:,2], X[:,1]* X[:,2]**2, X[:,2]**3])
```

In [1009]:

```
def perceptron(X, Y):
    newx = X
    newy = y
    w = [0,0,0,0,0,0,0,0,0,0]
    eta = 1
    sig = []
    iter = 0
    error =0
    best_error = 1000
    err = []
    while True:
        sig = np.sign(np.dot(np.transpose(w),newx))
        correct = sig == newy
        if np.all(correct):
            print('total no of iterations',iter)
            return err
        else:
            i = np.argmax(~correct)
            s = np.dot(newx[i][:],np.transpose(w))
            error = (error+np.mean(np.sqrt(((Y[i] - s)) **2))/2000)/100000
            if((error<best_error and iter>2)):
                best_error = error
            w = w + eta*newx[i][:]*newy[i]
        if(iter == 100000):
            return err
        iter = iter+1
        err.append(best_error)

def plot_hypothesis(x_axis, w):
    y_axis = w
    plt.plot(x_axis, y_axis)
    plt.plot(x_axis, y_axis)

err1 = perceptron(new,y)
iter = np.arange(0,100000)
plot_hypothesis(iter,err1)
plt.xlim(0,100)
plt.ylim(0,0.1)
plt.xlabel("Number of iterations")
plt.ylabel("In sample error")
plt.show()
```



As we can see, the error here is much less than the previous case and continues to reduce as the number of iterations increase

In [33]:

```
import timeit
def perceptron(X, Y):
    start = timeit.default_timer()
    newx = X
    newy = y
    w = [0,0,0,0,0,0,0,0,0,0]
    eta = 1
    iter = 0
    error =0.0
    while True:
        sig = np.sign(np.dot(np.transpose(w),newx))
        correct = sig == newy
        if np.all(correct):
            print('total no of iterations',iter)
            return w
        else:
            i = np.argmax(~correct)
            s = np.dot(newx[i][:],np.transpose(w))
            w = w + eta*newx[i][:]*newy[i]
            error = (error+np.mean(np.sqrt(((Y[i] - s)) **2))/2000)/100000
        if(iter == 100000):
            print('Computation time using Pocket algorithm: ',timeit.default_timer()-start)
            print(error/100000)
            return w
        iter = iter+1

weight = perceptron(new,y)
```

```
Computation time using Pocket algorithm:  159.62321640151663
4.2179417077604425e-08
```

Here the computation time of third order polynomial for pocket algorithm is 159.62 and in-sample error is 4.217e-08 whick is equivqlent to 0.00155

In [45]:

```
def perceptron(X, Y):
    start = timeit.default_timer()
    w = [0,0,0,0,0,0,0,0,0,0]
    eta = 1
    iter = 0
# calculate inverse of (t(X).X) - where t(X) is transpose of X
    x1 = np.linalg.inv(np.dot(np.transpose(X),X))
# calculate Xcrosee by taking x1.t(X)
    Xcross = np.dot(x1,np.transpose(X))
#One-step algorithm, weight is calculated by Xcross.Y
    w = np.dot(Xcross, Y)
    print('Computation time using Linear Regression: ',timeit.default_timer()-start)
    s = np.dot(X,np.transpose(w))
    error = np.mean(np.square(Y-s)/2000)
    print('Error measure: ',error)
    return w
def plot_hypothesis(x_axis, w):
    m = -w[1]/w[2] if w[2] != 0 else 0
    b = -w[0]/w[2] if w[2] != 0 else 0
    y_axis = m*x_axis + b
    plt.plot(x_axis, y_axis)

weight = perceptron(np.transpose(new),y)
```

```
Computation time using Linear Regression:  0.008626216660559294
Error measure:  0.0004940042524050023
```

Here the computation time of third order polynomial is 0.00862 and in-sample error is 0.00049 which is quite low and better than the case with pocket algorithm

In [39]:

```
p1 = [X[:,0]* weight[0],X[:,1]*weight[1],X[:,2]* weight[2], X[:,1]**2*weight[3], X[:,1]*X[:,2]* weig
ht[4], X[:,2]**2* weight[5] , X[:,1]**3* weight[6], X[:,1]**2 * X[:,2] * weight[7], X[:,1]* X[:,2]**
2 * weight[8], X[:,2]**3* weight[9]]
```

In [40]:

```
p2 = X[:,0]* weight[0] + X[:,1]*weight[1] +X[:,2]* weight[2] + X[:,1]**2*weight[3] + X[:,1]*X[:,2]* w
eight[4] + X[:,2]**2* weight[5] + X[:,1]**3* weight[6]+ X[:,1]**2 * X[:,2] * weight[7]+ X[:,1]* X[:,2
]**2 * weight[8]+ X[:,2]**3* weight[9]
```

In [53]:

```
weight
```

Out[53]:

```
array([-9.01685816e-02, -1.94394406e-02, -1.59667890e-02,  1.60718024e-04,
        3.34341196e-03,  5.88966295e-03,  3.50507004e-05, -9.39098964e-05,
       -1.89102466e-04, -2.50219434e-04])
```

In [88]:

```
a = weight * np.transpose(new)
```

In [90]:

```
z =np.polyfit(weight,np.sum(a) ,9)
```

```
C:\Users\kashi\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: RankWarning: Polyfit may be po
orly conditioned
  """Entry point for launching an IPython kernel.
```

In [91]:

```
z
```

Out[91]:

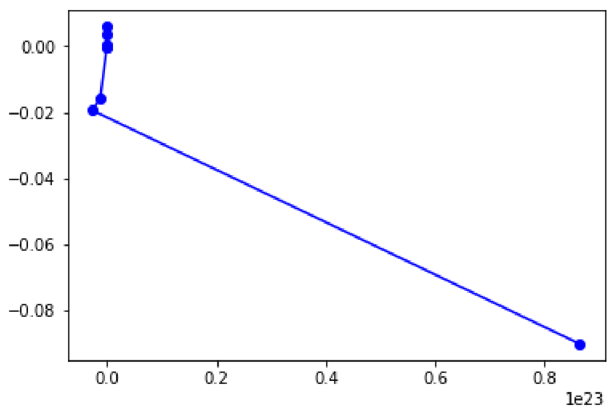
```
array([ 8.64131318e+22, -2.77568052e+21, -1.29336807e+21, -3.10624900e+19,
       -3.96858425e+15,  2.48917566e+15, -6.12004095e+12, -2.90117889e+09,
        3.29540118e+05,  7.43712027e+01])
```

In [128]:

```
plt.plot(z, weight,'bo-')
```

Out[128]:

```
[<matplotlib.lines.Line2D at 0x1d0a7359f98>]
```



3.2- Vary sep in the range of {0.2,0.4,...5}. Generate PLA until converges and start with w=0. Plot sep vs no. of iterations

In [845]:

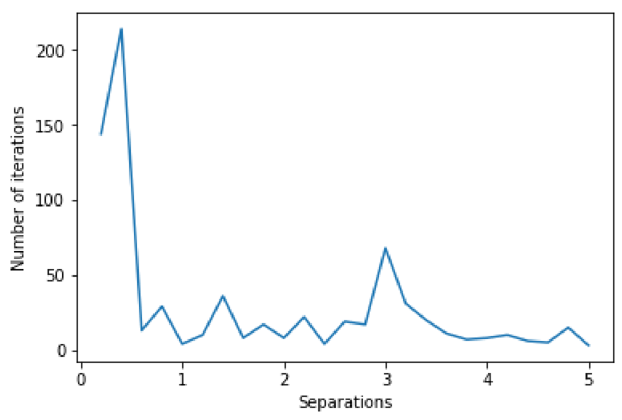
```
#parameters
rad = 10
thk = 5
sep = 0.2
space =[]
iterate = []
def perceptron(X, Y):
    w = [0,0,0]
    eta = 1
    n = 20
    sig = []
    iter = 0
    while True:
        sig = np.sign(np.dot(X, np.transpose(w)))
        correct = sig == Y
        if np.all(correct):
            print('total no of iterations for Sep :',sep, ' is',iter)
            return sep,iter
        else:
            i = np.argmax(~correct)
            w = w + eta*X[i]*Y[i]
        iter = iter+1

while(sep<5.1):
    top,bottom = generatedata(rad,thk,sep,1000)
    X1 = [i[0] for i in top]
    Y1 = [i[1] for i in top]

    X2 = [i[0] for i in bottom]
    Y2 = [i[1] for i in bottom]

    # pre-processing the data for (a)
    x1 = [[1] + i + [1] for i in top]
    x2 = [[1] + i + [-1] for i in bottom]
    data = x1 + x2
    data = np.array(data)
    np.random.shuffle(data)

    X = data[:,0:3]
    y = data[:,3]
    separater, iteration = perceptron(X,y)
    space.append(separater)
    iterate.append(iteration)
    sep = round(sep +0.2 , 2)
```

```
total no of iterations for Sep : 0.2  is 37
total no of iterations for Sep : 0.4  is 5
total no of iterations for Sep : 0.6  is 13
total no of iterations for Sep : 0.8  is 15
total no of iterations for Sep : 1.0  is 9
total no of iterations for Sep : 1.2  is 28
total no of iterations for Sep : 1.4  is 24
total no of iterations for Sep : 1.6  is 10
total no of iterations for Sep : 1.8  is 13
total no of iterations for Sep : 2.0  is 13
total no of iterations for Sep : 2.2  is 8
total no of iterations for Sep : 2.4  is 7
total no of iterations for Sep : 2.6  is 28
total no of iterations for Sep : 2.8  is 41
total no of iterations for Sep : 3.0  is 18
total no of iterations for Sep : 3.2  is 14
total no of iterations for Sep : 3.4  is 12
total no of iterations for Sep : 3.6  is 21
total no of iterations for Sep : 3.8  is 42
total no of iterations for Sep : 4.0  is 8
total no of iterations for Sep : 4.2  is 9
total no of iterations for Sep : 4.4  is 24
total no of iterations for Sep : 4.6  is 11
total no of iterations for Sep : 4.8  is 63
total no of iterations for Sep : 5.0  is 62
```

In the above solution, the list shows the number of iterations taken for the PLA to converge for varying value of sep

In [676]:

```
def plot_hypothesis(x_axis, w):
    y_axis = w
    plt.plot(x_axis, y_axis)
    plt.xlabel('Separations')
    plt.ylabel('Number of iterations')
plot_hypothesis(space,iterate)
```



As we can see from the plot above the with the increase in the separation between the data, the number of iterations taken for the PLA to converge to form the final hypothesis reduces.