Semih Akbayrak
29.03.2015

EE 550 HW2

Perceptron Model

In this assignment, a classification of points by using perceptron model is asked from us. Perceptron Model is an appropriate model for this kind of classification. Because the points are separated from each other with a hyperplane.

weights = rand(3)*2 - 1

coef = 0.6

First of all, I used the rand function to specify a random weight matrix which elements take values between -1 and 1. I also used learning factor as 0.6. I tried the algorithm for different learning factor values and it is not surprising that small values like 0.1 sometimes isn't big step enough to make an exact separation. 0.6 is a good value which supply satisfactory convergence.

```
def samplemachine():
    r = rand(1)
    if r[0]>=0.5:
        x1 = rand(1)
        x2 = rand(1)
        x3 = rand(1)
        d = 1
    else:
        x1 = -rand(1)
        x2 = -rand(1)
        x3 = -rand(1)
        d = -1
    s = []
    s.append(x1[0])
    s.append(x2[0])
    s.append(x3[0])
    s.append(d)
    return s
```

Above you can find my function which produces random points for the 1st and 8th quadrants of the {x1,x2,x3} frame. Choosing opposing quadrants is helpful for the more clear separation.

As it can be seen clearly, for this case, because of choosing points in opposite quadrants ables us to choose threshold T as zero and the separation plane equation becomes w1*x1+w2*x2+w3*x3 = 0 which is passing through the origin.

```
def output(s,w):
    u = s[0]*w[0] + s[1]*w[1] + s[2]*w[2]
    if u>=0:
        y = 1
    else:
        y = -1
    return y
```

Output function behaves as a hard limiter. It first determines the u by summing the muliplications of inputs and corresponding weights. Then it makes classification with hard limiter function.

```
def newweight(wold,coef,error,x):
    wnew = wold + coef*error*x
    return wnew
```

The logic of perceptron model is based on weight changes and energy minimization with respect to weight. After cost minimization calculations, we reach the above formula I applied with newweight function.

```
def train():
    sx1 = []
    sx2 = []
    sx3 = []
    sx4 = []
    sx5 = []
    for i in range (35):
        s = samplemachine()
        sx1.append(s[0])
        sx2.append(s[1])
        sx3.append(s[2])
        y = output(s,weights)
        if s[3] == 1:
            sx4.append("red")
        else:
            sx4.append("blue")
        if y == 1:
```

```
            sx5.append("red")
        else:
            sx5.append("blue")
        if y != s[3]:
            e = s[3] - y
            weights[0] = newweight(weights[0],coef,e,s[0])
            weights[1] = newweight(weights[1],coef,e,s[1])
            weights[2] = newweight(weights[2],coef,e,s[2])
        else:
            e = 0
    fig1 = figure()
    fig2 = figure()
    ax = Axes3D(fig1)
    ax.scatter(sx1,sx2,sx3,c=sx4,marker='o')
    ax2 = Axes3D(fig2)
    ax2.scatter(sx1,sx2,sx3,c=sx5,marker='x')
    show()
```

Above is the function where the learning takes place. In the assignment, it is stated to go over the sample set once, thus I defined a for loop which makes calculation for 35 times. Each time it generates a sample patterns, and calculates new weight matrix. Because it is an online learning algorithm, each additional pattern affects the weight matrix and changes it of course desired output and actual output different. I used sx arrays for plotting, they are not directly related with the perceptron model and learning algorithm.

```
def test(n):
    tx1 = []
    tx2 = []
    tx3 = []
    tx4 = []
    tx5 = []
    for i in range (n):
        s = samplemachine()
        print s
        tx1.append(s[0])
        tx2.append(s[1])
        tx3.append(s[2])
        if s[3] == 1:
            tx4.append("red")
        else:
            tx4.append("blue")
        y = output(s,weights)
        if y == 1:
```

```
            tx5.append("red")
        else:
            tx5.append("blue")
        print y

    fig3 = figure()
    fig4 = figure()
    ax = Axes3D(fig3)
    ax.scatter(tx1,tx2,tx3,c=tx4,marker='o')
    ax2 = Axes3D(fig4)
    ax2.scatter(tx1,tx2,tx3,c=tx5,marker='x')
    show()
```

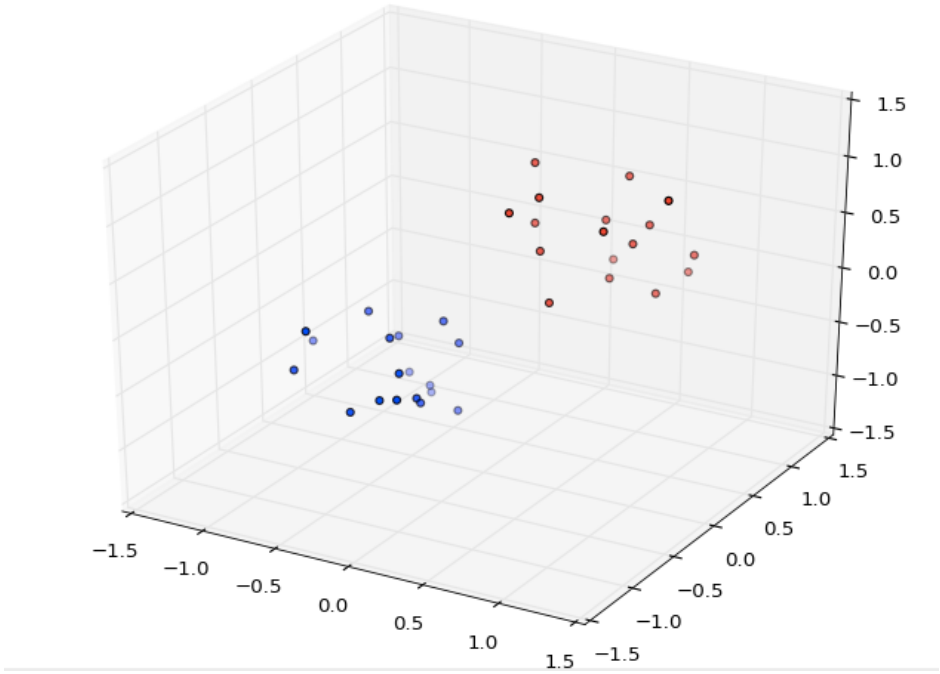
Test function has exactly same principle with the training, but this time of course it doesn't change the weight matrix because we normally don't know the desired output for test inputs. I just used desired output for comparetion whether algorithm works correctly.

```
def execute(n):
    train()
    test(n)
```

That is the execution function and let's execute the algorithm with five test inputs.

```
execute(5)
```

Fig1



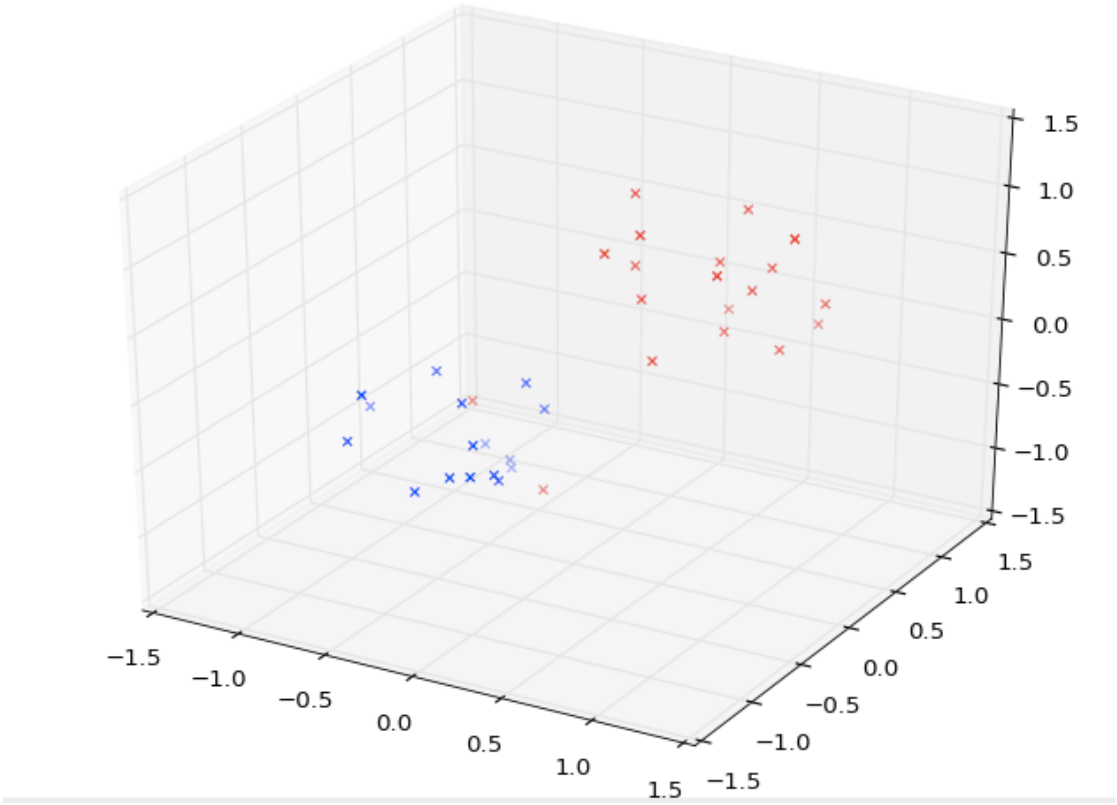


Fig2

Fig1 shows the desired classification of sample patterns in x,y,z coordinate system. In Fig2 you see how the algorithm perceives and classifies the sample patterns. As it can be realized, two of the sample patterns are classified wrong.

But overall weight matrix value is good enough to classify 5 test inputs correctly as you can find below. Fig3 represents how the test inputs should be classified and Fig4 shows the classification of the algorithm.

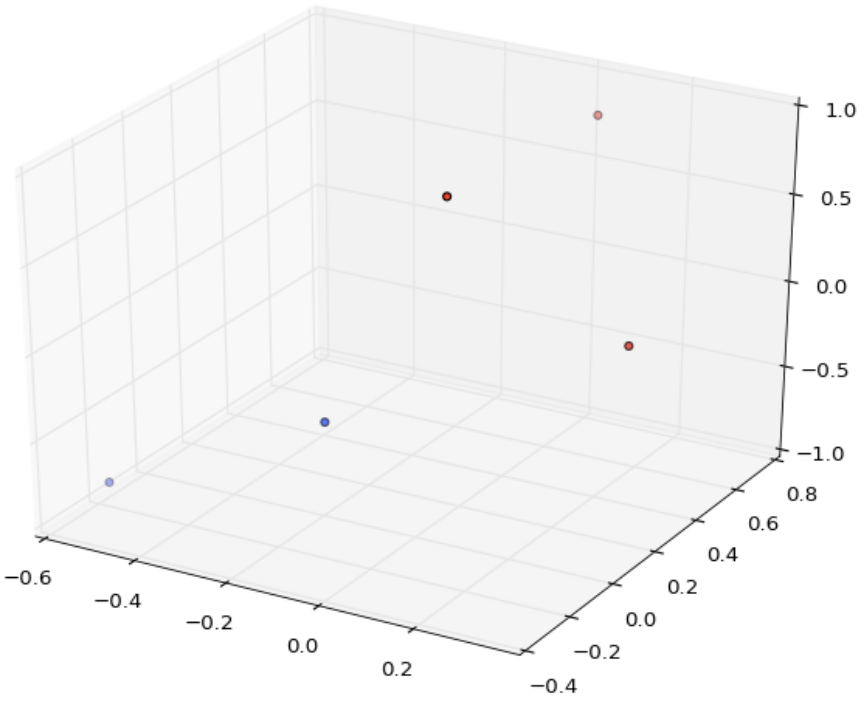


Fig3

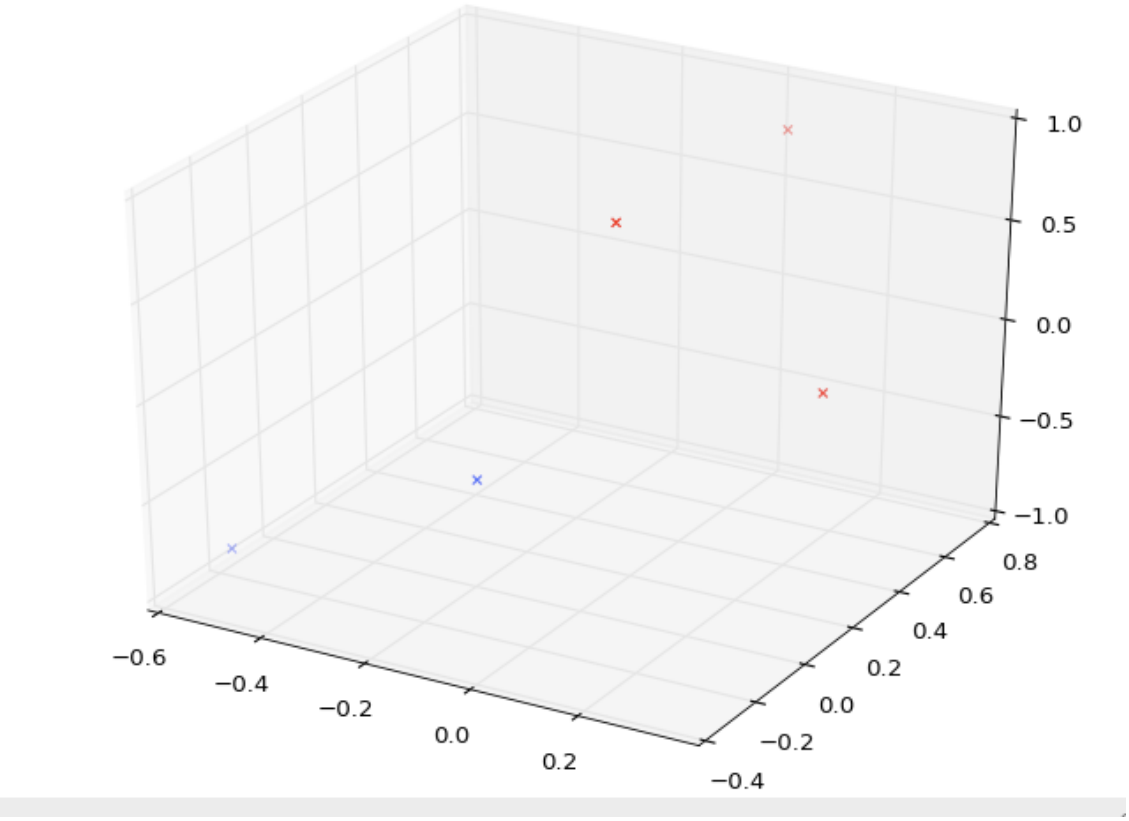


Fig4

Lastly I wrote a code to display the separation plane, and to plot it I utilized the fact that plane is passing through origin and weight matrix is normal to that plane.

```
point  = np.array([0, 0, 0])
    normal = weights
    d = -point.dot(normal)
    xx, yy = np.meshgrid(range(-1,2), range(-1,2))
    z = (-normal[0] * xx - normal[1] * yy - d) * 1. /normal[2]
    plt3d = plt.figure().gca(projection='3d')
    plt3d.plot_surface(xx, yy, z)
    plt.show()
```

Below you can find corresponding figures

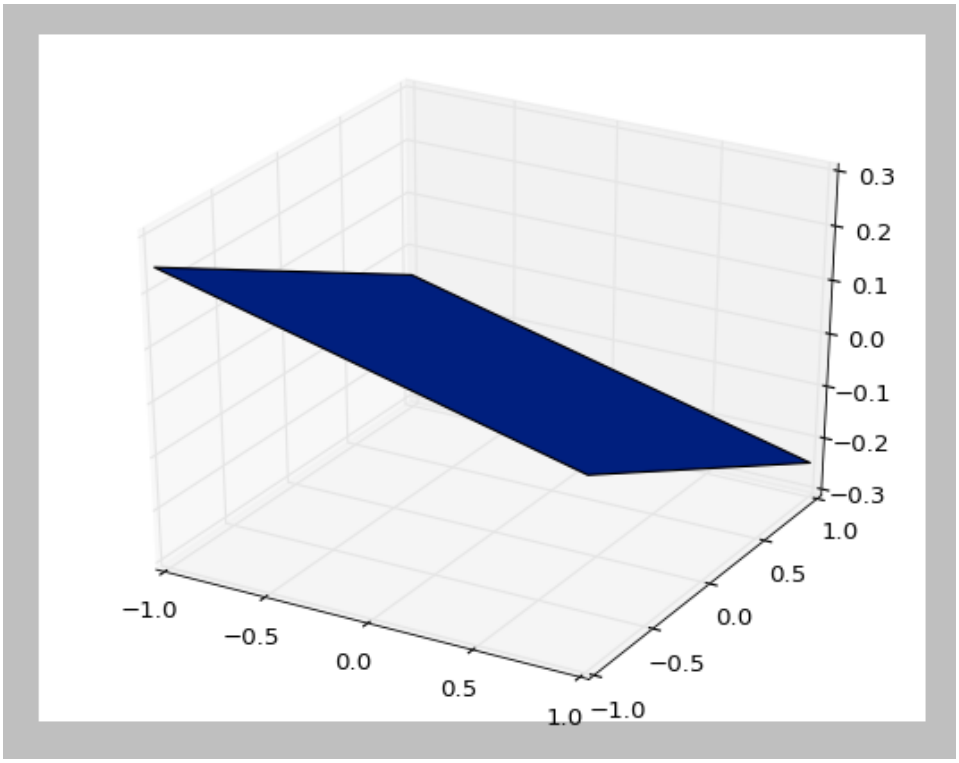

0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

fig5

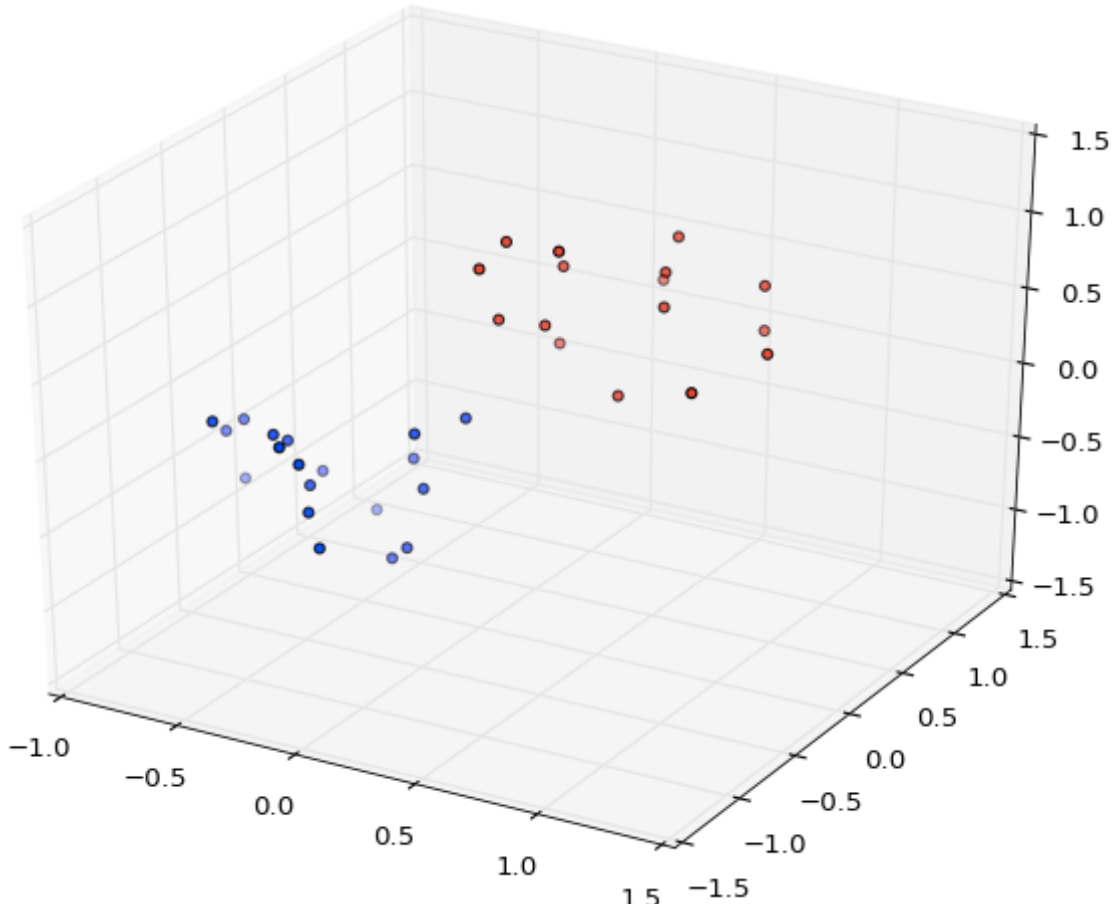

1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.0
−0.5
0.0
0.5
1.0

Fig6

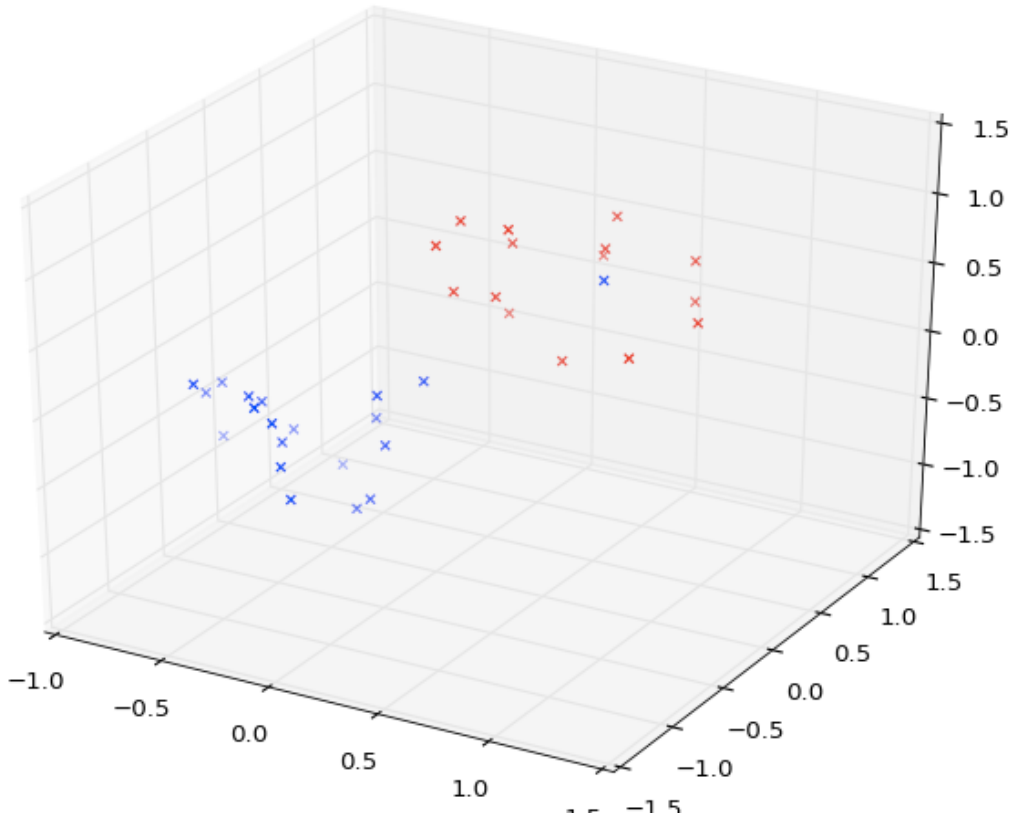

1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
1.0
0.5
0.0
−0.5
−1.0

Fig7

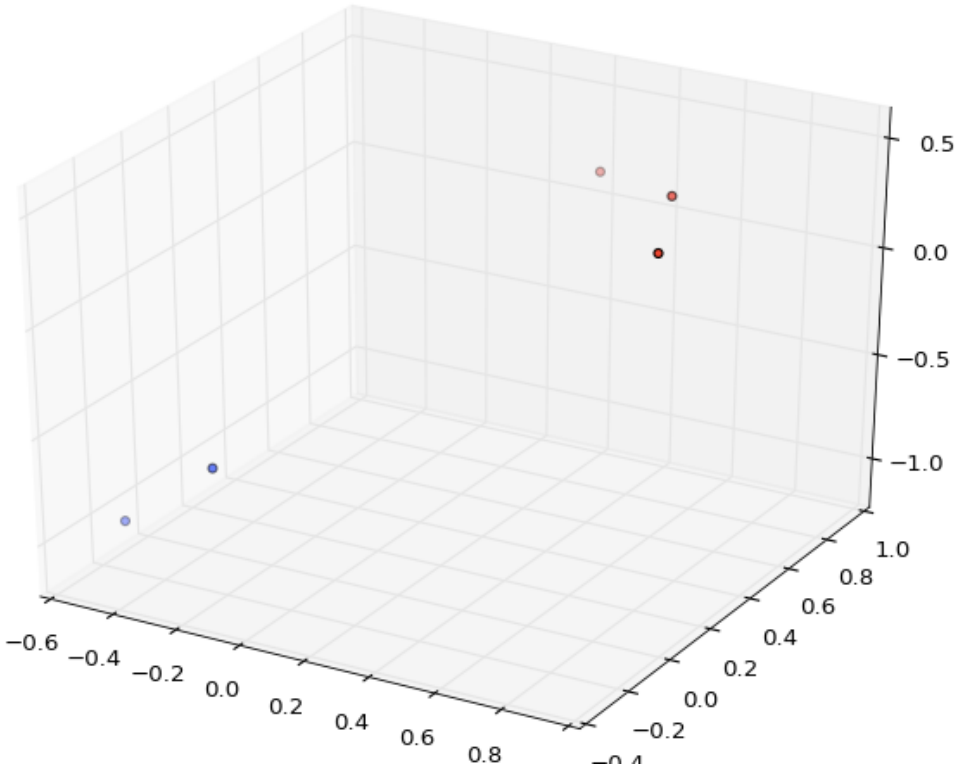

0.5
0.0
−0.5
−1.0
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
0.8
0.6
0.4
0.2
0.0
−0.2

Fig8

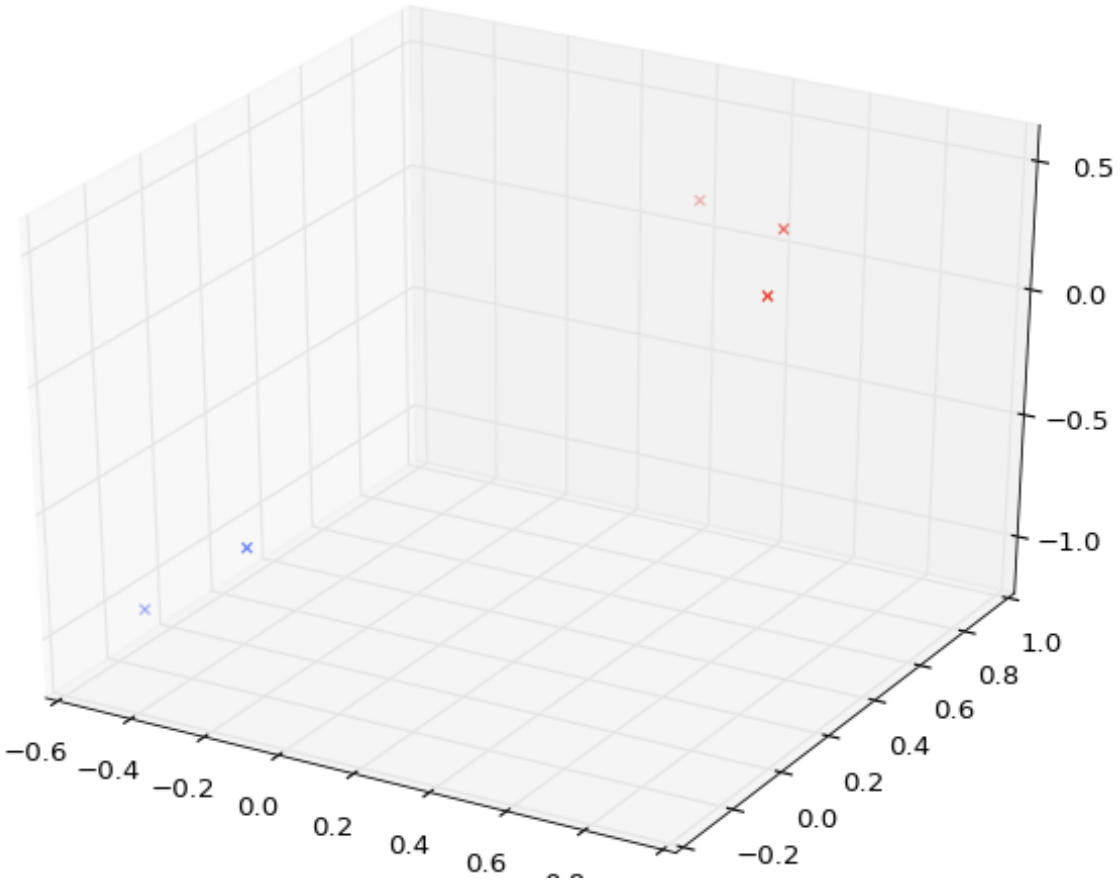


Fig9

Entire Code

```
import numpy as np
from numpy import array, zeros
from pylab import rand, plot, show,figure
from mpl_toolkits.mplot3d import Axes3D
import math
import matplotlib.pyplot as plt

weights = rand(3)*2 - 1
coef = 0.6

def samplemachine():
    r = rand(1)
    if r[0]>=0.5:
        x1 = rand(1)
```

```
        x2 = rand(1)
        x3 = rand(1)
        d = 1
    else:
        x1 = -rand(1)
        x2 = -rand(1)
        x3 = -rand(1)
        d = -1
    s = []
    s.append(x1[0])
    s.append(x2[0])
    s.append(x3[0])
    s.append(d)
    return s

def output(s,w):
    u = s[0]*w[0] + s[1]*w[1] + s[2]*w[2]
    if u>=0:
        y = 1
    else:
        y = -1
    return y

def newweight(wold,coef,error,x):
    wnew = wold + coef*error*x
    return wnew


def train():
    sx1 = []
    sx2 = []
    sx3 = []
    sx4 = []
    sx5 = []
    for i in range (35):
        s = samplemachine()
        sx1.append(s[0])
        sx2.append(s[1])
        sx3.append(s[2])
        y = output(s,weights)
        if s[3] == 1:
            sx4.append("red")
```

```
        else:
            sx4.append("blue")
        if y == 1:
            sx5.append("red")
        else:
            sx5.append("blue")
        if y != s[3]:
            e = s[3] - y
            weights[0] = newweight(weights[0],coef,e,s[0])
            weights[1] = newweight(weights[1],coef,e,s[1])
            weights[2] = newweight(weights[2],coef,e,s[2])
        else:
            e = 0
    fig1 = figure()
    fig2 = figure()
    ax = Axes3D(fig1)
    ax.scatter(sx1,sx2,sx3,c=sx4,marker='o')
    ax2 = Axes3D(fig2)
    ax2.scatter(sx1,sx2,sx3,c=sx5,marker='x')
    show()

    point  = np.array([0, 0, 0])
    normal = weights
    d = -point.dot(normal)
    xx, yy = np.meshgrid(range(-1,2), range(-1,2))
    z = (-normal[0] * xx - normal[1] * yy - d) * 1. /normal[2]
    plt3d = plt.figure().gca(projection='3d')
    plt3d.plot_surface(xx, yy, z)
    plt.show()

def test(n):
    tx1 = []
    tx2 = []
    tx3 = []
    tx4 = []
    tx5 = []
    for i in range (n):
        s = samplemachine()
        print s
        tx1.append(s[0])
        tx2.append(s[1])
        tx3.append(s[2])
```

```
        if s[3] == 1:
            tx4.append("red")
        else:
            tx4.append("blue")
        y = output(s,weights)
        if y == 1:
            tx5.append("red")
        else:
            tx5.append("blue")
        print y
        #if y != s[3]:
        #    e = s[3] - y
        #    weights[0] = newweight(weights[0],coef,e,s[0])
        #    weights[1] = newweight(weights[1],coef,e,s[1])
        #    weights[2] = newweight(weights[2],coef,e,s[2])
        #else:
        #    e = 0
    fig3 = figure()
    fig4 = figure()
    ax = Axes3D(fig3)
    ax.scatter(tx1,tx2,tx3,c=tx4,marker='o')
    ax2 = Axes3D(fig4)
    ax2.scatter(tx1,tx2,tx3,c=tx5,marker='x')
    show()


def execute(n):
    train()
    test(n)


execute(5)
```