homework2

April 27, 2016

1 Homework 2

Titipat Achakulvisut

1.1 Problem 1: Classifier – back propagation

Use the code below to create three data clusters which are vertically stacked and therefore not linearly separable, as we discussed in class. Your job is to create code implementing back propagation for a two layer neural network which can perform this classification. Use a network with 4 hidden units, as indicated in the shell code. Don't worry about cross validation and all that – feel free to just use all the data.

In [1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

In [2]:
```
sd = .85
X1 = np.vstack((np.random.normal(0, sd, size=(50,1)),
                np.random.normal(0, sd, size=(50,1)),
                np.random.normal(0, sd, size=(50,1))))
X2 = np.vstack((np.random.normal(0, sd, size=(50,1)),
                np.random.normal(5, sd, size=(50,1)),
                np.random.normal(10, sd, size=(50,1))))
# X3 = np.ones((150, 1))
X = np.concatenate((X1, X2), axis=1)
y = np.zeros((150, 3))
y[0:50, 0] = 1
y[50:100, 1] = 1
y[100:150, 2] = 1
```

In [3]:
```
plt.scatter(X[:,0], X[:,1], c=y);
```

In [4]:
```
def sigmoid(z):
    return 1/(1 + np.exp(-z))

def sigmoid_grad(z):
    return sigmoid(z)*(1 - sigmoid(z))
```

In [5]:
```
ninput = 2
nhidden = 4
noutput = 3
W = np.random.uniform(-1, 1, size=(ninput+1, nhidden)) - 0.5
V = np.random.uniform(-1, 1, size=(nhidden+1, noutput)) - 0.5
```

In [6]:
```
def predict(W, V, X):
    """Do forward propagation given first and second layers"""
    n, m = X.shape
    a1 = np.concatenate((np.ones((n,1)), X), axis=1)
    z2 = a1.dot(W)
    a2 = sigmoid(z2)
    a2 = np.concatenate((np.ones((n,1)), a2), axis=1)
    z3 = a2.dot(V)
    a3 = sigmoid(z3)
    h = a3
    return h
```

In [7]:
```
def compute_grad_bp(W, V, X, Y):
    """
```

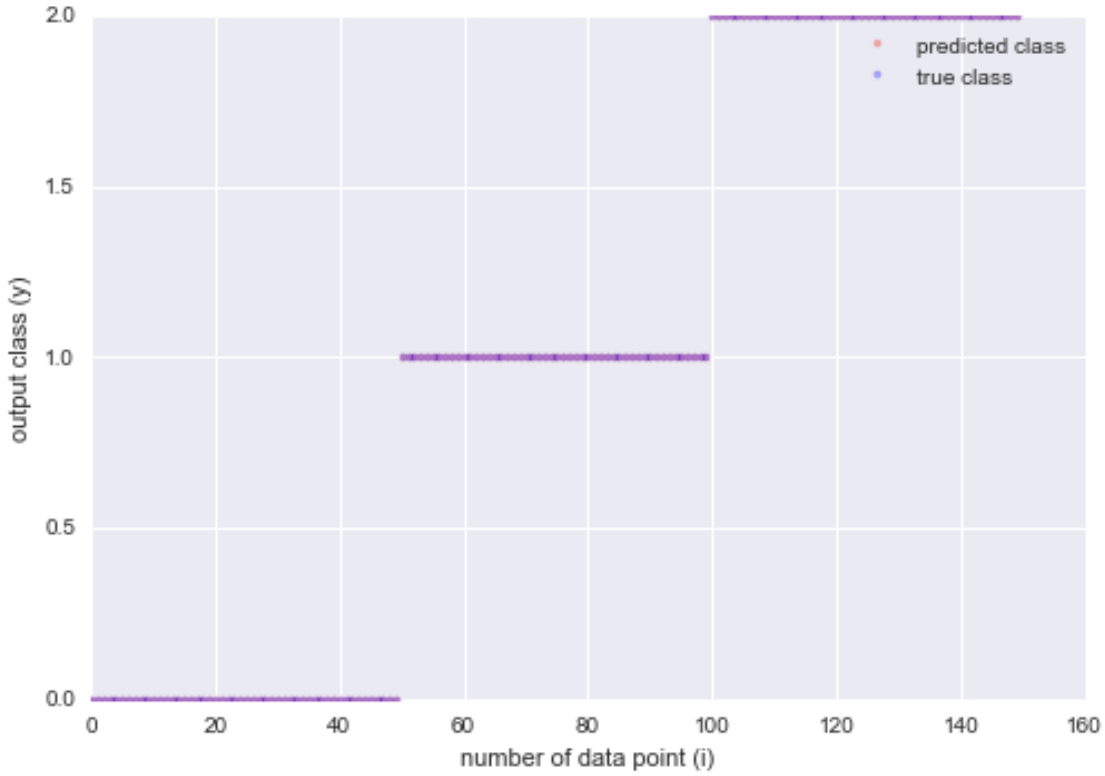
```
    Compute gradient of NN parameters for one iteration
    using back propagation
    """
    # initialize few parameters
    n, m = X.shape

    dW = np.zeros_like(W)
    dV = np.zeros_like(V)

    a1 = np.concatenate((np.ones((n,1)), X), axis=1)
    z2 = a1.dot(W)
    a2 = sigmoid(z2)
    a2 = np.concatenate((np.ones((n,1)), a2), axis=1)
    z3 = a2.dot(V)
    a3 = sigmoid(z3)
    h = a3

    # back propagation
    delta3 = h - Y
    delta2 = delta3.dot(V[1::,:].T)*sigmoid_grad(z2)

    dW = (1/m)*((delta2.T).dot(a1)).T
    dV = (1/m)*((delta3.T).dot(a2)).T

    return dW, dV
```

In [8]:
```
# gradient descent to find final neural nets parameter
n_iter = 4000
mu = 0.01
dW = np.zeros_like(W)
dV = np.zeros_like(V)
for i in range(n_iter):
    dW, dV = compute_grad_bp(W, V, X, y)
    W = W - mu*dW
    V = V - mu*dV
```

In [9]:
```
y_hat = predict(W, V, X)
print('Predicted output class: \n', y_hat.argmax(axis=1))
print( )
```

```
Predicted output class:
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

In [10]:
```
plt.plot(y_hat.argmax(axis=1), '.r', alpha=0.3)
plt.plot(y.argmax(axis=1), '.b', alpha=0.3) # output class for each datapoint
plt.xlabel('number of data point (i)')
plt.ylabel('output class (y)')
plt.legend(['predicted class', 'true class'])
plt.show()
```



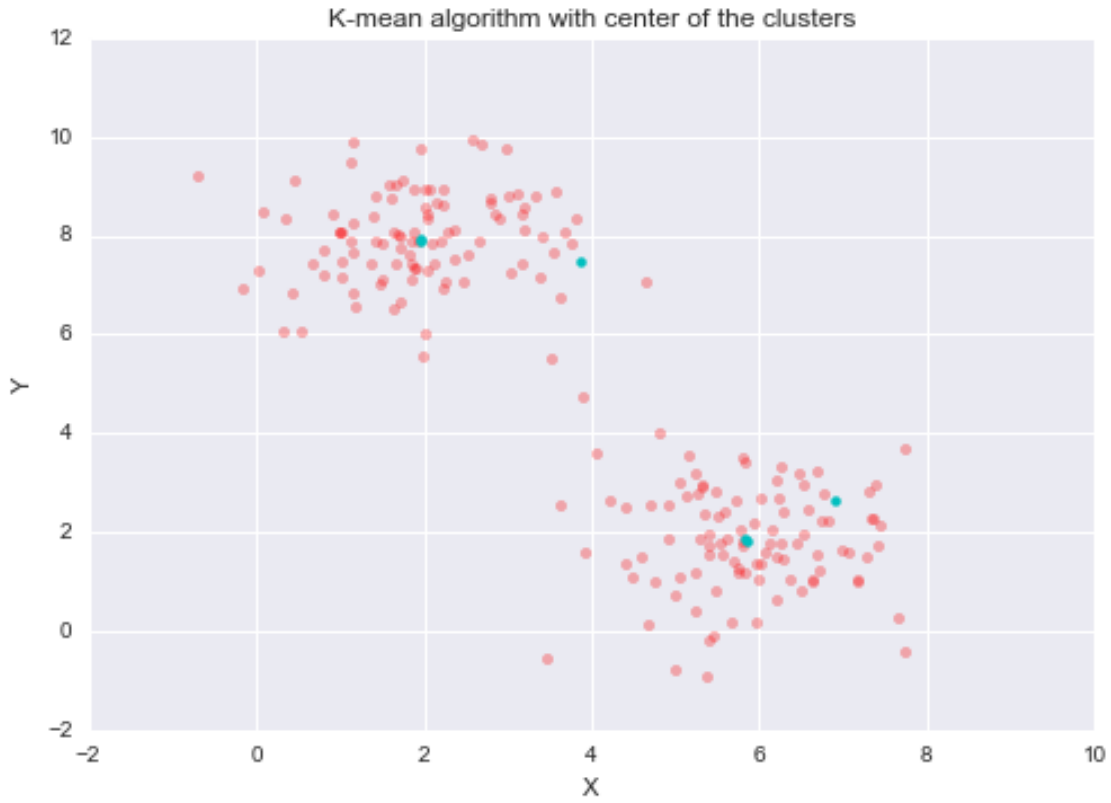
1.2 Problem 2: Unsupervised learning – k-means clustering

In each of the next problems you are given data that appears to be produced by two clusters (use the code in ps1 `datasets.m`, 2a). Your job in this problem is to use k-means clustering to classify the data according to two clusters. Plot the trajectory of the means as they are updated over iterations, with this trajectory superimposed over the data sets.

```
In [11]: X1 = np.hstack((np.random.normal(6, 1, size=(100,1)),
                          np.random.normal(2, 1, size=(100, 1))))
         X2 = np.hstack((np.random.normal(2, 1, size=(100,1)),
                          np.random.normal(8, 1, size=(100, 1))))
         X = np.vstack((X1, X2))
```

```
In [12]: def update(X, centers, alpha=0.05):
             """Update K-mean center and compute cost in that iteration"""
             # number of cluster
             K = len(centers)
             # compute distance
             D = np.vstack([np.linalg.norm(X - center, axis=1) for center in centers])
             clusters = np.argmin(D.T, axis=1) # assign clusters
             centers = np.vstack([X[clusters==k].mean(axis=0) for k in range(K)])

             # compute cost of K-mean
             J = np.sum([np.sum(np.linalg.norm(X[clusters==k] - centers[k], axis=1))
                         for k in range(K)])
             return centers, J, clusters
```

```
In [13]: def random_centers(X, K=2):
             """
             Randomly generate K centers from data X
             """
             X_min = X.min(axis=0)
             X_max = X.max(axis=0)
             centers = []
             for k in range(K):
                 center = [np.random.uniform(X_min[i], X_max[i]) for i in range(len(X_min))]
                 centers.append(center)
             return np.array(centers)

In [14]: n_iter = 100
         n_cluster = 2
         centers = random_centers(X, K=n_cluster) # random center
         print('Initial centers are \n', centers)
         clusters = np.zeros(X.shape[0])
         u = True
         J_list = []
         centers_list = []
         while u:
             clusters_prev = clusters
             centers_list.append(centers)
             centers, J, clusters = update(X, centers) # compute new center and cost
             J_list.append(J)
             if np.all(clusters_prev == clusters):
                 u = False

Initial centers are
 [[ 6.90985114  2.63489775]
 [ 3.88114647  7.45748333]]

In [15]: plt.scatter(X[:, 0], X[:, 1], color='r', alpha=0.3)
         for c in centers_list:
             for k in range(n_cluster):
                 plt.scatter(c[k][0], c[k][1], color='c', lw = 0)
         plt.xlabel('X')
         plt.ylabel('Y')
         plt.title('K-mean algorithm with center of the clusters')
         plt.show()
         print('Center for final cluster are \n',
               list(centers_list[-1][0]), ' and \n',
               list(centers_list[-1][1]))
```




```
Center for final cluster are
 [5.8350106130994046, 1.845847253250521]  and
 [1.9500944871562012, 7.9396316730666197]
```

1.3 Problem 3: Unsupervised learning – ML gradient descent

Your job is to classify the same dataset as used in problem 2 into two clusters using ML gradient descent to update the values of mean and standard deviation of two Gaussian clusters. Assume that the prior probabilities of each cluster are already given and fixed at 0.5 each.

Ans. Basically, we have mixure of Gaussians on variable $\mathbf{x}$ as follows:

$$p(\mathbf{x}) = \sum_{k=1}^{K} \rho_k \frac{1}{\sqrt{det(2\pi\Sigma_k)}} \exp[-\frac{1}{2}(\mathbf{x}-\mu_k)^T \sigma_k^{-1}(\mathbf{x}-\mu_k)]$$

where ρ_k sum up to 1. Now the log-likelihood will be, as following

$$\ln p(\mathbf{x}) = \sum_{k=1}^{K} \log(\rho_k) - \log(C) - \log(det(2\pi\Sigma_k)) - \frac{1}{2}((\mathbf{x}-\mu_k)^T \Sigma^{-1}(\mathbf{x}-\mu_k))$$

where C in this case is a constant term (in front of Gausian distribution)

Thus the gradient of log-likelihood with respect to mean and variance (giving we assuming that we know ρ_k) is as follows

$$\frac{\partial \ln p(\mathbf{x})}{\partial \mu_k} = \Sigma_j^{-1}(\mathbf{x}-\mu_k)$$

and

$$\frac{\partial \ln p(\mathbf{x})}{\partial \Sigma_k} = \frac{1}{2}[-\Sigma_k^{-1} + \Sigma_k^{-1}(\mathbf{x}-\mu_k)(\mathbf{x}-\mu_k)^T\Sigma_k^{-1}]$$

note (some matrix tricks):

- use this trick in finding gradient with respect to mean $\frac{\partial}{\partial \mathbf{x}}(\mathbf{x}-\mathbf{s})^T\mathbf{W}(\mathbf{x}-\mathbf{s}) = 2\mathbf{W}(\mathbf{x}-\mathbf{s})$
- use this trick in finding gradient with respect to covariance matrix (in this case, we know that covariace matrix is symmetric so we can get rid of transpose) $\frac{\partial ln|\det(\mathbf{X})|}{\partial(\mathbf{X})} = (X^{-1})^T$

In this homework, we consider special case where Spherical covariance is as follows $\Sigma = \sigma_i^2 I$
The likelihood and log-likelihood can be written as follows:

$$p(\mathbf{x}) = \sum_{k=1}^{K} \rho_k \frac{1}{\sqrt{2\pi\sigma_k}^{2d}} \exp[-\frac{1}{2}\frac{(\mathbf{x}-\mu_k)^T(\mathbf{x}-\mu_k)}{\sigma_k^2}]$$

$$\ln p(\mathbf{x}) = \sum_{k=1}^{K} \log(\rho_k) - \log(C) - d\log\sigma_k - \frac{1}{2}\frac{(\mathbf{x}-\mu_k)^T(\mathbf{x}-\mu_k)}{\sigma_k^2}$$

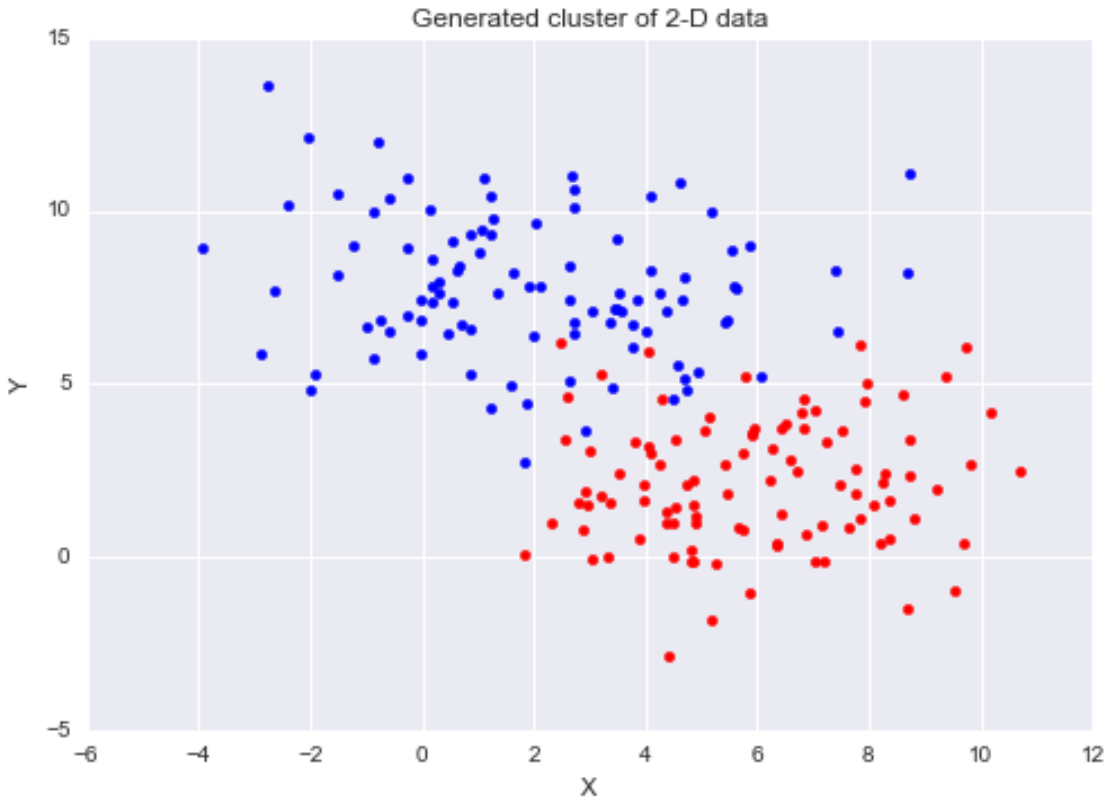
where d is dimension of the Gaussian (in our case, $d = 2$)
Therefore, the partial derivative with respect to mean and standard deviation is as follows:

$$\frac{\partial \ln p(\mathbf{x})}{\partial \mu_k} = \frac{(\mathbf{x}-\mu_k)}{\sigma_k^2}$$

$$\frac{\partial \ln p(\mathbf{x})}{\partial \sigma_k} = -\frac{d}{\sigma_k} + \frac{(\mathbf{x}-\mu)^T(\mathbf{x}-\mu)}{\sigma_k^3}$$

In [16]:
```
import numpy.linalg as la
def inv(A):
    """return inverse of input matrix A"""
    return la.inv(A)
```

In [17]:
```
# generate data for problem
X1 = np.hstack((np.random.normal(6, 2, size=(100,1)),
                np.random.normal(2, 2, size=(100, 1))))
X2 = np.hstack((np.random.normal(2, 3, size=(100,1)),
                np.random.normal(8, 2, size=(100, 1))))
X = np.vstack((X1, X2))
y = np.vstack([np.ones((100, 1)), np.zeros((100, 1))]).flatten()
plt.scatter(X[(y==1).flatten(), 0], X[(y==1).flatten(), 1], color='r')
plt.scatter(X[(y==0).flatten(), 0], X[(y==0).flatten(), 1], color='b')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Generated cluster of 2-D data')
plt.show()
print('Mean for each clusters is \n',
      np.mean(X1, axis=0),
      np.mean(X2, axis=0))
```




```
Mean for each clusters is
 [ 5.92435433  2.15143611] [ 1.99348394  7.75918159]
```

This is the case when we want to Gaussian mixure of mean and arbitrary covariance matrix. Here we print out final center of clusters and covariance matrix.

```
In [18]: def data_cluster_matrix(x, centers, sigma):
             """compute class of data point x for all centers and sigma"""
             p = [float(rbf_kernel_matrix(x, center, s))
                  for (center, s) in zip(centers, sigma)]
             c = np.argmax(p)
             return c

In [19]: def rbf_kernel_matrix(x, y, S):
             """
             Compute radial basis kernel of two vector
             k(x, y) = exp(-||x-y||^2/s^2)
             """
             nom = np.exp(-(x-y).T.dot(np.linalg.inv(S)).dot(x-y))
             denom = np.sqrt(np.linalg.det(2.*np.pi*S))
             return nom/denom

In [23]: mu = 0.005
         mu_sigma = 0.001
         n_iter = 150
         # centers = random_centers(X, K=2)
```

```
        centers = np.array([[3., 8.], [5., 4.]])
        centers = [np.atleast_2d(c).T for c in centers]
        sigma = [np.random.uniform(0.5, 1.5, 2)*np.eye(2), np.random.uniform(0.5, 1.5, 2)*np.eye(2)]
        print('Initialized center = \n', np.array(centers))
        print('Initializerd covariance matrix = \n', np.array(sigma))

        for n in range(n_iter):
            for i in range(len(X)):
                x = np.atleast_2d(X[i]).T
                c = data_cluster_matrix(x, centers, sigma)
                centers[c] = centers[c] + mu*(inv(sigma[c]).dot(x - centers[c]))
                c = data_cluster_matrix(x, centers, sigma)
                cov = (x - centers[c]).dot((x - centers[c]).T)
                sigma[c] = sigma[c] + mu_sigma*(-inv(sigma[c]) + inv(sigma[c]).dot(cov).dot(inv(sigma[
        print('Final center = \n', np.array(centers))
        print('Final covariance matrix = \n', np.array(sigma))
        out = np.array([data_cluster_matrix(np.atleast_2d(x).T, centers, sigma) for x in X])
        print('Output class for each data points: \n', out)
```

```
Initialized center =
 [[[ 3.]
  [ 8.]]

 [[ 5.]
  [ 4.]]]
Initializerd covariance matrix =
 [[[ 0.58919867  0.        ]
  [ 0.          0.9038983 ]]

 [[ 1.48167468  0.        ]
  [ 0.          1.2915064 ]]]
Final center =
 [[[ 1.81923247]
  [ 7.91747142]]

 [[ 5.91971954]
  [ 2.22827303]]]
Final covariance matrix =
 [[[ 4.7361565  -0.22237403]
  [-0.22237403  3.26363719]]

 [[ 3.77239754  0.40246277]
  [ 0.40246277  3.2523976 ]]]
Output class for each data points:
 [1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 0 0 0
 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

here is when we want to estimate mean and standard deviation for each clusters assuming spherical Gaussian distribution $\Sigma_k = \sigma_k^2 I$

```
In [24]: def rbf_kernel(x, y, s):
             """
```

```
            Compute radial basis kernel of two vector
            k(x, y) = exp(-||x-y||^2/s^2)
            """
            d = len(x)
            x = np.array(x)
            y = np.array(y)
            return (1./(np.sqrt(2*np.pi*(s**(2*d)))))*np.exp(-np.linalg.norm(x - y)**2/(s**2))
```

In [25]:
```
def data_cluster(x, centers, sigma):
    """
    compute probability belonging to each clusters
    and class that data belong to
    """
    p = [rbf_kernel(x, center, s) for (center, s) in zip(centers, sigma)]
    p = np.array(p)
    c = np.argmax(p) # class of the data point
    return int(c)
```

In [26]:
```
mu = 0.01
mu_sigma = 0.001
n_iter = 200
#centers = random_centers(X, K=2)
centers = np.array([[3., 8.], [5., 4.]])
sigma = np.random.uniform(low=0.5, high=1.5, size=2)
print('Initialized center = \n', centers)
print('Initializerd standard deviation = \n', sigma)

for n in range(n_iter):
    for i in range(len(X)):
        x = X[i]
        c = data_cluster(x, centers, sigma)
        centers[c] = centers[c] + mu*((x - centers[c])/(sigma[c]**2))
        c = data_cluster(x, centers, sigma)
        cov = (x - centers[c]).dot((x - centers[c]))
        sigma[c] = sigma[c] + mu_sigma*((-1/sigma[c]) + cov*(0.5/(sigma[c]**3)))
print('Final center = \n', centers)
print('Final standard deviation = \n', sigma)
out = np.array([data_cluster(x, centers, sigma) for x in X]) # output class for each datapoint
print('Output class for each data point: \n', out)
```

```
Initialized center =
 [[ 3.  8.]
 [ 5.  4.]]
Initializerd standard deviation =
 [ 1.49390432  0.65359494]
Final center =
 [[ 1.82060197  7.91788002]
 [ 5.90642541  2.23049265]]
Final standard deviation =
 [ 2.2394823   1.96478332]
Output class for each data point:
 [1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 0 0 0
 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
```

```
1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

This is quite interesting in the case of $\Sigma_k = \sigma_k^2 I$ that we get standard deviation around 2 for the Guassian generated data that has standard deviation of 2 for both axis. And standard deviation around 2.5 for generated data that has SD of 3, 2 for x, y axis respectively.

Here we miss classify 5 data points (see printed output)

```
In [27]: fig = plt.figure(1)
         plt.axis([-5,15,-5,15])
         ax = fig.add_subplot(1,1,1)
         colors = ['b', 'r']
         circles = [plt.Circle((c[0], c[1]), s, color=co, fill=False)
                    for (c, s, co) in zip(centers, sigma, colors)]
         for circle in circles:
             ax.add_patch(circle)
         for (center, co) in zip(centers, colors):
             plt.scatter(center[0], center[1], color=co)
         plt.scatter(X[(y==1).flatten(), 0], X[(y==1).flatten(), 1], color='r', alpha=0.3)
         plt.scatter(X[(y==0).flatten(), 0], X[(y==0).flatten(), 1], color='b', alpha=0.3)
         plt.xlabel('X')
         plt.ylabel('Y')
         plt.title('Final cluster center and standard deviation')
         plt.show()
```