STA 414 Assignment 2

Yubo Wang 1002138377

For question 1, 2, 3

clip\_train\_labels = np.argmax(train\_labels[:10000], axis=1)  
clip\_train\_images = train\_images[:10000]  
clip\_bi\_train\_images = 1.0 \*(train\_images[:10000] > 0.5)  
clip\_test\_labels = np.argmax(test\_labels[:10000], axis=1)  
clip\_test\_images = test\_images[:10000]  
clip\_bi\_test\_images = 1.0 \*(test\_images[:10000] > 0.5)

N\_data, train\_images, train\_labels, test\_images, test\_labels = load\_mnist()



MLE for :

, we have

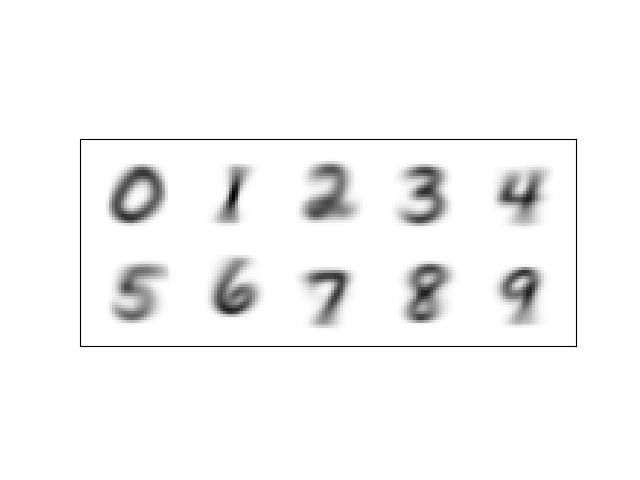


MAP for :

=



def q1c(train\_images, train\_labels):  
  
 theta = np.zeros((10, 784))  
 train\_labels = np.argmax(train\_labels, axis = 1)  
 count\_label =np.zeros(10)  
 for i in range(train\_labels.shape[0]):  
 theta[train\_labels[i]] += train\_images[i]  
 count\_label[train\_labels[i]] += 1  
 for i in range(10):  
 theta[i] = (theta[i]+1) / (count\_label[i] + 2)  
 save\_images(theta, 'theta1.png')  
 return theta  
  
  
theta = q1c(train\_images, train\_labels)







Since

is expanded in the problem,



def p\_x\_given\_c\_theta(theta, x, c):  
 ones = np.ones((1, 784))  
 return theta[c, :] \*\* x.reshape(1, 784) \* (1-theta[c, :]) \*\* np.subtract(ones, x.reshape(1, 784))  
  
  
def p\_xc\_given\_theta\_pi(theta, x, c):  
 result = np.prod(p\_x\_given\_c\_theta(theta, x, c))  
 result /= 10  
 return result  
  
  
def p\_c\_given\_x(theta, data, c):  
 num = p\_xc\_given\_theta\_pi(theta, data, c)  
 denom = 0  
 for i in range(10):  
 denom += p\_xc\_given\_theta\_pi(theta, data, i)  
 return num / denom  
  
  
def average\_log\_llh(theta, data, labels):  
 result = 0  
 for i in range(10000):  
 if p\_c\_given\_x(theta, data[i], labels[i]) == 0:  
 print("error")  
 log = math.log(p\_c\_given\_x(theta, data[i], labels[i]))  
 result += log  
 return result / 10000  
  
  
def accuracy(theta, images, labels):  
 prediction = []  
 for i in range(10000):  
 computed\_p = []  
 for c in range(10):  
 p = p\_c\_given\_x(theta, images[i], c)  
 # print(p)  
 computed\_p.append(p)  
 prediction.append(np.argmax(computed\_p))  
 # print(prediction)  
 labels = np.array(labels)  
 prediction = np.array(prediction)  
 correct = prediction[prediction==labels]  
 correct\_count = correct.shape[0]  
 return correct\_count / len(labels)  
def q1e(theta, train\_images, train\_labels, test\_images, test\_labels):  
 print("Training set accuracy: {}".format(accuracy(theta, train\_images, train\_labels)))  
 print("Test set accuracy: {}".format(accuracy(theta, test\_images, test\_labels)))  
 return average\_log\_llh(theta, train\_images, train\_labels)  
  
print(q1e(theta, clip\_train\_images, clip\_train\_labels, clip\_test\_images, clip\_test\_labels))

Training set accuracy: 0.8366

Test set accuracy: 0.8445

Training average log likelihood: -3.142

Test average log likelihood: -2.977



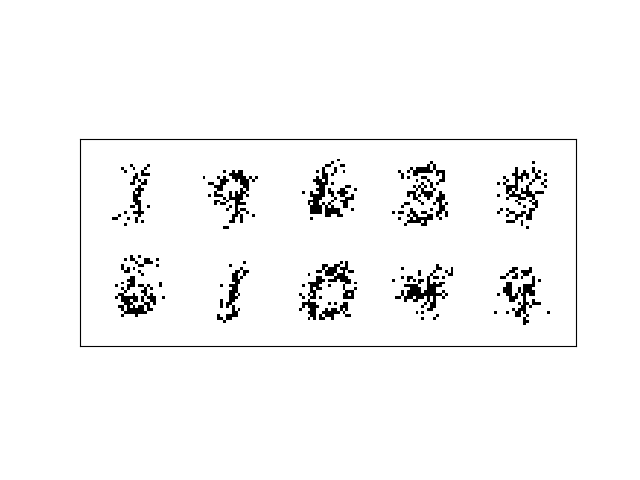
True.



False.



def q2c(theta):  
 image = np.zeros((10, 784))  
 for iter in range(1, 11):  
 index = np.random.choice(np.arange(10), 1, p=[0.1] \* 10)  
 print(index)  
 sample = []  
 for i in range(784):  
 prob = [theta[index, i][0], 1 - theta[index, i][0]]  
 print(prob)  
 s = np.random.choice(np.array([1, 0]), 1, p=prob)  
 sample.append(s)  
 sample = np.array(sample).reshape((1, 784))  
 image[iter-1,:] = sample  
 save\_images(image, 'random\_sample.png'))  
  
q2c(theta)



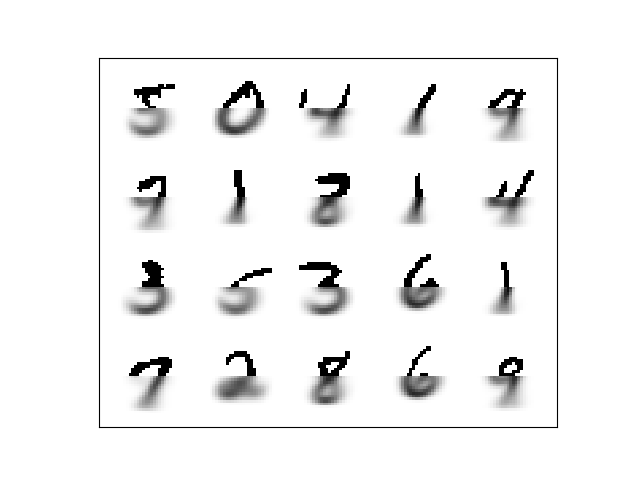








def q2e(x, theta):  
 x\_top = x[:,:392]  
 theta\_top = theta[:, :392]  
 joint\_top = np.exp(np.dot(x\_top, np.log(theta\_top.T)) + np.dot(1-x\_top, np.log(1-theta\_top.T)))[:20]  
 full\_new\_theta = []  
 for i in range(20):  
 new\_theta = []  
 for j in range(392, 784):  
 joint\_bot = np.exp(x[i,j] \* np.log(theta[:, j]) + (1-x[i,j]) \* (1-np.log(1-theta[:,j])))  
 t1 = np.dot(theta[:, j], joint\_top[i] \* joint\_bot)  
 t2 = np.dot(1 - theta[:, j], joint\_top[i] \* joint\_bot)  
 new\_theta.append(t1/(t1+t2))  
 full\_new\_theta.append(new\_theta)  
 result = np.zeros((20, 784))  
 for i in range(20):  
 result[i][:392] = x[i][:392]  
 result[i][392:] = full\_new\_theta[i]  
 save\_images(result, "q2e.png")  
  
q2e(clip\_bi\_train\_images, theta)








Number of parameter is 10\*784 = 7840



To compute , we have two cases:

Case 1:

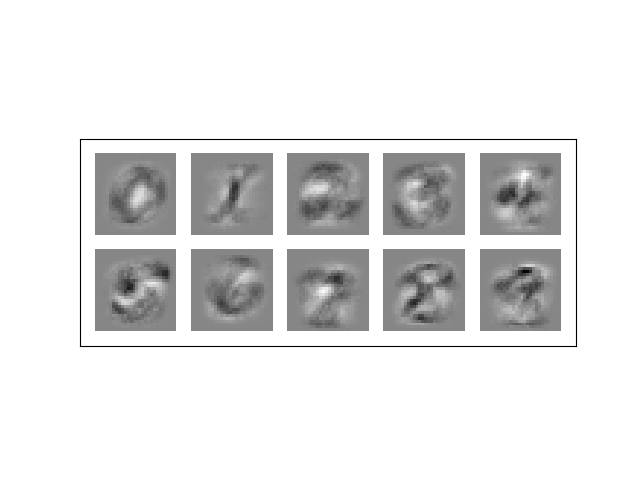
Case 2:

By using the one of k encoding for label

We have matrix form for code:



def compute\_prob(x, w):  
 return softmax(np.dot(w, x))  
  
  
def gd(x, c, w):  
 result = np.zeros((10, 784))  
 result[c, ] = x  
 x\_tile = np.tile(x, (10, 1))  
 result -= np.dot(np.diag(compute\_prob(x, w)), x\_tile)  
 return result  
  
def lr(x, y):  
 w = np.zeros((10, 784))  
 ratio = x.shape[0] // 10000  
 for i in range(50):  
 for j in range(ratio):  
 t1, t2 = j \* 10000, (j+1) \* 10000  
 x\_b = x[t1:t2]  
 y\_b = y[t1:t2]  
 dw = sum([gd(x\_b[k], y\_b[k], w) for k in range(10000)])  
 w += 0.01 \* dw  
 save\_images(w, "q3c.png")  
 return w  
  
w = lr(clip\_bi\_train\_images, clip\_train\_labels)







Training accuracy: 0.87

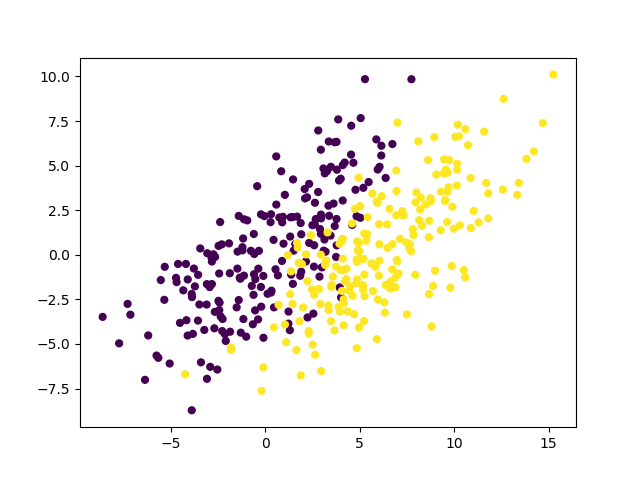
Training average log likelihood: -1.41

Test accuracy: 0.86

Test average log likelihood: -1.69



mu1 = np.array([0.1, 0.1])  
mu2 = np.array([6.0, 0.1])  
va = [10, 10]  
cova = 7  
VCVmatrix = np.array([[va[0], cova],  
 [cova, va[1]]])  
mvn1 = np.random.multivariate\_normal(mu1, VCVmatrix, size=200)  
mvn2 = np.random.multivariate\_normal(mu2, VCVmatrix, size=200)  
x1, y1 = mvn1.T  
x2, y2 = mvn2.T  
xc = np.concatenate((x1, x2))  
yc = np.concatenate((y1, y2))  
full\_data = np.concatenate((mvn1, mvn2))  
true\_label = []  
for i in range(200):  
 true\_label.append(1)  
for i in range(200):  
 true\_label.append(2)  
fig = plt.figure()  
plt.scatter(xc, yc, 24, c=true\_label)  
# plt.show()  
fig.savefig("true-cluster.png")

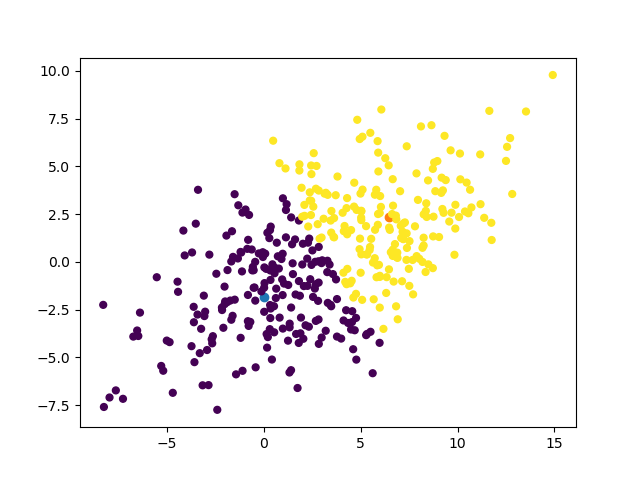
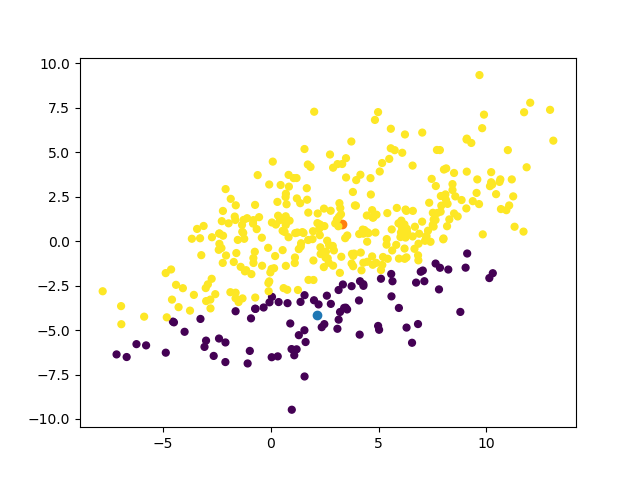






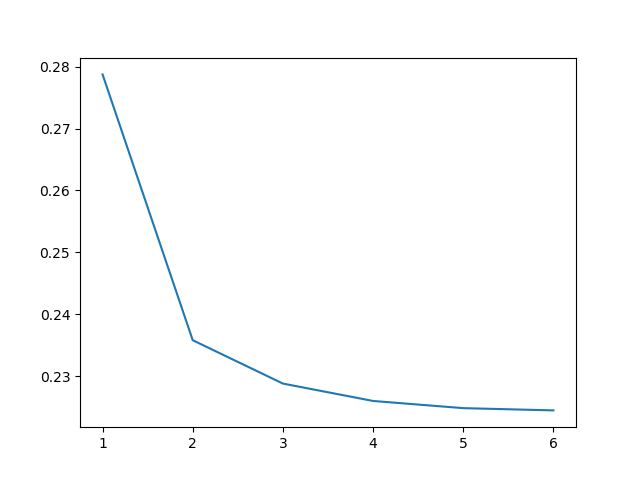
class K\_Means:  
 def \_\_init\_\_(self, k=2, tol=0.0001, max\_iter=50):  
 self.k = k  
 self.tol = tol  
 self.max\_iter = max\_iter  
  
 def cost(self, dp):  
 return [np.linalg.norm(dp-self.centroids[c]) for c in range(2)]  
  
 def km\_e\_step(self, data):  
 self.classifications = {}  
 cost = 0  
 for i in range(self.k):  
 self.classifications[i] = []  
 for d in data:  
 cost\_lst = self.cost(d)  
 min\_idx = cost\_lst.index(min(cost\_lst))  
 self.classifications[min\_idx].append((d[0], d[1]))  
 cost += sum(cost\_lst)  
 return cost  
  
 def km\_m\_step(self):  
 for c in range(len(self.classifications)):  
 self.centroids[c] = np.average(self.classifications[c], axis=0)  
  
 def fit(self,data):  
  
 # initialize centroids randomly  
 self.centroids = {}  
 idx\_lst = []  
 for i in range(self.k):  
 idx\_lst.append(random.randint(0, 400))  
 for i in range(self.k):  
 self.centroids[i] = data[idx\_lst[i]]  
 print("Initial centroids: " + str(self.centroids))  
 optimized = False  
 i = 0  
 log\_lst, iter\_lst = [], []  
 while i < self.max\_iter and not optimized:  
 i += 1  
 iter\_lst.append(i)  
 cost = self.km\_e\_step(data)  
 log\_lst.append(cost)  
 prev\_centroids = dict(self.centroids)  
 self.km\_m\_step()  
  
 for c in range(self.k):  
 original\_centroid = prev\_centroids[c]  
 current\_centroid = self.centroids[c]  
 print(original\_centroid, current\_centroid)  
 diff = np.sum((current\_centroid-original\_centroid)/original\_centroid\*100.0)  
 print("Iteration: {}. Shift: {}".format(i, diff))  
 if abs(diff) < self.tol:  
 optimized = True  
 print("finished")  
 fig = plt.figure()  
 for cent in range(2):  
 plt.scatter(self.centroids[cent][0], self.centroids[cent][1])  
 label = [0] \* len(self.classifications[0]) + [1] \* len(self.classifications[1])  
 full\_cluster = np.concatenate((self.classifications[0], self.classifications[1]))  
 x\_pos = full\_cluster[:, 0]  
 y\_pos = full\_cluster[:, 1]  
 plt.scatter(x\_pos, y\_pos, 24, c=label)  
 fig.savefig("k\_means\_iteration{}.png".format(i))  
 wrong1, wrong2 = 0, 0  
 for i in data[:200]:  
 if (i[0], i[1]) not in self.classifications[0]:  
 wrong1 += 1  
 if (i[0], i[1]) not in self.classifications[1]:  
 wrong2 += 1  
 fig = plt.figure()  
 plt.plot(iter\_lst, log\_lst)  
 fig.savefig("cost\_iter.png")  
 print("Error percentage: " + str(min(wrong1/200, wrong2 / 200)))

Iteration 1 Iteration 14



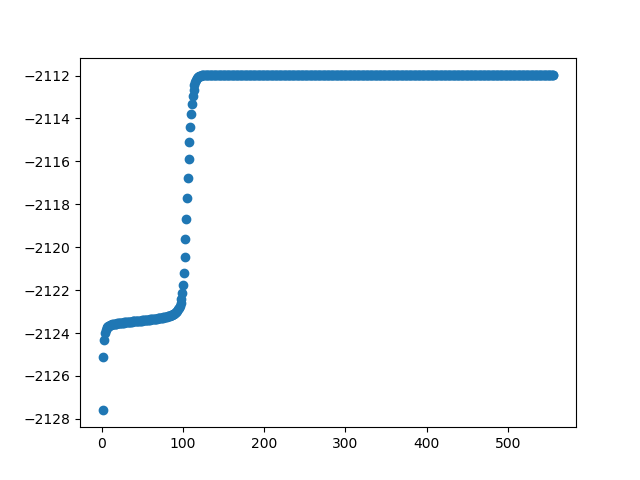
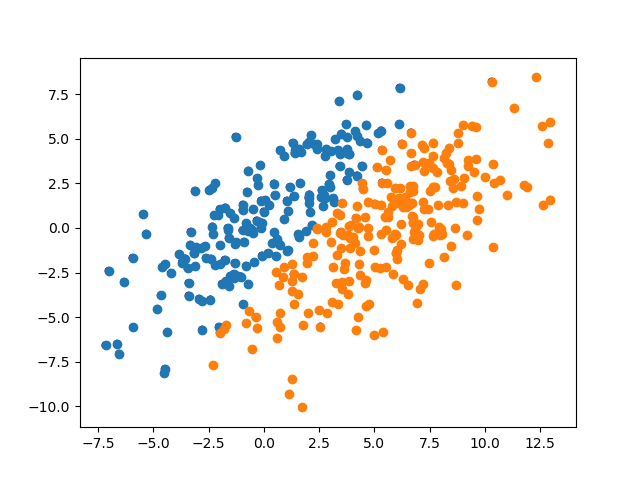
Error percentage: 0.235

Cost-iteration graph



class EM:  
 def \_\_init\_\_(self, k=2, tol = 0.00001, max\_iter=50):  
 self.imat = np.array([1, 0, 0, 1]).reshape(2,2)  
 self.mu1 = np.array([0, 0])  
 self.mu2 = np.array([1,1])  
 self.k = k  
 self.tol = tol  
 self.max\_iter = max\_iter  
 self.params = {'mu1': self.mu1, 'mu2': self.mu2, 'cov1' : self.imat, 'cov2' : self.imat, 'ratio' : [0.5, 0.5]}  
 self.pi = {0:0, 1:0}  
  
 def normal\_density(self, d, mu, cov):  
 return multivariate\_normal.pdf(d, mu, cov)  
  
 def log\_likelihood(self, d, mu, cov, ratio):  
 r = ratio  
 for j in range(len(d)):  
 r \*= self.normal\_density(d[j], mu[j], cov[j, j])  
 return r  
  
 def em\_e\_step(self, data):  
 self.classifications = {0:[], 1:[]}  
 xc = data[:, 0]  
 yc = data[:, 1]  
 label = {}  
 for i in range(data.shape[0]):  
 p1 = self.log\_likelihood([xc[i], yc[i]], self.params['mu1'], self.params['cov1'], self.params['ratio'][0])  
 p2 = self.log\_likelihood([xc[i], yc[i]], self.params['mu2'], self.params['cov2'], self.params['ratio'][1])  
 self.pi[0] += p1  
 self.pi[1] += p2  
 if p1 > p2:  
 self.classifications[0].append((xc[i], yc[i]))  
 else:  
 self.classifications[1].append((xc[i], yc[i]))  
  
 def em\_f\_step(self):  
 points\_in\_cluster\_1, points\_in\_cluster\_2 = np.array(self.classifications[0]), np.array(self.classifications[1])  
 cluster\_1\_ratio = self.pi[0] / 200  
 cluster\_2\_ratio = self.pi[1] / 200  
 print(cluster\_1\_ratio, cluster\_2\_ratio)  
 self.params['mu1'] = np.array([points\_in\_cluster\_1[:, 0].mean(), points\_in\_cluster\_1[:, 1].mean()])  
 self.params['mu2'] = np.array([points\_in\_cluster\_2[:, 0].mean(), points\_in\_cluster\_2[:, 1].mean()])  
 self.params['cov1'] = np.array([[points\_in\_cluster\_1[:, 0].std(), 0], [0, points\_in\_cluster\_1[:, 1].std()]])  
 self.params['cov2'] = np.array([[points\_in\_cluster\_2[:, 0].std(), 0], [0, points\_in\_cluster\_2[:, 1].std()]])  
 self.params['ratio'] = np.array([cluster\_1\_ratio, cluster\_2\_ratio])  
  
 def compute\_shift(self, cur\_params, old\_params):  
 result = 0  
 for p in ['mu1', 'mu2']:  
 for i in range(2):  
 result += (cur\_params[p][i] - old\_params[p][i]) \*\* 2  
 return result \*\* 0.5  
  
 def fit(self, data):  
 optimized = False  
 iteration = 0  
 while iteration < self.max\_iter and not optimized:  
 iteration += 1  
 prev\_params = dict(self.params)  
 self.em\_e\_step(data)  
 self.em\_f\_step()  
 shift = self.compute\_shift(self.params, prev\_params)  
 print('Iteration: {}, Shift; {}'.format(iteration, shift))  
 if shift < self.tol:  
 optimized = True  
 fig = plt.figure()  
 label = [0] \* len(self.classifications[0]) + [1] \* len(self.classifications[1])  
 full\_cluster = np.concatenate((self.classifications[0], self.classifications[1]))  
 x\_pos = full\_cluster[:, 0]  
 y\_pos = full\_cluster[:, 1]  
 plt.scatter(x\_pos, y\_pos, 24, c=label)  
 fig.savefig("iteration{}.png".format(iteration))  
 wrong1, wrong2 = 0, 0  
 for i in data[:200]:  
 if (i[0], i[1]) not in self.classifications[0]:  
 wrong1 += 1  
 if (i[0], i[1]) not in self.classifications[1]:  
 wrong2 += 1  
 print("Error percentage: " + str(min(wrong1/200, wrong2 / 200)))  
  
  
em = EM()  
em.fit(full\_data)

Error percentage: 0.11250







By comparing the result from K-mean and EM, we can conclude that EM has better accuracy compared with K-means. K-means does not identify the real clusters pattern. (It divide two cluster vertically instead of horizontally).

For error percentage, EM has misclassification error of 11.2 and K-mean has 23.5.

In terms of running time, EM usually has more iterations then K-means.