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()
```

1.

a.

MLE for θ:

$\mathrm{L} = \prod_{i=1}^{10000} p(x_i, c|\theta, \pi) = \prod_{i=1}^{10000} p(c_i|\pi) \prod_{d=1}^{784} \theta_{cd}^{xd_i}(1-\theta_{cd})^{1-xd_i}$

By setting $\frac{\mathrm{dlog(L)}}{\mathrm{d}\theta_{cd}} = 0$, we have

$\widehat{\theta_{cd}} = \frac{\sum_{i=1}^{10000} I(c_i=k)x_{di}}{\sum_{i=1}^{10000} I(c_i=k)}$

b.

MAP for θ:

$\widehat{\theta_{MAP}} = \mathrm{argmax}\left(\prod_{i=1}^{10000} p(x_i, c|\theta, \pi)\right)$

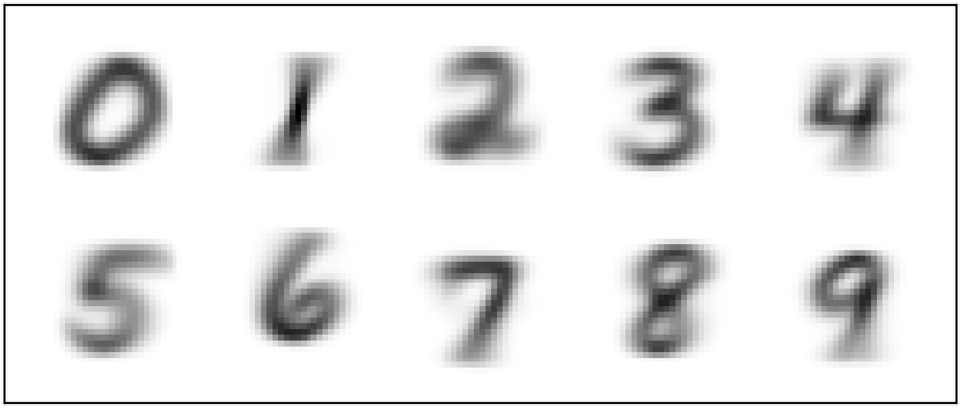
$\mathrm{M} = \prod_{i=1}^{10000} p(x_i, c|\theta, \pi)\ \theta_{cd}(1-\theta_{cd})$

$\frac{dM}{d\theta_{cd}} = \frac{1}{\theta_{cd}} - \frac{1}{1-\theta_{cd}} + \frac{dlog(L)}{d\theta_{cd}}$

$\widehat{\theta_{MAP}}_{cd} = \frac{1+\sum_{i=1}^{10000} I(c_i=k)x_{di}}{2+\sum_{i=1}^{10000} I(c_i=k)}$

c.

```
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)
```



d.

$$\log p(c|x,\theta,\pi) = \log(p(x,c|\theta,\pi) - \log(p(x|\theta,\pi)$$

Since $p(c|x,\theta,\pi) = \frac{p(x,c|\theta,\pi)}{p(x|\theta,\pi)}$

$p(x,c|\theta,\pi)$ is expanded in the problem, $p(x|\theta,\pi) = \sum_{i=0}^{9} p(x,c_i|\theta,\pi)$

e.

```python
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)
```

```python
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

2.

a.

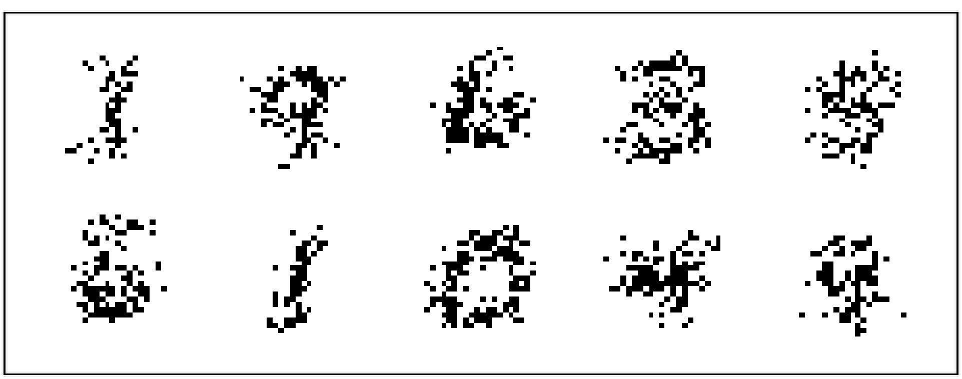
True.

b.

False.

c.

```python
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)
```

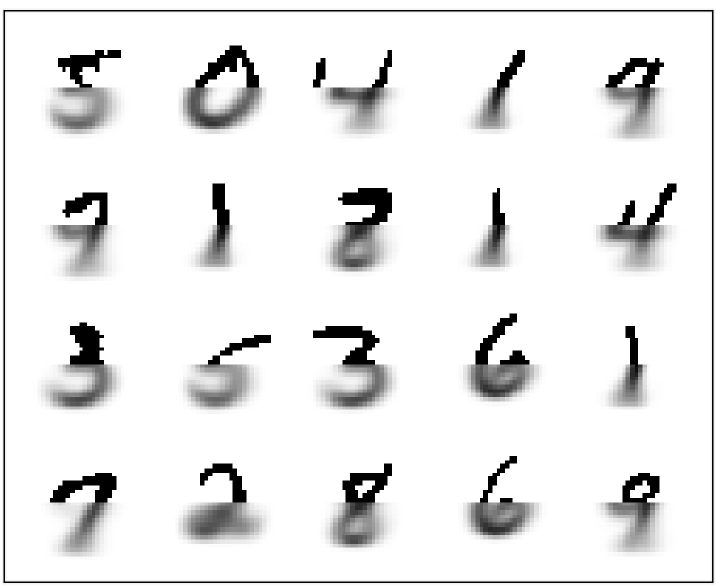


d.

$$p\left(x_i \in bottom \middle| x_{top}, \theta, \pi\right)$$
$$= \sum_{c=0}^{9} p\left(x_i \in bottom \middle| c, x_{top}, \theta, \pi\right) * p\left(c \middle| x_{top}, \theta, \pi\right)$$
$$= \sum_{c=0}^{9} p\left(x_i \in bottom \middle| c, x_{top}, \theta, \pi\right) * \frac{p\left(x_{top}|c,\theta,\pi\right)p(c|\theta,\pi)}{\sum_{c=0}^{9} p\left(x_{top}, c \middle| \theta, \pi\right)}$$
$$= \sum_{c=0}^{9}\left(\theta_{cd}^{xd}\left(1-\theta_{cd}\right)^{1-x_d}\right) \frac{\pi_C \prod_{d=1}^{392} \theta_{cd}^{xd}\left(1-\theta_{cd}\right)^{1-x_d}}{\sum_{c=0}^{9} \pi_C \prod_{d=1}^{392} \theta_{cd}^{xd}\left(1-\theta_{cd}\right)^{1-x_d}}$$

e.

```
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)
```



3.

a.

Number of parameter is 10*784 = 7840

b.

To compute $\nabla_{wc'} \log p(c|x,w)$, we have two cases:

Case 1: $c \neq c'$

$$\nabla_{wc'} \log p(c|x,w) = \frac{\sum_{i=0}^{9} \exp(w_i^T x)}{\exp(w_c^T x)} * \frac{0-\exp(w_c^T x)\exp(w_{c'}^T x)x}{\left(\sum_{i=0}^{9} \exp(w_i^T x)\right)^2}$$

$$= -\frac{\exp\left(w_{c'}^T x\right)x}{\sum_{i=0}^{9} \exp(w_i^T x)} = -p(c'|x,w)x$$

Case 2: $c = c'$

$$\nabla_{wc'} \log p(c|x,w) = \frac{\sum_{i=0}^{9} \exp(w_i^T x)}{\exp(w_c^T x)} * \frac{\exp(w_c^T x) * \left(\sum_{i=0}^{9} \exp(w_i^T x) - \exp(w_c^T x)\right)x}{\left(\sum_{i=0}^{9} \exp(w_i^T x)\right)^2}$$

$$= \frac{\left(\sum_{i=0}^{9} \exp(w_i^T x) - \exp(w_c^T x)\right)x}{\sum_{i=0}^{9} \exp(w_i^T x)} = \left(1 - p(c'|x,w)\right)x$$

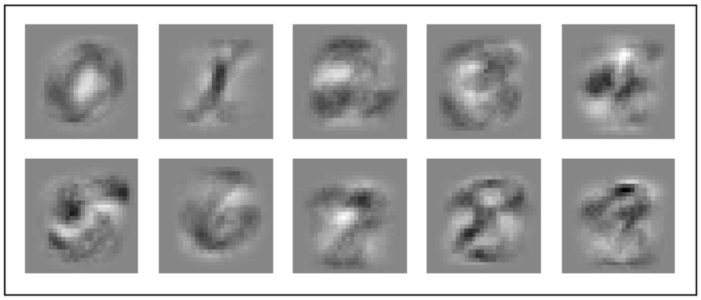
By using the one of k encoding for label

We have matrix form for code:

$$\left(label - \begin{bmatrix} p(0|x,w) \\ \dots \\ p(9|x,w) \end{bmatrix}\right) \times \begin{bmatrix} x \\ \dots \\ x \end{bmatrix}$$

c.

```
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)
```



d.

Training accuracy: 0.87

Training average log likelihood: -1.41

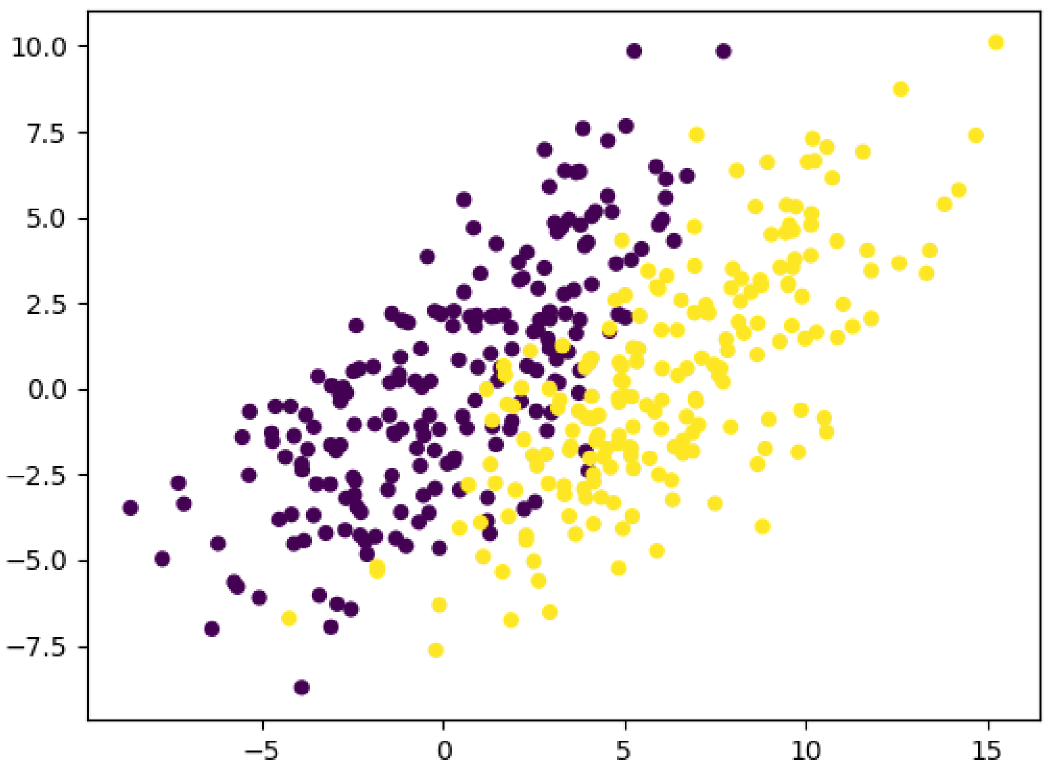
Test accuracy: 0.86

Test average log likelihood: -1.69

4.

a.

```
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")
```

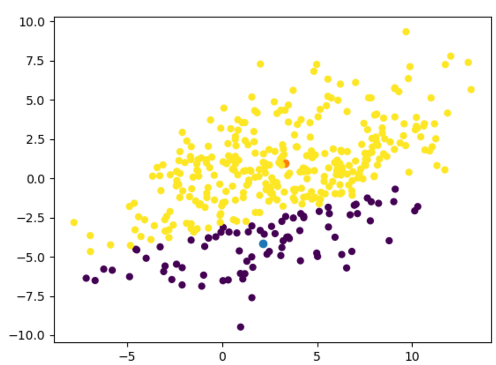


b.

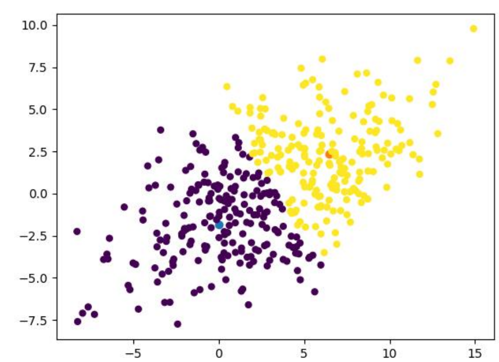
```
class K_Means:
    def __init__(self, k=2, tol=0.0001, max_iter=50):
        self.k = k
        self.tol = tol
        self.max_iter = max_iter

    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

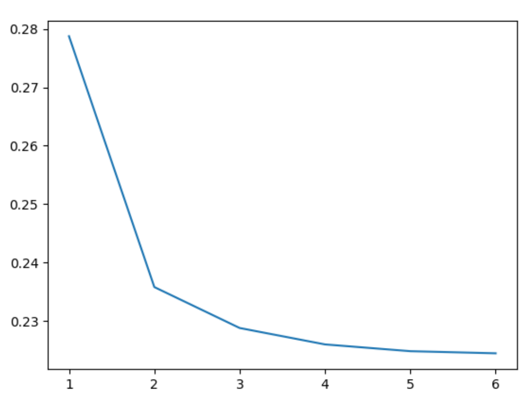


Iteration 14



Error percentage: 0.235

Cost-iteration graph

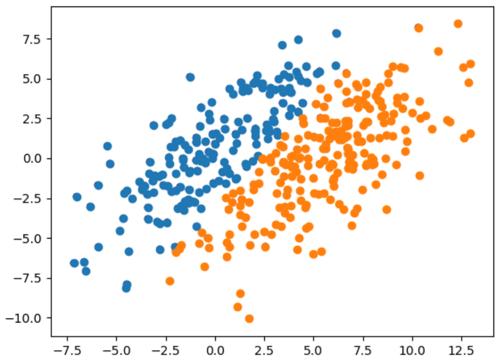


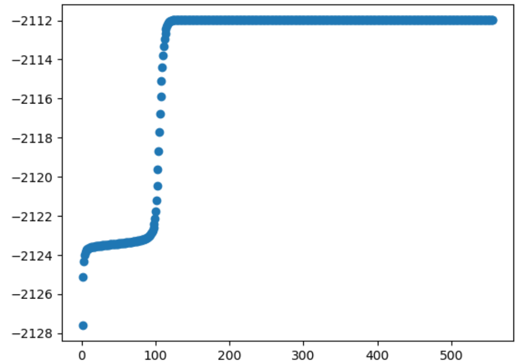
c.

```
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





d.

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.