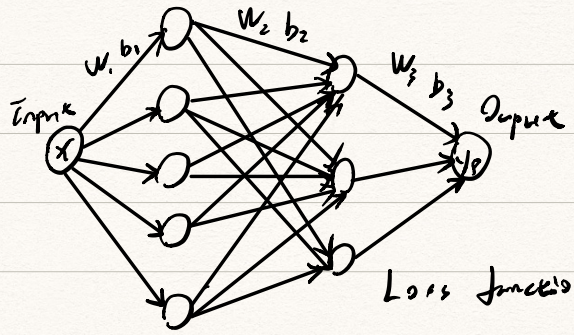
1. Backpropagation Algorithm for Neural Networks

(a)



Loss function: $\sum_{i=1}^{N} L(y_i, f(x_i | w, b))$

$= \sum_{i=1}^{N} (y_i - f(x_i | w, b))^2$

(b) $MSE(\hat{y}) = \frac{1}{2n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$

$$\frac{\partial MSE(\hat{y})}{\partial \hat{y}} = \frac{1}{2n} \times 2(y_i - \hat{y}_i) \times (-1)$$

$$= \frac{(\hat{y}_i - y_i)}{n}$$

The code is included in the appendix.

(c) Linear function

$$\sigma_{linear}(z) = z$$

$$\frac{\partial \sigma_{linear}(z)}{\partial z} = 1$$

ReLU function

$$\sigma_{ReLU}(z) = \begin{cases} 0 & z < 0 \end{cases}$$

$$| z \quad \text{otherwise}$$

$$\frac{\partial \sigma_{ReLU}(z)}{\partial z} = \begin{cases} 0 & z \le 0 \\ 1 & \text{otherwise} \end{cases}$$

tanh function

$$\sigma_{tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\frac{\partial \sigma_{tanh}(z)}{\partial z} = \frac{(e^z + e^{-z})^2 - (e^z - e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2}$$

$$= 1 - \sigma_{tanh}^2(z)$$

The code is included in the appendix.

(d) The code is included in the appendix

$$\frac{\partial MSE}{\partial y_p} \times \frac{\partial y_p}{\partial z_i} \times \frac{\partial z_i}{\partial a_i} \times \frac{\partial a_i}{\partial z_{i-1}} \times \frac{\partial z_{i-1}}{\partial a_{i-1}}$$

(e) With the same batch size, the training error decreases as the increase of epoch. This is obvious because the number of iterations match my expec increases. However, with the same epoch, the training error increases with the increase of bach size. This maybe due to as the increase of batch not match size, large batch tend to converge sharp minimizers. my expec

The performance of three activation function is: tanh > ReLU > linear

The code is included in appendix.

(f) With 2 different activation function : tanh, ReLU. The MSE both decrease first and then increase as the increase of network width. The best width is around 8. This matches my expectation. This is the trade off between width and MSE.

2. Regularized and Kernel k-means

(a) 0

(b)

$$\lambda \mu_i + \sum_{x_j \in C_i} 2(\mu_i - x_j) = 0.$$

$$\lambda \mu_i + |C_i| \mu_i - \sum_{x_j \in C_i} x_j = 0$$

$$\mu_i = \frac{1}{|C_i| + \lambda} \sum_{x_j \in C_i} x_j$$

(c)

$$\min_{C_1 C_2 \cdots C_k} \sum_{i=1}^{k} \left(\|\mu_i\|_2 + \sum_{x_j \in C_i} \|x_j - \mu_i\|_2 \right)$$

(d)

$$\text{Class}(j) = \text{argmin}_k \| x_j - \mu_k \|^2$$

$$= \text{argmin}_k \left\| x_j - \frac{\sum_{l \in S_k} x_l}{|S_k|} \right\|^2$$

$$= \text{argmin}_k \; k(x_j, x_j) - \frac{2 \sum_{l \in S_k} k(x_j, x_l)}{|S_k|} + \frac{\sum_{l \in S_k} \sum_{l' \in S_k} k(x_l, x_{l'})}{|S_k|^2}$$

(e) I see there are multiple $k(x_1, x_2)$ through out the algorithm. In order to save

time, all kernels should not be calculated twice or more. So the idea is to precompute $K(x_i, x_j)$ for all possible combinations of i and j and store them for use in the algorithm. Where $i, j \in \{1, 2, \ldots, n\}$ and i can be equal to j.

3. Expectation Maximization (EM) Algorithm: in action!

(a) The initial guesses for 3 different algorithms are very different because the distance between different clusters are small.

After looking at the results of 3 algorithms. The EM performs the best, next is Kmeans, the worse is KNN

(b) After the increase of factor from 1 to 10, the distance between different clusters are much larger (The clusters are much obvious). Now the initial guesses are much closer to the truth. And the results show that the performances of the 3 algorithms are very close. They all perform very well.

4. One dimensional Mixture of Two Gaussians

(a) $$p_Z(z) = \begin{cases} \beta & z = 1 \\ 1-\beta & z = 2 \end{cases}$$

$$(X \mid Z = k) \sim N(\mu_k, \sigma_k^2)$$

$$\therefore \quad P_\theta(X=x_i, Z_i=1)$$
$$= P_\theta(X=x_i \mid Z_i=1)\, P_\theta(Z_i=1)$$
$$= \frac{1}{\sqrt{2\pi}\sigma_1} e^{-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}} \times \beta = \frac{\beta}{\sqrt{2\pi}\sigma_1} e^{-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}}$$

$$P_\theta(X=x_i, Z_i=2)$$
$$= P_\theta(X=x_i \mid Z_i=2)\, P_\theta(Z_i=2)$$
$$= \frac{1}{\sqrt{2\pi}\sigma_2} e^{-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}} \times (1-\beta) = \frac{1-\beta}{\sqrt{2\pi}\sigma_2} e^{-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}}$$

$$\therefore \quad P_\theta(X=x_i) = \frac{\beta}{\sqrt{2\pi}\sigma_1} e^{-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}} + \frac{1-\beta}{\sqrt{2\pi}\sigma_2} e^{-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}}$$

(b) $$\ell_\theta(x) = \sum_i \log \sum_{k=1}^{2} N(x_i \mid \mu_k, \sigma_k)\, P(z=k)$$

$$= \sum_i \log\left(\frac{\beta}{\sqrt{2\pi}\sigma_1} e^{-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}} + \frac{1-\beta}{\sqrt{2\pi}\sigma_2} e^{-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}} \right)$$

It is obvious that this log likelihood function is very hard to take derivative. So it is very hard to set the derivative to 0 and get the closed form solution of MLE.

(c) $$\ell_\theta(x_i) = \log\left(\frac{\beta}{\sqrt{2\pi}\sigma_1} e^{-\frac{(x_i-\mu_1)^2}{2\sigma_1^2}} + \frac{1-\beta}{\sqrt{2\pi}\sigma_2} e^{-\frac{(x_i-\mu_2)^2}{2\sigma_2^2}} \right)$$

According to Jensen's Inequality

$$\log P(x_i \mid \theta) = \log \sum P(x_i, z_i \mid \theta)$$

$$l_\theta(x_i) = \log p(x_i|\theta) = \log \sum_z p(x_i, z_i|\theta)$$

$$= \log \sum_z \frac{q(z_i|x_i,\theta)\, p(x_i, z_i|\theta)}{q(z_i|x_i,\theta)}$$

$$= \log E_q\left(\frac{p(x_i, z_i|\theta)}{q(z_i|x_i,\theta)}\right) \quad \text{(log is concave function)}$$

$$\therefore \geq E_q\left[\log\left(\frac{p(x_i, z_i|\theta)}{q(z_i|x_i,\theta)}\right)\right]$$

$$\text{when } q = p_\theta \Rightarrow l_\theta(x_i) \geq E_q\left[\log \frac{p_\theta(x = x_i, z = k)}{q_\theta(z_i = k | x = x_i)}\right]$$

(d) lower bound: $E_q\left[\log \frac{p_\theta(x = x_i, z_i = k)}{q_\theta(z_i = k | x = x_i)}\right]$

$$= -\sum_z q_\theta(z_i = k | x = x_i) \log\left[q_\theta(z_i = k | x = x_i)\right]$$

$$+ \sum_z q_\theta(z_i = k | x = x_i) \log\left[p_\theta(x = x_i, z_i = k)\right]$$

$$= H\left(q_\theta(z_i = k | x = x_i)\right) + E_q\left[L_c(x_i, z_i|\theta)\right]$$

$$\equiv F_i(q, \theta)$$

$\therefore$ In E-step: lower bound: $F_i(q, \theta^t)$

In M-step: lower bound: $F_i(q^{t+1}, \theta)$

$\therefore$ In E-step: $q^{t+1} = \arg\max_q F(q, \theta^t)$

In M-step: $\theta^{t+1} = \arg\max_\theta F(q^{t+1}, \theta^t)$

$\therefore$ Alternating between E-step and and M-step until

both try to maximize $F(q, \theta)$

(e) $p(z_i = k \mid x_i, \theta^t) = \frac{p(z_i, x_i \mid \theta^t)}{p(x_i \mid \theta^t)}$

$= \frac{p(z_i, x_i \mid \theta^t)}{\sum_{z_i} p(x_i, z_i \mid \theta^t)} = \frac{p(x_i \mid z_i, \theta^t)\, p(z_i \mid \theta^t)}{\sum_{z_i} p(x_i \mid z_i, \theta^t)\, p(z_i \mid \theta^t)}$

$= \frac{\alpha_k^t \, \mathcal{N}(x_i \mid \mu_k^t, \sigma_k^t)}{\sum_k \alpha_k^t \, \mathcal{N}(x_i \mid \mu_k^t, \sigma_k^t)}$

$\therefore q_{i,1}^{t+1} = \frac{\frac{\beta^t}{\sqrt{2\pi}\sigma_1^t} e^{-\frac{(x_i - \mu_1^t)^2}{2\sigma_1^{t2}}}}{\frac{\beta^t}{\sqrt{2\pi}\sigma_1^t} e^{-\frac{(x_i - \mu_1^t)^2}{2\sigma_1^{t2}}} + \frac{1-\beta^t}{\sqrt{2\pi}\sigma_2^t} e^{-\frac{(x_i - \mu_2^t)^2}{2\sigma_2^{t2}}}}$

$q_{i,2}^{t+1} = \frac{\frac{1-\beta^t}{\sqrt{2\pi}\sigma_2^t} e^{-\frac{(x_i - \mu_2^t)^2}{2\sigma_2^{t2}}}}{\frac{\beta^t}{\sqrt{2\pi}\sigma_1^t} e^{-\frac{(x_i - \mu_1^t)^2}{2\sigma_1^{t2}}} + \frac{1-\beta^t}{\sqrt{2\pi}\sigma_2^t} e^{-\frac{(x_i - \mu_2^t)^2}{2\sigma_2^{t2}}}}$

(f) $\mu_1^{t+1} = \frac{\sum_i q_{i,1}^{t+1} x_i}{\sum_i q_{i,1}^{t+1}}$

(g) MOG fits very well with EM. But k-means and soft k-means does not fit well with EM.

- k-means gives more flexibility than k-means.

MoG allows non-spherical clusters And different scales of covariance.

MoG makes explicit assumptions in the form of statistical distributions thus easier to generalize.

1.

(b)

```
# Cost function used to compute prediction errors
class QuadraticCost(object):

    # Compute the squared error between the prediction yp and the observation y
    # This method should compute the cost per element such that the output is the
    # same shape as y and yp
    @staticmethod
    def fx(y,yp):
        # TODO: PART B #########################################################################
        return (np.square(y-yp)) / 2
        # PART B ###############################################################################

    # Derivative of the cost function with respect to yp
    @staticmethod
    def dx(y,yp):
        # TODO: PART B #########################################################################
        return yp - y
        # PART B ###############################################################################
```

(c)

```
# Linear activation
class LinearActivation(object):
    @staticmethod
    def fx(z):
        # TODO: PART C #########################################################################
        return z
        # PART C ###############################################################################

    @staticmethod
    def dx(z):
        # TODO: PART C #########################################################################
        return np.ones_like(z)
        # PART C ###############################################################################
```

```
# Rectified linear unit
class ReLUActivation(object):
    @staticmethod
    def fx(z):
        # TODO: PART C #########################################################################
        return z * (z > 0)
        # PART C ###############################################################################

    @staticmethod
    def dx(z):
        # TODO: PART C #########################################################################
        return 1.0 * (z > 0)
        # PART C ###############################################################################
```

```
# Hyperbolic tangent function
class TanhActivation(object):

    # Compute tanh for each element in the input z
    @staticmethod
    def fx(z):
        # TODO: PART C #########################################################################
        return (np.exp(z) - np.exp((-1)*z)) / (np.exp(z) + np.exp((-1)*z))
        # PART C ###############################################################################

    # Compute the derivative of the tanh function with respect to z
    @staticmethod
    def dx(z):
        # TODO: PART C #########################################################################
        return 1 - np.square(np.tanh(z))
        # PART C ###############################################################################
```

(d)

```
def compute_grad(self, x, y):
    # Feed forward, computing outputs of each layer and
    # intermediate outputs before the non-linearities
    yp, a, z = self.evaluate(x)

    # d is inialized here to be (dMSE / dyp)
    d = self.cost.dx(y.T, yp)
    grad = []

    # Backpropogate the error
    for layer, curZ in zip(reversed(self.layers), reversed(z)):
        # TODO: PART D ###########################################################
        # grad[i] should correspond with the gradient of the output of layer i before the
        # activation is applied (dMSE / dz_i); be sure values are stored in the correct
        # ordering!
        grad_i = np.multiply(d,layer.dx(curZ))
        grad = [grad_i] + grad
        d = layer.W.T.dot(grad_i)
        # PART D #################################################################
    return grad, a
```

(e)

```
def update(self, layers, g, a):
    m = a[0].shape[1]
    for layer, curGrad, curA in zip(layers, g, a):
        # TODO: PART E ###########################################################
        dw = self.eta / m * curGrad.dot(curA.T)
        db = self.eta * np.mean(curGrad)
        layer.updateWeights(dw)
        layer.updateBias(db)
        # PART E #################################################################
```

The screenshot of the output is shown on the next page.

```
Using SGD
ReLU
Batch size:  10 Epoch:  10 Train Error:  0.038940902163643945 Test Error:  0.04268340529430519
ReLU
Batch size:  10 Epoch:  20 Train Error:  0.03398623730871783 Test Error:  0.03901132256200904
ReLU
Batch size:  10 Epoch:  40 Train Error:  0.025295588106085103 Test Error:  0.032432730623558816
ReLU
Batch size:  50 Epoch:  10 Train Error:  0.04961180226095961 Test Error:  0.05089793635992317
ReLU
Batch size:  50 Epoch:  20 Train Error:  0.03616977205659915 Test Error:  0.038395452096749695
ReLU
Batch size:  50 Epoch:  40 Train Error:  0.03172142656662911 Test Error:  0.03689713204732637
ReLU
Batch size:  100 Epoch:  10 Train Error:  0.057722400026334386 Test Error:  0.059188938064516805
ReLU
Batch size:  100 Epoch:  20 Train Error:  0.051104045879460214 Test Error:  0.052136885403051274
ReLU
Batch size:  100 Epoch:  40 Train Error:  0.04615838828381433 Test Error:  0.048414862174646286
ReLU
Batch size:  200 Epoch:  10 Train Error:  0.07280394011924103 Test Error:  0.06748819617915305
ReLU
Batch size:  200 Epoch:  20 Train Error:  0.05343513678231537 Test Error:  0.05617441692849356
ReLU
Batch size:  200 Epoch:  40 Train Error:  0.051160752714599717 Test Error:  0.049774555851365754
linear
Batch size:  10 Epoch:  10 Train Error:  0.0727351601129363 Test Error:  0.07317435738647052
linear
Batch size:  10 Epoch:  20 Train Error:  0.07236665939514171 Test Error:  0.07086851335170909
linear
Batch size:  10 Epoch:  40 Train Error:  0.07000222890973087 Test Error:  0.07025739101842948
linear
Batch size:  50 Epoch:  10 Train Error:  0.08173160235681685 Test Error:  0.07859452034026554
linear
Batch size:  50 Epoch:  20 Train Error:  0.07713969813491513 Test Error:  0.07772281412470534
linear
Batch size:  50 Epoch:  40 Train Error:  0.07189231215702374 Test Error:  0.06997570643465631
linear
Batch size:  100 Epoch:  10 Train Error:  0.11341860299151152 Test Error:  0.10718976601251433
linear
Batch size:  100 Epoch:  20 Train Error:  0.08197563747409743 Test Error:  0.08293042707799692
linear
Batch size:  100 Epoch:  40 Train Error:  0.07521767884249866 Test Error:  0.0752114043512611
linear
Batch size:  200 Epoch:  10 Train Error:  0.11345466592783111 Test Error:  0.10736382416598486
linear
Batch size:  200 Epoch:  20 Train Error:  0.09794456196988321 Test Error:  0.09776371586734292
linear
Batch size:  200 Epoch:  40 Train Error:  0.08544199497860958 Test Error:  0.08444151428651339
tanh
Batch size:  10 Epoch:  10 Train Error:  0.020242870920021982 Test Error:  0.027970702580415768
tanh
Batch size:  10 Epoch:  20 Train Error:  0.016514599524955147 Test Error:  0.025819751969557304
tanh
Batch size:  10 Epoch:  40 Train Error:  0.014805041289188896 Test Error:  0.024058836727491525
tanh
Batch size:  50 Epoch:  10 Train Error:  0.028154151619177184 Test Error:  0.031755971256969745
tanh
Batch size:  50 Epoch:  20 Train Error:  0.02300981151470193 Test Error:  0.027831070175586108
tanh
Batch size:  50 Epoch:  40 Train Error:  0.01806434898885613 Test Error:  0.02543065814347185
tanh
Batch size:  100 Epoch:  10 Train Error:  0.036326582883758146 Test Error:  0.03735277852964145
tanh
Batch size:  100 Epoch:  20 Train Error:  0.03002334647779314 Test Error:  0.03365817116995559
tanh
Batch size:  100 Epoch:  40 Train Error:  0.023746500808689078 Test Error:  0.027363133818666934
tanh
Batch size:  200 Epoch:  10 Train Error:  0.056203822076940614 Test Error:  0.05516429406088538
tanh
Batch size:  200 Epoch:  20 Train Error:  0.04118625149338277 Test Error:  0.037987091413543325
tanh
Batch size:  200 Epoch:  40 Train Error:  0.03340792330228364 Test Error:  0.03441898946107204
```

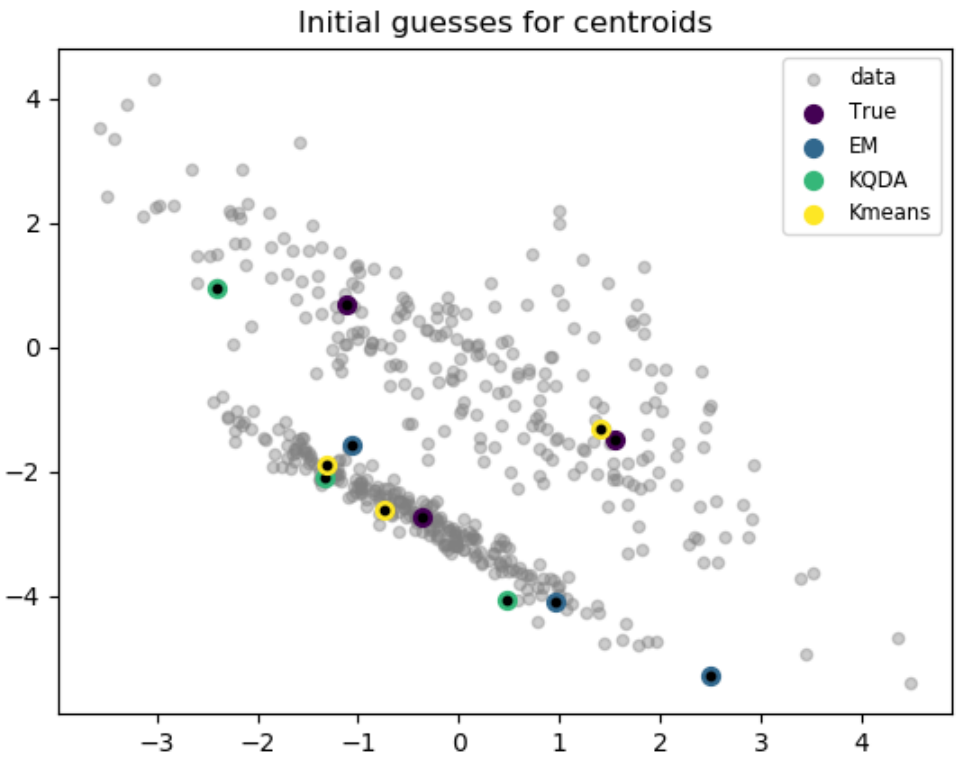
(f)

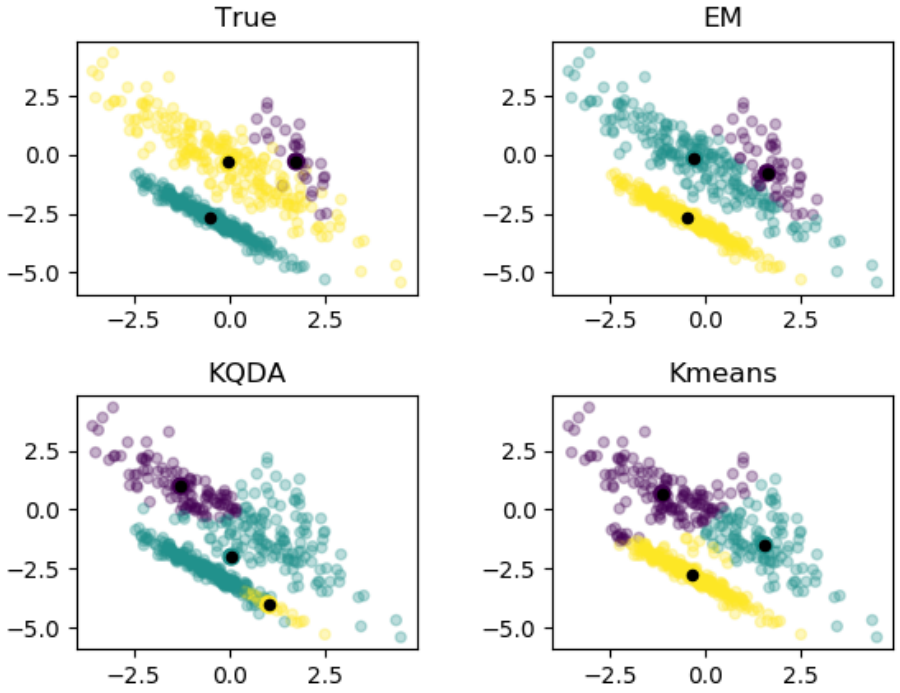
Training error result.

```
Training with various sized network
ReLU neural nework width(2) train MSE: 0.11956186960504603
ReLU neural network width(2) test MSE: 0.10702909157901459
ReLU neural nework width(4) train MSE: 0.10537989698393437
ReLU neural network width(4) test MSE: 0.09559344186755417
ReLU neural nework width(8) train MSE: 0.07962746814255602
ReLU neural network width(8) test MSE: 0.07697495981416637
ReLU neural nework width(16) train MSE: 0.08033642500894943
ReLU neural network width(16) test MSE: 0.07530008704515469
ReLU neural nework width(32) train MSE: 0.08352998775544901
ReLU neural network width(32) test MSE: 0.07774239248716214
tanh neural nework width(2) train MSE: 0.06673495490624651
tanh neural network width(2) test MSE: 0.06964296423493636
tanh neural nework width(4) train MSE: 0.060168656693197466
tanh neural network width(4) test MSE: 0.056303872396740805
tanh neural nework width(8) train MSE: 0.05635002689846051
tanh neural network width(8) test MSE: 0.05702097818340173
tanh neural nework width(16) train MSE: 0.0646145889425098
tanh neural network width(16) test MSE: 0.05850410216609744
tanh neural nework width(32) train MSE: 0.06773170948693948
tanh neural network width(32) test MSE: 0.06402980813858601
```

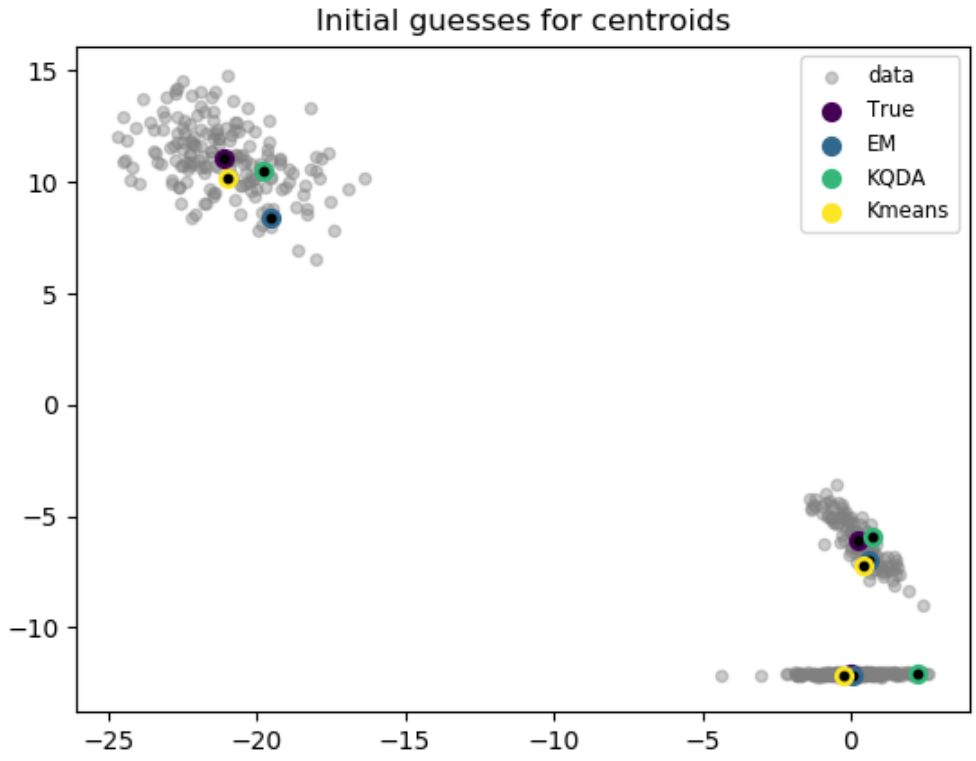
3.

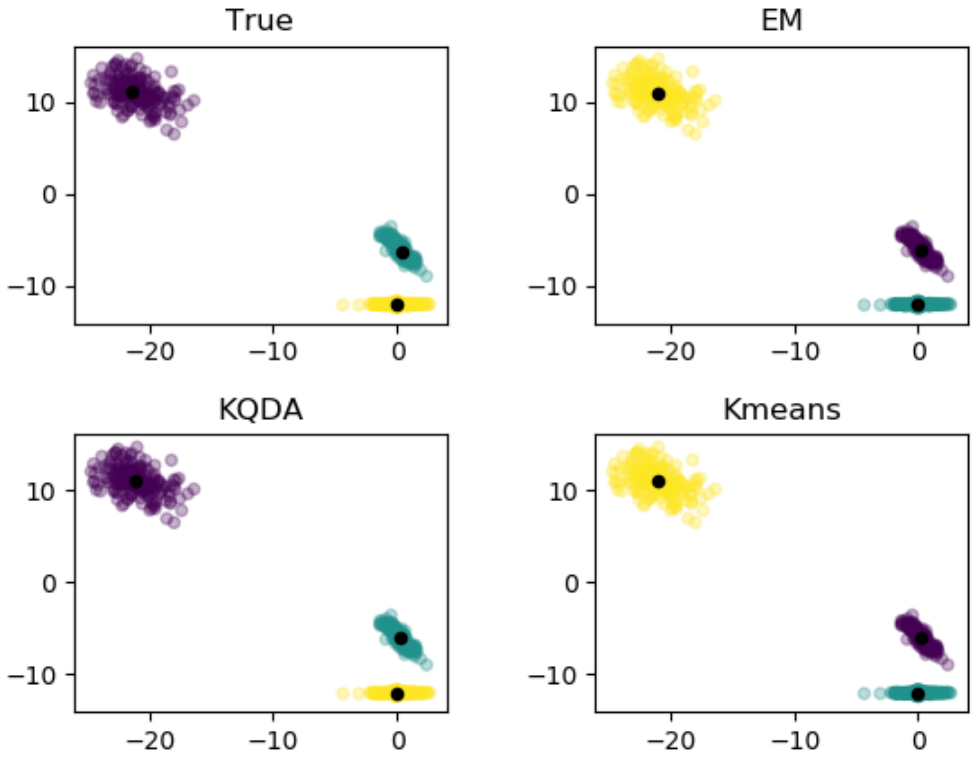
(a)




True
EM
KQDA
Kmeans
2.5
0.0
−2.5
−5.0
−2.5
0.0
2.5

(b)


Initial guesses for centroids
data
True
EM
KQDA
Kmeans
15
10
5
0
−5
−10
−25
−20
−15
−10
−5
0


True
EM
KQDA
Kmeans
10
0
−10
−20
−10
0

```
import matplotlib
import matplotlib.pyplot as plt
import mnist
import numpy as np

# Gradient descent optimization
# The learning rate is specified by eta
class GDOptimizer(object):
    def __init__(self, eta):
        self.eta = eta

    def initialize(self, layers):
        pass

    # This function performs one gradient descent step
    # layers is a list of dense layers in the network
    # g is a list of gradients going into each layer before the nonlinear
     activation
    # a is a list of outputs from previous layer
    def update(self, layers, g, a):
        m = a[0].shape[1]
        for layer, curGrad, curA in zip(layers, g, a):
            # TODO: PART E
             ###################################################################
            dw = self.eta / m * curGrad.dot(curA.T)
            db = self.eta * np.mean(curGrad)
            layer.updateWeights(dw)
            layer.updateBias(db)
            # PART E
             ###################################################################
             ######


# Cost function used to compute prediction errors
class QuadraticCost(object):

    # Compute the squared error between the prediction yp and the
     observation y
    # This method should compute the cost per element such that the output
     is the
    # same shape as y and yp
    @staticmethod
    def fx(y,yp):
        # TODO: PART B
         ###################################################################
        return (np.square(y-yp)) / 2
        # PART B
         ###################################################################
         ######

    # Derivative of the cost function with respect to yp
    @staticmethod
    def dx(y,yp):
        # TODO: PART B
         ###################################################################
```

```
        return yp - y
        # PART B
         ###########################################################################
         ######

# Sigmoid function fully implemented as an example
class SigmoidActivation(object):
    @staticmethod
    def fx(z):
        # PART C Example
         ###########################################################################
        return 1 / (1 + np.exp(-z))
        # PART C
         ###########################################################################
         ######

    @staticmethod
    def dx(z):
        # PART C Example
         ###########################################################################
        return SigmoidActivation.fx(z) * (1 - SigmoidActivation.fx(z))
        # PART C
         ###########################################################################
         ######


# Hyperbolic tangent function
class TanhActivation(object):

    # Compute tanh for each element in the input z
    @staticmethod
    def fx(z):
        # TODO: PART C
         ###########################################################################
        return (np.exp(z) - np.exp((-1)*z)) / (np.exp(z) + np.exp((-1)*z))
        # PART C
         ###########################################################################
         ######

    # Compute the derivative of the tanh function with respect to z
    @staticmethod
    def dx(z):
        # TODO: PART C
         ###########################################################################
        return 1 - np.square(np.tanh(z))
        # PART C
         ###########################################################################
         ######


# Rectified linear unit
class ReLUActivation(object):
    @staticmethod
    def fx(z):
```

```
        # TODO: PART C
        ###########################################################################
        return z * (z > 0)
        # PART C
        ###########################################################################
        ######

    @staticmethod
    def dx(z):
        # TODO: PART C
        ###########################################################################
        return 1.0 * (z > 0)
        # PART C
        ###########################################################################
        ######


# Linear activation
class LinearActivation(object):
    @staticmethod
    def fx(z):
        # TODO: PART C
        ###########################################################################
        return z
        # PART C
        ###########################################################################
        ######

    @staticmethod
    def dx(z):
        # TODO: PART C
        ###########################################################################
        return np.ones_like(z)
        # PART C
        ###########################################################################
        ######


# This class represents a single hidden or output layer in the neural
 network
class DenseLayer(object):

    # numNodes: number of hidden units in the layer
    # activation: the activation function to use in this layer
    def __init__(self, numNodes, activation):
        self.numNodes = numNodes
        self.activation = activation

    def getNumNodes(self):
        return self.numNodes

    # Initialize the weight matrix of this layer based on the size of the
     matrix W
    def initialize(self, fanIn, scale=1.0):
        s = scale * np.sqrt(6.0 / (self.numNodes + fanIn))
```

```
        self.W = np.random.normal(0, s, (self.numNodes,fanIn))
        self.b = np.random.uniform(-1, 1, (self.numNodes,1))

    # Apply the activation function of the layer on the input z
    def a(self, z):
        return self.activation.fx(z)

    # Compute the linear part of the layer
    # The input a is an n x k matrix where n is the number of samples
    # and k is the dimension of the previous layer (or the input to the
     network)
    def z(self, a):
        return self.W.dot(a) + self.b # Note, this is implemented where we
         assume a is k x n

    # Compute the derivative of the layer's activation function with
     respect to z
    # where z is the output of the above function.
    # This derivative does not contain the derivative of the matrix
     multiplication
    # in the layer.  That part is computed below in the model class.
    def dx(self, z):
        return self.activation.dx(z)

    # Update the weights of the layer by adding dW to the weights
    def updateWeights(self, dW):
        self.W = self.W - dW

    # Update the bias of the layer by adding db to the bias
    def updateBias(self, db):
        self.b = self.b - db


# This class handles stacking layers together to form the completed neural
 network
class Model(object):

    # inputSize: the dimension of the inputs that go into the network
    def __init__(self, inputSize):
        self.layers = []
        self.inputSize = inputSize

    # Add a layer to the end of the network
    def addLayer(self, layer):
        self.layers.append(layer)

    # Get the output size of the layer at the given index
    def getLayerSize(self, index):
        if index >= len(self.layers):
            return self.layers[-1].getNumNodes()
        elif index < 0:
            return self.inputSize
        else:
            return self.layers[index].getNumNodes()
```

```
# Initialize the weights of all of the layers in the network and set
 the cost
# function to use for optimization
def initialize(self, cost, initializeLayers=True):
    self.cost = cost
    if initializeLayers:
        for i in range(0,len(self.layers)):
            if i == len(self.layers) - 1:
                self.layers[i].initialize(self.getLayerSize(i-1))
            else:
                self.layers[i].initialize(self.getLayerSize(i-1))

# Compute the output of the network given some input a
# The matrix a has shape n x k where n is the number of samples and
# k is the dimension
# This function returns
# yp - the output of the network
# a - a list of inputs for each layer of the newtork where
#     a[i] is the input to layer i
#     (note this does not include the network output!)
# z - a list of values for each layer after evaluating layer.z(a) but
#     before evaluating the nonlinear function for the layer
def evaluate(self, x):
    curA = x.T
    a = [curA]
    z = []
    for layer in self.layers:
        z.append(layer.z(curA))
        curA = layer.a(z[-1])
        a.append(curA)
    yp = a.pop()
    return yp, a, z

# Compute the output of the network given some input a
# The matrix a has shape n x k where n is the number of samples and
# k is the dimension
def predict(self, a):
    a, _, _ = self.evaluate(a)
    return a.T

# Computes the gradients at each layer. y is the true labels, yp is the
# predicted labels, and z is a list of the intermediate values in each
# layer. Returns the gradients and the forward pass outputs (per layer).
#
# In particular, we return a list of dMSE/dz_i. The reasoning behind
 this is that
# in the update function for the optimizer, we do not give it the z
 values
# we compute from evaluating the network.
def compute_grad(self, x, y):
    # Feed forward, computing outputs of each layer and
    # intermediate outputs before the non-linearities
    yp, a, z = self.evaluate(x)

    # d is inialized here to be (dMSE / dyp)
```

```
        d = self.cost.dx(y.T, yp)
        grad = []

        # Backpropogate the error
        for layer, curZ in zip(reversed(self.layers), reversed(z)):
            # TODO: PART D
             ##################################################################
             ####
            # grad[i] should correspond with the gradient of the output of
             layer i before the
            # activation is applied (dMSE / dz_i); be sure values are
             stored in the correct
            # ordering!
            grad_i = np.multiply(d,layer.dx(curZ))
            grad = [grad_i] + grad
            d = layer.W.T.dot(grad_i)
            # PART D
             ##################################################################
             ##########
        return grad, a

    # Train the network given the inputs x and the corresponding
     observations y
    # The network should be trained for numEpochs iterations using the
     supplied
    # optimizer
    def train(self, x, y, numEpochs, optimizer):

        # Initialize some stuff
        n = x.shape[0]
        x = x.copy()
        y = y.copy()
        hist = []
        optimizer.initialize(self.layers)

        # Run for the specified number of epochs
        for epoch in range(0,numEpochs):

            # Compute the gradients
            grad, a = self.compute_grad(x, y)

            # Update the network weights
            optimizer.update(self.layers, grad, a)

            # Compute the error at the end of the epoch
            yh = self.predict(x)
            C = self.cost.fx(y, yh)
            C = np.mean(C)
            hist.append(C)
        return hist

    def trainBatch(self, x, y, batchSize, numEpochs, optimizer):
        # Copy the data so that we don't affect the original one when
         shuffling
        x = x.copy()
```

```
        y = y.copy()
        hist = []
        n = x.shape[0]

        for epoch in np.arange(0,numEpochs):

            # Shuffle the data
            r = np.arange(0,x.shape[0])
            x = x[r,:]
            y = y[r,:]
            e = []

            # Split the data in chunks and run SGD
            for i in range(0,n,batchSize):
                end = min(i+batchSize,n)
                batchX = x[i:end,:]
                batchY = y[i:end,:]
                e += self.train(batchX, batchY, 1, optimizer)
            hist.append(np.mean(e))

        return hist


if __name__ == '__main__':
    N0 = 3
    N1 = 9

    # Switch these statements to True to run the code for the corresponding
     parts
    # Part E
    SGD = False
    # Part F
    DIFF_SIZES = True

    # Generate the training set
    np.random.seed(9001)
    y_train = mnist.train_labels()
    y_test = mnist.test_labels()
    X_train = (mnist.train_images()/255.0)
    X_test = (mnist.test_images()/255.0)
    train_idxs = np.logical_or(y_train == N0, y_train == N1)
    test_idxs = np.logical_or(y_test == N0, y_test == N1)
    y_train = y_train[train_idxs].astype('int')
    y_test = y_test[test_idxs].astype('int')
    X_train = X_train[train_idxs]
    X_test = X_test[test_idxs]
    y_train = (y_train == N0).astype('int')
    y_test = (y_test == N0).astype('int')
    y_train *= 2
    y_test *= 2
    y_train -= 1
    y_test -= 1
    X_train = X_train.reshape(X_train.shape[0], -1)
    X_test = X_test.reshape(X_test.shape[0], -1)
    y_test = y_test[:, np.newaxis]
```

```
y = y_train[:, np.newaxis]
x = X_train
xLin=np.linspace(-np.pi,np.pi,250).reshape((-1,1))



yHats = {}

activations = dict(ReLU=ReLUActivation,
                   tanh=TanhActivation,
                   linear=LinearActivation)
lr = dict(ReLU=0.02,tanh=0.02,linear=0.005)
names = ['ReLU','linear','tanh']

#### PART E ####
if SGD:
    print('\n----------------------------------------\n')
    print('Using SGD')
    for key in names[0:3]:
        for batchSize in [10, 50, 100, 200]:
            for epoch in [10, 20, 40]:
                activation = activations[key]
                model = Model(x.shape[1])
                model.addLayer(DenseLayer(4,activation()))
                model.addLayer(DenseLayer(1,LinearActivation()))
                model.initialize(QuadraticCost())
                hist = model.trainBatch(x, y, batchSize, epoch,
                 GDOptimizer(eta=lr[key]))
                y_hat_train = model.predict(x)
                y_pred_train = np.sign(y_hat_train)
                y_hat_test = model.predict(X_test)
                y_pred_test = np.sign(y_hat_test)
                error_train = np.mean(np.square(y_hat_train - y))/2
                error_test = np.mean(np.square(y_hat_test - y_test))/2
                print(key)
                print('Batch size: ', batchSize ,'Epoch: ', epoch,
                 'Train Error: ', error_train, 'Test Error: ',
                 error_test)


#### PART F ####
# Train with different sized networks
if DIFF_SIZES:
    print('\n----------------------------------------\n')
    print('Training with various sized network')
    names = ['ReLU', 'tanh']
    widths = [2, 4, 8, 16, 32]
    errors = {}
    for key in names:
        error = []
        for width in widths:
            activation = activations[key]
            model = Model(x.shape[1])
            model.addLayer(DenseLayer(width,activation()))
            model.addLayer(DenseLayer(1,LinearActivation()))
```

```
model.initialize(QuadraticCost())
epochs = 256
hist =
 model.trainBatch(x,y,x.shape[0],epochs,GDOptimizer(eta=lr
 [key]))

y_hat_train = model.predict(x)
y_hat_test = model.predict(X_test)

error_train = np.mean(np.square(y_hat_train - y))/2
error_test = np.mean(np.square(y_hat_test - y_test))/2

print(key+' neural nework width('+str(width)+') train MSE:
 '+str(error_train))
print(key+' neural network width('+str(width)+') test MSE:
 '+str(error_test))
```

```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
import os

#This function generate random mean and covariance
def gauss_params_gen(num_clusters, num_dims, factor):
    mu = np.random.randn(num_clusters,num_dims)*factor
    sigma = np.random.randn(num_clusters,num_dims,num_dims)
    for k in range(num_clusters):
        sigma[k] = np.dot(sigma[k],sigma[k].T)

    return (mu, sigma)

#Given mean and covariance generate data
def data_gen(mu, sigma, num_clusters, num_samples):
    labels = []
    X = []
    cluster_prob = np.array([np.random.rand() for k in
     range(num_clusters)])
    cluster_num_samples = (num_samples * cluster_prob /
     sum(cluster_prob)).astype(int)
    cluster_num_samples[-1] = num_samples-sum(cluster_num_samples[:-1])

    for k, ks in enumerate(cluster_num_samples):
        labels.append([k]*ks)
        X.append(np.random.multivariate_normal(mu[k], sigma[k], ks))

    # shuffle data
    randomize = np.arange(num_samples)
    np.random.shuffle(randomize)
    X =  np.vstack(X)[randomize]
    labels =  np.array(sum(labels,[]))[randomize]

    return X, labels


def data2D_plot(ax, x, labels, centers, cmap, title):
    data = {'x0': x[:,0], 'x1': x[:,1], 'label': labels}
    ax.scatter(data['x0'], data['x1'], c=data['label'], cmap=cmap, s=20,
     alpha=0.3)
    ax.scatter(centers[:, 0], centers[:, 1],
     c=np.arange(np.shape(centers)[0]), cmap=cmap, s=50, alpha=1)
    ax.scatter(centers[:, 0], centers[:, 1], c='black', cmap=cmap, s=20,
     alpha=1)
    ax.title.set_text(title)

def plot_init_means(x, mus, algs, fname):
    import matplotlib.cm as cm
    fig = plt.figure()
    plt.scatter(x[:,0], x[:,1], c='gray', cmap='viridis', s=20, alpha= 0.4,
     label='data')
    for mu, alg, clr in zip(mus, algs, cm.viridis(np.linspace(0, 1,
     len(mus)))):
        plt.scatter(mu[:,0], mu[:, 1], c=clr, s=50, label=alg)
```

```
        plt.scatter(mu[:, 0], mu[:, 1], c='black', s=10, alpha=1)
    legend = plt.legend(loc='upper right', fontsize='small')
    plt.title('Initial guesses for centroids')
    fig.savefig(fname)

def loss_plot(loss, title, xlabel, ylabel, fname):
    fig = plt.figure(figsize = (13, 6))
    plt.plot(np.array(loss))
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    fig.savefig(fname)


def gaussian_pdf (X, mu, sigma):
    # Gaussian probability density function
    return np.linalg.det(sigma) ** -.5 ** (2 * np.pi) ** (-X.shape[1]/2.) \
                    * np.exp(-.5 * np.einsum('ij, ij -> i',\
                    X - mu, np.dot(np.linalg.inv(sigma) , (X - mu).T).T ) )

def EM_initial_guess (num_dims, data, num_samples, num_clusters):
    # randomly choose the starting centroids/means
    # as num_clusters of the points from datasets
    mu = data[np.random.choice(num_samples, num_clusters, False), :]

    # initialize the covariance matrice for each gaussian
    sigma = [np.eye(num_dims)] * num_clusters

    # initialize the probabilities/weights for each gaussian
    # begin with equal weight for each gaussian
    alpha = [1./num_clusters] * num_clusters

    return mu, sigma, alpha

def EM_E_step (num_clusters, num_samples, data, mu, sigma, alpha):
    ## Vectorized implementation of e-step equation to calculate the
    ## membership for each of k -gaussians
    Q = np.zeros((num_samples, num_clusters))
    for k in range(num_clusters):
        Q[:, k] = alpha[k] * gaussian_pdf(data, mu[k], sigma[k])

    ## Normalize so that the responsibility matrix is row stochastic
    Q = (Q.T / np.sum(Q, axis = 1)).T

    return Q

def EM_M_step (num_clusters, num_dims, num_samples, Q, data):

    # M Step
    ## calculate the new mean and covariance for each gaussian by
    ## utilizing the new responsibilities
    mu      = np.zeros((num_clusters, num_dims))
    sigma   = np.zeros((num_clusters, num_dims, num_dims))
    alpha = np.zeros(num_clusters)
```

```
    ## The number of datapoints belonging to each gaussian
    num_samples_per_cluster = np.sum(Q, axis = 0)

    for k in range(num_clusters):
        ## means
        mu[k] = 1. / num_samples_per_cluster[k] * np.sum(Q[:, k] * data.T,
         axis = 1).T
        centered_data = np.matrix(data - mu[k])

        ## covariances
        sigma[k] = np.array(1. / num_samples_per_cluster[k] *
         np.dot(np.multiply(centered_data.T,  Q[:, k]), centered_data))

        ## and finally the probabilities
        alpha[k] = 1. / (num_clusters*num_samples) *
         num_samples_per_cluster[k]

    return mu, sigma, alpha

def EM_log_likelihood_calc (num_clusters, num_samples, data, mu, sigma,
 alpha):
    L = np.zeros((num_samples, num_clusters))
    for k in range(num_clusters):
        L[:, k] = alpha[k] * gaussian_pdf(data, mu[k], sigma[k])
    return np.sum(np.log(np.sum(L, axis = 1)))


def EM_calc (num_dims, num_samples, num_clusters, x):
    log_likelihoods = []
    labels          = []
    iter_cnt        = 0
    epsilon         = 0.0001
    max_iters       = 200
    update          = 2*epsilon

    # initial guess
    mu, sigma, alpha = EM_initial_guess(num_dims, x, num_samples,
     num_clusters)
    mus = [mu]
    sigmas = [sigma]
    alphas = [alpha]
    while (update > epsilon) and (iter_cnt < max_iters):
        iter_cnt += 1

        # E - Step
        Q = EM_E_step (num_clusters, num_samples, x, mu, sigma, alpha)

        # M - Step
        mu, sigma, alpha = EM_M_step (num_clusters, num_dims, num_samples,
         Q, x)

        mus.append(mu)
        sigmas.append(sigma)
        alphas.append(alpha)
```

```
        # Likelihood computation
        log_likelihoods.append(EM_log_likelihood_calc(num_clusters,
         num_samples, x, mu, sigma, alpha))

        # check convergence
        if iter_cnt >= 2 :
            update = np.abs(log_likelihoods[-1] - log_likelihoods[-2])

        # logging
        print("iteration {}, update {}".format(iter_cnt, update))

        # print current iteration
        labels.append(np.argmax(Q, axis = 1))

    return labels, log_likelihoods, {'mu': mus, 'sigma': sigmas, 'alpha':
     alphas}

def kmeans_initial_guess (data, num_samples, num_clusters):
    # randomly choose the starting centroids/means
    # as num_clusters of the points from datasets
    mu = data[np.random.choice(num_samples, num_clusters, False), :]
    return mu

def kmeans_get_labels(num_clusters, num_samples, num_dims, data, mu):
    # set all dataset points to the best cluster according to minimal
     distance
    #from centroid of each cluster
    dist = np.zeros((num_clusters, num_samples))
    for k in range(num_clusters):
        dist[k] = np.linalg.norm(data - mu[k], axis=1)

    labels = np.argmin(dist, axis=0)

    return labels

def kmeans_get_means(num_clusters, num_dims, data, labels):
    # Compute the new means given the reclustering of the data
    mu = np.zeros((num_clusters, num_dims))
    for k in range(num_clusters):
        idx_list = np.where(labels == k)[0]
        if (len(idx_list) == 0):
            r = np.random.randint(len(data))
            mu[k] = data[r,:]
        else:
            mu[k] = np.mean(data[idx_list], axis=0)
    return mu

def kmeans_calc_loss(num_clusters, num_samples, data, mu, labels):
    dist = np.zeros((num_samples, num_clusters))
    for j in range(num_samples):
        for k in range(num_clusters):
            if (labels[j] == k) :
                dist[j,k] = np.linalg.norm(data[j] - mu[k])
    return sum(sum(dist))
```

```
def k_means_calc (num_dims, num_samples, num_clusters, x):
    loss            = []
    labels          = []
    iter_cnt        = 0
    epsilon         = 0.00001
    max_iters       = 100
    update          = 2*epsilon

    # initial guess
    mu = [kmeans_initial_guess(x, num_samples, num_clusters)]

    while (update > epsilon) and (iter_cnt < max_iters):
        iter_cnt += 1
        # Assign labels to each datapoint based on centroid
        labels.append(kmeans_get_labels(num_clusters, num_samples,
         num_dims, x, mu[-1]))

        # Assign centroid based on labels
        mu.append(kmeans_get_means(num_clusters, num_dims, x, labels[-1]))
        # check convergence
        if iter_cnt >= 2 :
            update = np.linalg.norm(mu[-1] - mu[-2], None)

        # Print distance to centroids vs iteration
        loss.append(kmeans_calc_loss(num_clusters, num_samples, x, mu[-1],
         labels[-1]))

        # logging
        print("iteration {}, update {}".format(iter_cnt, update))


    return labels, loss, mu

def k_qda_initial_guess (num_dims, data, num_samples, num_clusters):
    # randomly choose the starting centroids/means
    # as num_clusters of the points from datasets
    mu = data[np.random.choice(num_samples, num_clusters, False), :]

    # initialize the covariance matrice for each gaussian
    sigma = [np.eye(num_dims)] * num_clusters

    return mu, sigma

def k_qda_get_parms(num_clusters, num_dims, data, labels):
    ## calculate the new mean and covariance for each gaussian
    mu      = np.zeros((num_clusters, num_dims))
    sigma   = np.zeros((num_clusters, num_dims, num_dims))

    for k in range(num_clusters):
        c_k = labels==k
        if (len(data[c_k]) == 0):
            r = np.random.randint(len(data))
            mu[k] = data[r,:]
        else:
```

```
            mu[k] = np.mean(data[c_k], axis=0)

        if (len(data[c_k]) > 1):
            centered_data = np.matrix(data[c_k] - mu[k])
            sigma[k] = np.array(1. / len(data[c_k]) *
             np.dot(centered_data.T, centered_data))
        else:
            sigma[k] = np.eye(num_dims)

    return mu, sigma

def k_qda_get_labels(num_clusters, num_samples, mu, sigma, data):
    # set all dataset points to the best cluster according to best
    # probability given calculated means and covariances
    dist = np.zeros((num_clusters, num_samples))
    for k in range(num_clusters):
        data_center = (data - mu[k])
        dist[k] = np.einsum('ij, ij -> i', data_center,
         np.dot(np.linalg.inv(sigma[k]) , data_center.T).T )
        labels = np.argmin(dist, axis=0)
    return labels


def k_qda_calc(num_dims, num_samples, num_clusters, x):
    loss            = []
    labels          = []
    iter_cnt        = 0
    epsilon         = 0.00001
    max_iters       = 100
    update          = 2*epsilon

    # initial guess
    mu, sigma = k_qda_initial_guess(num_dims, x, num_samples, num_clusters)
    mus = [mu]
    sigmas = [sigma]
    while (update > epsilon) and (iter_cnt < max_iters):
        iter_cnt += 1

       # Assign labels to each datapoint based on probability
        labels.append(k_qda_get_labels(num_clusters, num_samples, mus[-1],
         sigmas[-1], x))

       # Assign centroid and covarince based on labels
        mu, sigma = k_qda_get_parms(num_clusters, num_dims, x, labels[-1])

        mus.append(mu)
        sigmas.append(sigma)

       # check convergence
        if iter_cnt >= 2 :
            update = np.linalg.norm(mus[-1] - mus[-2], None)
            update += np.linalg.norm(sigmas[-1] - sigmas[-2], None)

       # logging
        print("iteration {}, update {}".format(iter_cnt, update))
```

```
    return labels, {'mu': mus, 'sigma': sigmas}


def experiments(seed, factor, dir='plots', num_samples=500, num_clusters=3):

    if not os.path.exists(dir):
        os.makedirs(dir)

    np.random.seed(seed)
    num_dims     = 2

    # generate data samples
    (mu, sigma) = gauss_params_gen(num_clusters, num_dims, factor)
    x, true_labels   = data_gen(mu, sigma, num_clusters, num_samples)
    #### Expectation-Maximization
    EM_labels, log_likelihoods, EM_parms = EM_calc (num_dims, num_samples,
     num_clusters, x)
    #### K QDA
    kqda_labels, kqda_parms = k_qda_calc(num_dims, num_samples,
     num_clusters, x)
    #### K means
    kmeans_labels, loss, kmean_mus = k_means_calc (num_dims, num_samples,
     num_clusters, x)

    #Collect all results
    labels = [true_labels, EM_labels[-1], kqda_labels[-1],
     kmeans_labels[-1]]
    mus_fin = np.array([mu, EM_parms['mu'][-1], kqda_parms['mu'][-1],
     kmean_mus[-1]])
    algs = np.array(['True', 'EM', 'KQDA', 'Kmeans'])

    #### Plot
    fig = plt.figure()
    fig.subplots_adjust(hspace=0.4, wspace=0.4)
    for i, (lbl,alg, mu) in enumerate(zip(labels, algs, mus_fin)):
        ax = fig.add_subplot(2, 2, i+1)
        data2D_plot(ax, x, lbl, mu, 'viridis', alg)

    fname = os.path.join(dir,
     'Results_s{}_f{}_n{}_k{}.png'.format(seed,factor,num_samples,
     num_clusters))
    fig.savefig(fname)

    mus_init = np.array([mu, EM_parms['mu'][0], kqda_parms['mu'][0],
     kmean_mus[0]])
    init_mu_fname = os.path.join(dir,
     'init_mu_s{}_f{}_n{}_k{}.png'.format(seed,factor, num_samples,
     num_clusters))
    plot_init_means(x, mus_init, algs, init_mu_fname)

if __name__ == "__main__":
    experiments(seed=63, factor=10, num_samples=500, num_clusters=3)
```