Morgan Rosenkranz EECE5644 5/2/21

Question 1:

I began by generating training data sets of 20, 200, and 2000 samples and a validation set of 10000 samples. To create the two classes I used the given means and covariances and generated a mixed Gaussian as specified by the problem and weights.

Part 1:

Using the given means and covariances I created a theoretically optimal classifier and applied it to the validation dataset. I then varied the threshold which gave the following ROC curve.

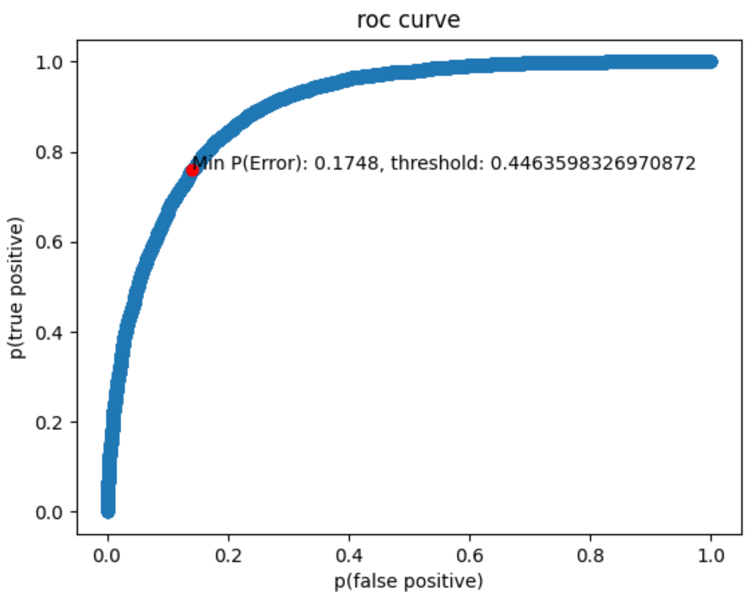

Figure 1: ROC Curve With Optimal Classifier

The threshold for which the minimum probability of error was reached is 0.44 while the minimum probability of error is 0.174. This point is marked on the ROC curve in red.

Using this optimal threshold I then recategorized the data in the validation set and plotted the original and estimated classes side by side:

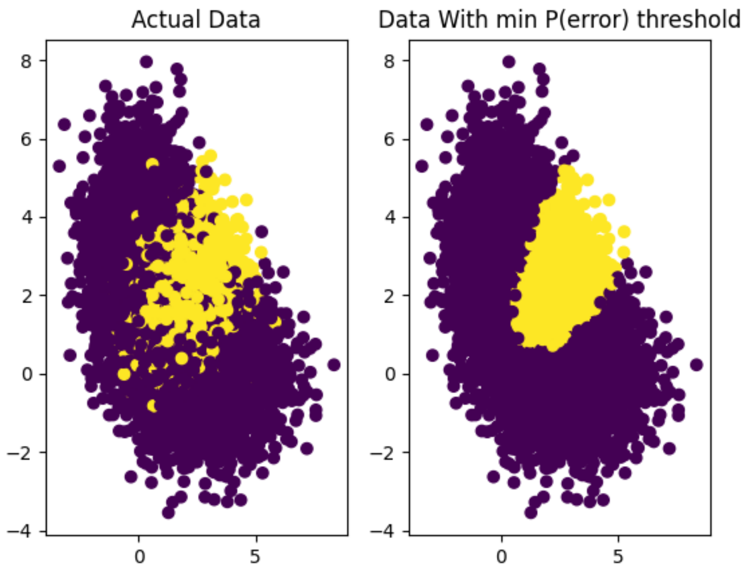

Figure 2: Actual Data Comparison to Optimal Classifier

This gave a pretty good estimate but some of the mixture is clearly lost in the boundary.

Part 2:

Using the three training datasets from earlier I created a linear and quadratic function based approximations of class labels.

For the linear case I used the function:

$$h(x, w) = \frac{1}{1 + e^{-w^T[1, x^T]^T}}$$

I then used the scipy Nelder-Mead minimize function to find the w vector which minimized the cost of the line drawn between class groups, effectively finding the w with the best fit to the data.

This resulted in the following division of the class labels:

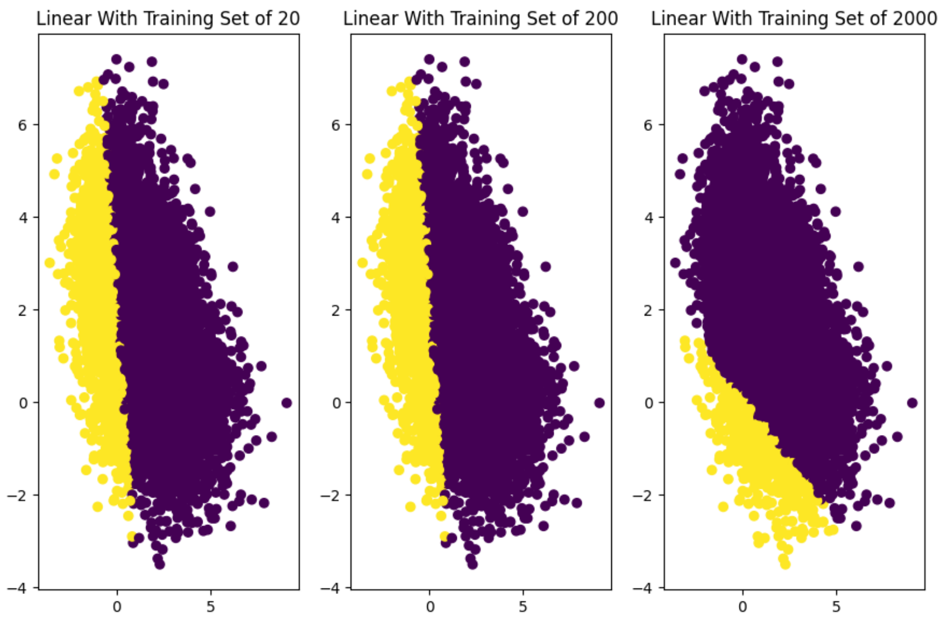

Figure 3: Results of Linear Classifier

The linear classifier does not compare at all to the ability of the optimal classifier. The line bisects the dataset and does not do well in the nuances of categorizing the mixture.

For the quadratic case I used the function:

$$h(x,w) = \frac{1}{1+e^{-w^T[1,x_1,x_2,x_1^2,x_1x_2,x_2^2]^T}}$$

I then used the same methods as above the train the function with the different training sets and applied them to the validation set.

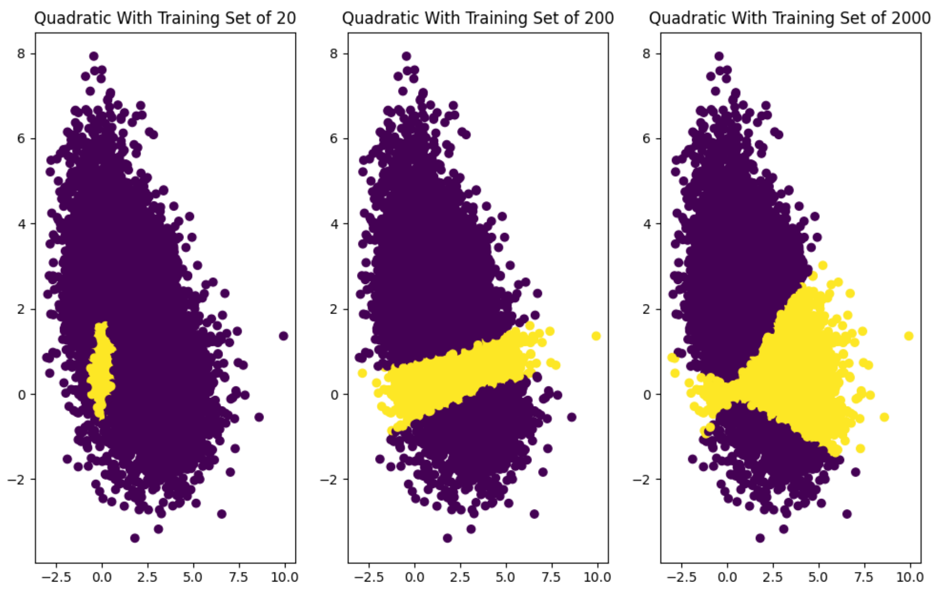

Figure 4: Results of Linear Classifier

The quad classifier performed much better than the linear but still was not close to the performance of the optimal.

Question 2:

For this question I used the provided code to generate a training and validation data set for a maximum likelihood estimator and a MAP estimator. For the ML estimator I used the following equation to estimate w:

$$w = [\frac{1}{N}\sum_{i=1}^{N} Z_i Z_i^T + \gamma I]^{-1}\frac{1}{N}\sum_{i=1}^{N} y_i Z_i$$

For the MAP estimator I used the following equation to estimate w:

$$w = [\frac{1}{N}\sum_{i=1}^{N} Z_i Z_i^T]^{-1}\frac{1}{N}\sum_{i=1}^{N} y_i Z_i$$

I then varied gamma from 10^-4 to 10^7 and calculated the average squared distance for all gamma values and plotted them below:

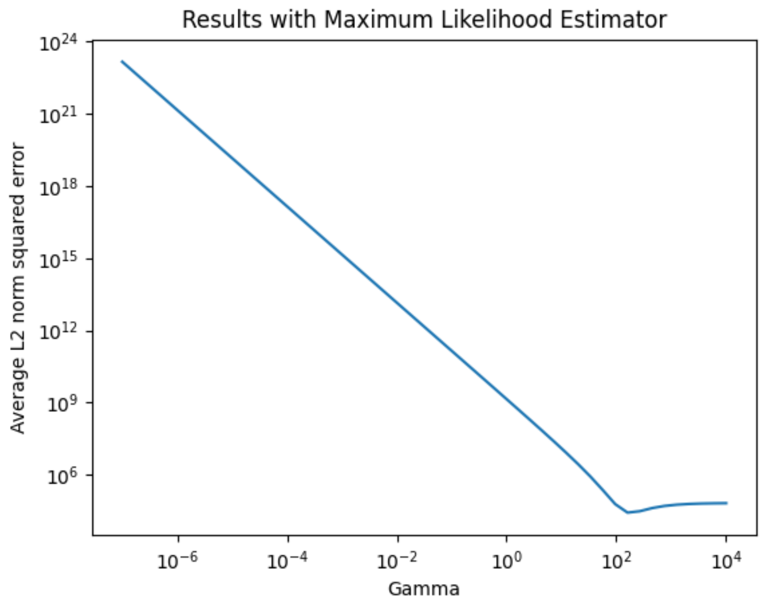

Figure 5: Gamma vs. Error for ML

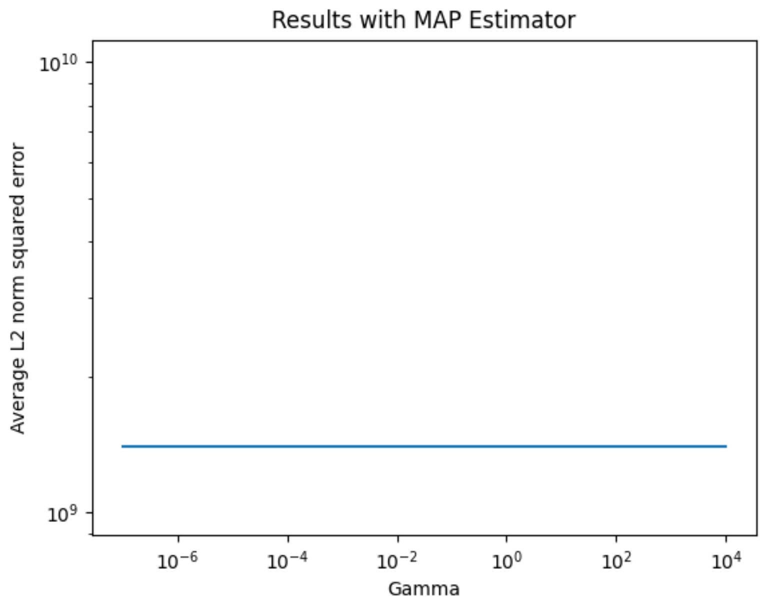

Figure 6: Gamma vs. Error for MAP

Below I compare the original data set, ML estimated, and MAP estimated, all on the validation data set.

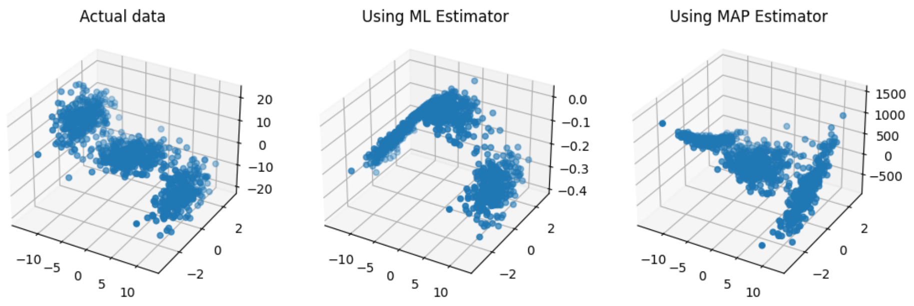

Figure 7: Comparison of Actual and Estimated Datasets

Neither performed particularly well but the ML version had a minimum error that is significantly lower than the MAP. MAP estimator did not vary it's error with different gammas which is likely due to the equations used for calculating w. One can see that the generated w's do create variation in y but not exactly the kind we want. The ML estimator seems to have settled on creating a parabola while the MAP estimator seems to have skewed the results in relation to x_2. I would say that the ML estimator appears to have done a better job with the first and last Gaussian distributions but failed to approximate the middle. The MAP estimator predicted each group but with seemingly equal amounts of error in their distributions.

Part 3:

ML Estimator for Θ

First I set up the maximum likelihood estimator for Θ:

$$\hat{\Theta}_{ml} = argmax_{\theta} \sum_{i=1}^{N} \ln p(Z = z_i|\theta)$$

I broke the dataset D into its individual Z elements so that I could do the next step of representing the likelihood in terms of the corresponding θ for each z.

$$\hat{\Theta}_{ml} = argmax_{\theta} \sum_{i=1}^{N} \ln \theta_{z_i}$$

From here I could create a Lagrange multiplier to find find values for Θ which satisfy the condition that all probabilities of θ_{z_i} add up to 1.

$$L(\theta, \lambda) = \sum_{i=1}^{N} \ln \theta_{z_i} - \lambda(\sum_{k=1}^{K} \theta_k - 1)$$

We can then solve for Θ with the system of equations:

$$\frac{\partial L(\theta, \lambda)}{\partial \theta} = 0, \frac{\partial L(\theta, \lambda)}{\partial \lambda} = 0$$

MAP Estimator for Θ

The MAP estimator is similar to the ML except we include the prior:

$$\hat{\Theta}_{ml} = argmax_{\theta} \sum_{i=1}^{N} \ln \theta_{z_i} + \ln p(\theta)$$

I then substituted the Dirichlet distribution given for $p(\Theta)$

$$\hat{\Theta}_{ml} = argmax_{\theta} \sum_{i=1}^{N} \ln \theta_{z_i} + \ln \frac{\Gamma(\sum_{k=1}^{K} \alpha_k)}{\prod_{k=1}^{K} \Gamma(\alpha_k)} \sum_{k=1}^{K} \theta_k^{\alpha_k - 1}$$

I then used the Lagrange as before, but this time one needs to use the hyper-parameter α, maybe using k-fold cross validation.

Code

Code for Question 1

```
import numpy as np
import numpy.matlib as ml
import matplotlib.pyplot as plt
from scipy.optimize import minimize

priors = np.array([[0.65, 0.35]])

m01 = np.array([3, 0])
C01 = np.array([
    [2,0],
    [0,1]
])
m02 = np.array([0, 3])
C02 = np.array([
    [1,0],
    [0,2]
])

m1 = np.array([2, 2])
C1 = np.array([
    [1,0],
    [0,1]
])

def dataFromGMM(N, priors, means, covs):
    components = priors.size
    x = np.zeros((2, N))
    labels = np.zeros(N)
    u = np.random.rand(N)
    csum = np.cumsum(priors)
    # go through all u and assign the labels to x
    for l in range(0, components):
        # find all u that are less than the threshold
        indl = np.where(u <= csum[l])[0]
        # track the number of samples below this threshold
        Nl = len(indl)
        # remove these samples from u
        u[indl] = 1.1
        # set all labels 0 to nl to i
        labels[indl] = l
        # generate values for x
        x[:, indl] = np.random.multivariate_normal(means[l], covs[l], Nl).T

    return (x, labels)

def genData(N, priors):
    u = np.random.rand(N)
    cumsum = np.cumsum(priors)
    x = np.zeros((2, N))
    labels = np.zeros(N)
    for l in range(2):
        indl = np.where(u <= cumsum[l])[0]
        # set the labels
        labels[indl] = l
        # make those values in u too high
        u[indl] = 1.1

        if l == 1:
            N1 = indl.size
            y = np.random.multivariate_normal(m1, C1, N1).T
            x[:, indl] = y
        elif l == 0:
            N2 = indl.size
            weights = np.array([0.5, 0.5])
            means = np.array([m01, m02])
            covs = np.array([C01, C02])
            # do the gmm
            z, zlabel = dataFromGMM(N2, weights, means, covs)
            x[:, indl] = z

    return x, labels

def evalGaussianPDF(x, mu, Sigma):
    N = x[1].size # number of items in x
    # normalization constant (const with respect to x)
    C = (2*np.pi)**(-2/2)*np.linalg.det(Sigma)**(1/2)
    #E = -0.5 * np.sum((x-ml.repmat(mu,1,N)).T * (np.linagl.inv(Sigma)* (x-ml.re
    like = np.zeros(N)
    for i in range(N):
        like[i]  = C * np.exp(-0.5 * np.sum(((x[:, i]-mu).T * np.linalg.inv(Sigm

    return like

def rocCurve(scores,labels):
    # thresholds start with the smallest ratio - eps, all in between, the max +
    thresholds = np.array(np.sort(scores))
    nt = thresholds.size
    pfp = np.zeros(nt)
    ptp = np.zeros(nt)
    perr = np.zeros(nt)

    for i in range(nt):
        tau = thresholds[i]
        # map all decisions to labels given the current tau
        #d = np.where(scores >= tau, 0, 1)
        d = np.greater_equal(scores, tau).astype(int)
        true_positive = np.equal(d, 1) & np.equal(labels, 1)
        true_negative = np.equal(d, 0) & np.equal(labels, 0)
        false_positive = np.equal(d, 1) & np.equal(labels, 0)
        false_negative = np.equal(d, 0) & np.equal(labels, 1)
        ptp[i] = true_positive.sum() / (true_positive.sum() + false_negative.sum
        pfp[i] = false_positive.sum() / (false_positive.sum() + true_negative.su
        perr[i] = np.not_equal(d, labels).nonzero()[0].size/labels.size

    return pfp, ptp, perr, thresholds


# get the training and validation data
d20, labels20 = genData(20, priors)
d200, labels200 = genData(200, priors)
d2000, labels2000 = genData(2000, priors)
validate, validate_labels = genData(10000, priors)

# classify and show ROC, and min perror
def part1(v, v_labels):
    pxgivenl = np.zeros((2, 10000))
    pxgivenl[0,:] = 0.5 * evalGaussianPDF(v,m01, C01) + 0.5 * evalGaussianPDF(v,
    pxgivenl[1,:] = evalGaussianPDF(v,m1, C1)
    px = (priors * pxgivenl[:].T).T.sum(axis=0)
    classPosteriors = pxgivenl * ml.repmat(priors.T, 1, 10000) / ml.repmat(px, 2
    expectedRisks = np.matmul(np.ones(2)-np.eye(2), classPosteriors)
    ratios = (classPosteriors[1,:]/classPosteriors[0,:])

    pfp, ptp, perr, thresholds = rocCurve(ratios, v_labels)

    part1 = plt.figure()
    p1plots = part1.add_subplot()
    p1plots.set_title("roc curve")
    plt.setp(p1plots, xlabel="p(false positive)", ylabel="p(true positive)")
    p1plots.scatter(pfp, ptp)
    # find the lowest p error
    minErrIdx = np.argmin(perr)
    p1plots.scatter(pfp[minErrIdx], ptp[minErrIdx], color="#ff0000")
    p1plots.annotate(f"Min P(Error): {perr[minErrIdx]}, threshold: {thresholds[m

    comparison, cplots = plt.subplots(1,2)
    # the actual data
    cplots[0].scatter(validate[0], validate[1], c=v_labels)
    cplots[0].set_title("Actual Data")
    # data with the threshold we just used
    decisions = np.where(ratios >= thresholds[minErrIdx], 1, 0)
    cplots[1].scatter(validate[0], validate[1], c=decisions)
    cplots[1].set_title("Data With min P(error) threshold")

    plt.show()

def part2_linear():
    fig, lall = plt.subplots(1, 3)

    # train three log near linear
    w20 = minimize(linCost, np.zeros(3), d20, method='Nelder-Mead').x
    decisions20 = np.zeros((1,10000))
    z = np.c_[np.ones((validate.shape[1])), validate.T].T
    h = 1/(1+np.exp(-(np.dot(w20.T,z))))
    decisions20[0,:] = (h[:]>=0.5).astype(int)
    lall[0].set_title("Linear With Training Set of 20")
    lall[0].scatter(validate[0], validate[1], c=decisions20)

    w200 = minimize(linCost, np.zeros(3), d200, method='Nelder-Mead').x
    decisions200 = np.zeros((1,10000))
    z = np.c_[np.ones((validate.shape[1])), validate.T].T
    h = 1/(1+np.exp(-(np.dot(w200.T,z))))
    decisions200[0,:] = (h[:]>=0.5).astype(int)
    lall[1].set_title("Linear With Training Set of 200")
    lall[1].scatter(validate[0], validate[1], c=decisions200)

    w2000 = minimize(linCost, np.zeros(3), d2000, method='Nelder-Mead').x
    decisions2000 = np.zeros((1,10000))
    z = np.c_[np.ones((validate.shape[1])), validate.T].T
    h = 1/(1+np.exp(-(np.dot(w2000.T,z))))
    decisions2000[0,:] = (h[:]>=0.5).astype(int)
    lall[2].set_title("Linear With Training Set of 2000")
    lall[2].scatter(validate[0], validate[1], c=decisions2000)


    plt.show()

def part2_quad():
    fig, lall = plt.subplots(1, 3)

    w20 = minimize(quadCost, np.zeros(6), d20, method='Nelder-Mead').x
    decisions20 = np.zeros((1,10000))
    z = np.c_[np.ones((validate.shape[1])), validate[0], validate[1], validate[
    h = 1/(1+np.exp(-(np.dot(w20.T,z))))
    decisions20[0,:] = (h[:]>=0.5).astype(int)
    lall[0].set_title("Quadratic With Training Set of 20")
    lall[0].scatter(validate[0], validate[1], c=decisions20)

    w200 = minimize(quadCost, np.zeros(6), d200, method='Nelder-Mead').x
    decisions200 = np.zeros((1,10000))
    z = np.c_[np.ones((validate.shape[1])), validate[0], validate[1], validate[
    h = 1/(1+np.exp(-(np.dot(w200.T,z))))
    decisions200[0,:] = (h[:]>=0.5).astype(int)
    lall[1].set_title("Quadratic With Training Set of 200")
    lall[1].scatter(validate[0], validate[1], c=decisions200)

    w2000 = minimize(quadCost, np.zeros(6), d2000, method='Nelder-Mead').x
    decisions2000 = np.zeros((1,10000))
    z = np.c_[np.ones((validate.shape[1])), validate[0], validate[1], validate[
    h = 1/(1+np.exp(-(np.dot(w2000.T,z))))
    decisions2000[0,:] = (h[:]>=0.5).astype(int)
    lall[2].set_title("Quadratic With Training Set of 2000")
    lall[2].scatter(validate[0], validate[1], c=decisions2000)

    plt.show()

def linCost(w, x):
    z = np.c_[np.ones((x.shape[1])), x.T].T
    denom = 1 + np.exp(-np.dot(w.T , z)).sum()
    return 1/denom

def quadCost(w, x):
    z = np.c_[np.ones((x.shape[1])), x[0], x[1], x[0]*x[0], x[0]*x[1], x[1]*x[1
    denom = 1 + np.exp(-np.dot(w.T , z)).sum()
    return 1/denom

part1(validate, validate_labels)

part2_linear()

part2_quad()
```

Code for Question 2

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

def hw2q2():
    Ntrain = 100
    data = generateData(Ntrain)
    #plot3(data[0,:],data[1,:],data[2,:])
    xTrain = data[0:2,:]
    yTrain = data[2,:]

    Ntrain = 1000
    data = generateData(Ntrain)
    #plot3(data[0,:],data[1,:],data[2,:])
    xValidate = data[0:2,:]
    yValidate = data[2,:]

    return xTrain,yTrain,xValidate,yValidate
```

```
def generateData(N):
    gmmParameters = {}
    gmmParameters['priors'] = [.3,.4,.3] # priors should be a row vector
    gmmParameters['meanVectors'] = np.array([[-10, 0, 10], [0, 0, 0], [10, 0, -
    gmmParameters['covMatrices'] = np.zeros((3, 3, 3))
    gmmParameters['covMatrices'][:,:,0] = np.array([[1, 0, -3], [0, 1, 0], [-3,
    gmmParameters['covMatrices'][:,:,1] = np.array([[8, 0, 0], [0, .5, 0], [0,
    gmmParameters['covMatrices'][:,:,2] = np.array([[1, 0, -3], [0, 1, 0], [-3,
    x,labels = generateDataFromGMM(N,gmmParameters)
    return x

def generateDataFromGMM(N,gmmParameters):
#    Generates N vector samples from the specified mixture of Gaussians
#    Returns samples and their component labels
#    Data dimensionality is determined by the size of mu/Sigma parameters
    priors = gmmParameters['priors'] # priors should be a row vector
    meanVectors = gmmParameters['meanVectors']
    covMatrices = gmmParameters['covMatrices']
    n = meanVectors.shape[0] # Data dimensionality
    C = len(priors) # Number of components
    x = np.zeros((n,N))
    labels = np.zeros((1,N))
    # Decide randomly which samples will come from each component
    u = np.random.random((1,N))
    thresholds = np.zeros((1,C+1))
    thresholds[:,0:C] = np.cumsum(priors)
    thresholds[:,C] = 1
    for l in range(C):
        indl = np.where(u <= float(thresholds[:,l]))
        Nl = len(indl[1])
        labels[indl] = (l+1)*1
        u[indl] = 1.1
        x[:,indl[1]] = np.transpose(np.random.multivariate_normal(meanVectors[:,

    return x,labels

def plot3(a,b,c,mark="o",col="b"):
  fig = plt.figure()
  ax = fig.add_subplot(projection='3d')
  ax.scatter(a, b, c,marker=mark,color=col)
  ax.set_xlabel("x1")
```

```
    ax.set_ylabel("x2")
    ax.set_zlabel("y")
    ax.set_title('Training Dataset')

# rate each w by using the validation set
def create_y(x, w):
    return w[0] + w[1]*x[0] + w[2]*x[1] + w[3]*x[0]*x[0] + w[4]*x[0]*x[1] + w[5]

xTrain,yTrain,xValidate,yValidate = hw2q2()

# assume that varience is 0.1, lambda i


# make z
Z = np.c_[np.ones((xTrain.shape[1])), xTrain[0], xTrain[1], xTrain[0]*xTrain[0],
R = np.zeros((6,6))
for i in range(Z.shape[1]):
    R += Z[:, i]*Z[:,i].T
R = R/Z.shape[1]

q = np.zeros((1,6))
for i in range(Z.shape[1]):
    q += yTrain[i]*Z[:,i]
q = q/Z.shape[1]
# range of gamma values
gammaList = np.logspace(-7, 4, num=50)

# estimate w
w_map = np.zeros((gammaList.size, 6))
w_ml = np.zeros((gammaList.size, 6,))
# average l2 norm of this model for each gamma
l2_ml = np.zeros((1, gammaList.size))
l2_map = np.zeros((1, gammaList.size))

for i in range(gammaList.size):
    gamma = gammaList[i]
    w_map[i] = np.dot(np.linalg.inv(R + np.eye(6)), q.T).T
    w_ml[i] = np.dot(np.linalg.inv(R + gamma*np.eye(6)), q.T).T

    # for all gamma, create y and evaluate the average squared error btwn gener
    mly = create_y(xValidate, w_ml[i])
    l2_ml[:, i] = pow(np.linalg.norm(yValidate - mly), 2)

    mapy = create_y(xValidate, w_map[i])
    l2_map[:, i] = pow(np.linalg.norm(yValidate - mapy), 2)


mlfig = plt.figure()
ml_plot = mlfig.add_subplot()
ml_plot.set_title("Results with Maximum Likelihood Estimator")
ml_plot.plot(gammaList, l2_ml[0])
ml_plot.set_yscale('log')
ml_plot.set_xscale('log')
plt.setp(ml_plot, xlabel="Gamma", ylabel="Average L2 norm squared error")

mapfig = plt.figure()
map_plot = mapfig.add_subplot()
map_plot.set_title("Results with MAP Estimator")
map_plot.plot(gammaList, l2_map[0])
map_plot.set_yscale('log')
map_plot.set_xscale('log')
plt.setp(map_plot, xlabel="Gamma", ylabel="Average L2 norm squared error")

allfig = plt.figure()
actual = allfig.add_subplot(1,3,1, projection='3d')
ml_data = allfig.add_subplot(1,3,2, projection='3d')
map_data = allfig.add_subplot(1,3,3, projection='3d')

actual.set_title("Actual data")
actual.scatter(xValidate[0], xValidate[1], yValidate)
ml_data.set_title("Using ML Estimator")
ml_data.scatter(xValidate[0], xValidate[1], mly)
map_data.set_title("Using MAP Estimator")
map_data.scatter(xValidate[0], xValidate[1], mapy)

plt.show()
```