Tyler Cerilli

EECE 5644

Homework 2

**Problem 1**

np.random.seed(1)

Number of samples from Class 1: 16, Class 2: 4

Number of samples from Class 1: 127, Class 2: 73

Number of samples from Class 1: 1309, Class 2: 691

Number of samples from Class 1: 6391, Class 2: 3609

Chart, scatter chart

Description automatically generated

**Part 1**

Threshold value: 1.8571428571428574

Confusion Matrix MAP (rows: Predicted class, columns: True class):

[[5634 908]

[ 757 2701]]

Total Number of Misclassified Samples (MAP): 1665

Gamma MAP (Theoretical): 1.8571428571428574

Probability of Error(MAP): 0.16649999999999998

Best Gamma (ERM): 1.649473324917088

Probability of Error(Empirical): 0.1649

Chart, line chart

Description automatically generatedChart, scatter chart

Description automatically generated

**Part 2**

LINEAR

2 batches of size 10:

Logistic-Linear N=20 GD Theta:

[-0.02831339 -0.50197184 -0.09912327]

Logistic-Linear N=20 NLL: 1.2686217324687203

The total error achieved with this classifier is 0.442

20 batches of size 10:

Logistic-Linear N=200 GD Theta:

[-0.87731068 0.18788255 0.09950977]

Logistic-Linear N=200 NLL: 12.60622593092242

The total error achieved with this classifier is 0.367

200 batches of size 10:

Logistic-Linear N=2000 GD Theta:

[-1.90320918 0.37300914 0.3563989 ]

Logistic-Linear N=2000 NLL: 121.38572234930577

The total error achieved with this classifier is 0.346

A picture containing application

Description automatically generated

QUADRATIC

2 batches of size 10:

Logistic-Linear N=20 GD Theta:

[ 0.08970478 -0.81070515 -1.2917546 -0.01122395 1.0294441 -0.0422405 ]

Logistic-Linear N=20 NLL: 0.4579893514019052

The total error achieved with this classifier is 0.276

20 batches of size 10:

Logistic-Linear N=200 GD Theta:

[ 0.35070649 -0.18782312 -0.47789395 -0.19579952 0.92950008 -0.20815267]

Logistic-Linear N=200 NLL: 8.668329342392768

The total error achieved with this classifier is 0.196

200 batches of size 10:

Logistic-Linear N=2000 GD Theta:

[-1.27830878 0.72130575 0.8245824 -0.32236347 0.55910594 -0.36879039]

Logistic-Linear N=2000 NLL: 81.01347096977517

The total error achieved with this classifier is 0.164

A picture containing diagram

Description automatically generated

Let us look at just the quadratic results, as they are the best, and the same concepts apply to both the quadratic and linear case…

The number of training samples increase the performance of the classifiers drastically. From 20 to 200 to 2000 training samples validated on 10000 samples, the probability of error (performance) decreases from 0.276 (Ntrain =20) to 0.196 (Ntrain =200) to 0.164 (Ntrain=2000). Since we want to mitigate error, a decrease in error is proportional to an increase in performance. The same exact pattern/trend exists in the linear function form as well.

The function form also has a drastic effect and is an especially important component to consider. For both the linear and quadratic case, the best performance is achieved using the highest training set (Ntrain=2000). The linear form achieves a probability of error of 0.346, while the quadratic achieves a probability of error of 0.164. Clearly the quadratic case outperforms the linear case. This is the same across all sizes of training samples tested.

In comparison to the theoretically optimal classifier from Part 1 (p(error) = 0.16649999999999998), the linear form performs much worse, whereas the quadratic form performs better (for this seed). The fits and classifies the data just as well as the theoretically optimal classifier when we have sample set of ~2000 samples.

**Problem 2**

np.random.seed(7)

10 batches of size 10:

theta start:

[ 0.20139884 0.80473259 -0.30712783 -0.19619016 -0.69316614 1.56372702

0.3546326 -0.36475778 -0.32149488 0.10854485]

theta MLE:

[ 0.20269365 0.79902827 -0.30291402 -0.00566083 -0.65097462 1.56869389

-0.01630005 0.02123632 -0.27712615 0.12454508]

theta MAP:

[-0.3993506 0.03506325 0.2527343 -0.00449423 -0.01386988 0.60370334

-0.01026672 -0.00096463 -0.0428286 -0.1144727 ]

MSE ML (GD): 40.81441651726255

MSE MAP: 5.583247585813262

**Chart, scatter chart

Description automatically generated**

**Chart

Description automatically generated**

As gamma in the MAP model increases, the MSE decreases rapidly until about ~ gamma = 50-100 where it flattens out. The curve seems to follow a somewhat exponential decay, especially when we take a closer look at it.

The MAP estimate is related to the ML estimate such that the MSE of the ML estimate is theoretically converging to the MSE point for MAP.

Chart

Description automatically generated

**APPENDIX:**

**PROBLEM 1**

from email.errors import MultipartInvariantViolationDefect

import matplotlib.pyplot as plt # For general plotting

import numpy as np

from scipy import rand

from scipy.stats import multivariate\_normal # MVN not univariate

from sklearn.metrics import confusion\_matrix

from modules import prob\_utils

from math import ceil, floor

np.set\_printoptions(suppress=True)

np.random.seed(1)      # seed 7 is really bad for quadratic

plt.rc('font', size=18)          # controls default text sizes

plt.rc('axes', titlesize=18)     # fontsize of the axes title

plt.rc('axes', labelsize=18)     # fontsize of the x and y labels

plt.rc('xtick', labelsize=14)    # fontsize of the tick labels

plt.rc('ytick', labelsize=14)    # fontsize of the tick labels

plt.rc('legend', fontsize=16)    # legend fontsize

plt.rc('figure', titlesize=18)   # fontsize of the figure title

def gen\_data(N):

    mu0 = np.array([[3, 0],

                   [0, 3]])  # Gaussian distributions means

    sigma0 = np.array([[[2, 0],

                       [0, 1]],

                      [[1, 0],

                       [0, 2]]])  # Gaussian distributions covariance matrices

    mu1 = np.array([[2,2]])

    sigma1 = np.array([[[1,0],

                        [0,1]]])

    n = 2 #mu0.shape[0]

    # Class priors

    priors = np.array([0.65, 0.35])  # Likelihood of each distribution to be selected

    weights = np.array([.5,.5])

    # Determine number of mixture components

    Cw = len(weights)

    C = len(priors)

    # Output samples and labels

    X = np.zeros([N, n])

    labels = np.zeros(N) # KEEP TRACK OF THIS

    # Plot for original data and their true labels

    labels = np.random.rand(N) >= priors[0]

    L = np.array(range(C))

    Nl = np.array([sum(labels == l) for l in L])

    print("Number of samples from Class 1: {:d}, Class 2: {:d}".format(Nl[0], Nl[1]))

    gmm\_params = prob\_utils.GaussianMixturePDFParameters(weights, Cw, mu0, np.transpose(sigma0))

    gmm\_X,\_ = prob\_utils.generate\_mixture\_samples(Nl[0], n, gmm\_params, True)

    X = np.zeros((N, n))

    X[labels == 0, :] =  gmm\_X.T

    X[labels == 1, :] =  multivariate\_normal.rvs(mu1[0], sigma1[0], Nl[1])

    return X, labels, Nl

from sys import float\_info # Threshold smallest positive floating value

# Generate ROC curve samples

def estimate\_roc(discriminant\_score, label,N):

    Nlabels = np.array((sum(label == 0), sum(label == 1)))

    sorted\_score = sorted(discriminant\_score)

    # Use tau values that will account for every possible classification split

    taus = ([sorted\_score[0] - float\_info.epsilon] +

            sorted\_score +

            [sorted\_score[-1] + float\_info.epsilon])

    # Calculate the decision label for each observation for each gamma

    decisions = [discriminant\_score >= t for t in taus]

    ind10 = [np.argwhere((d==1) & (label==0)) for d in decisions]

    p10 = [len(inds)/Nlabels[0] for inds in ind10]

    ind11 = [np.argwhere((d==1) & (label==1)) for d in decisions]

    p11 = [len(inds)/Nlabels[1] for inds in ind11]

    ind01 = [np.argwhere((d==0) & (label==1)) for d in decisions]

    p01 = [len(inds)/Nlabels[1] for inds in ind01]

    # To find the best value for gamma from the dataset (not theoretical)

    # Here, we find value with lowest probability or error, and convert

    # taus back from log using exp

    prob\_error\_erm = np.zeros(len(p01))

    for i in range(len(p10)):

        prob\_error\_erm[i] = np.array((p10[i], p01[i])).dot(Nlabels.T / N)

    best\_gamma = np.exp(taus[np.argmin(prob\_error\_erm)])

    p\_error\_erm = min(prob\_error\_erm)

    data\_point = np.array([p10[np.argmin(prob\_error\_erm)], p11[np.argmin(prob\_error\_erm)]])#.dot(Nlabels.T / N)

    # ROC has FPR on the x-axis and TPR on the y-axis

    roc = np.array((p10, p11))

    return roc, taus, best\_gamma, p\_error\_erm, data\_point

# Define the logistic/sigmoid function

def sigmoid(z):

    return 1.0 / (1 + np.exp(-z))

# Define the prediction function y = 1 / (1 + np.exp(-X\*theta))

# X.dot(theta) inputs to the sigmoid referred to as logits

def predict\_prob(X, theta):

    logits = X.dot(theta)

    return sigmoid(logits)

# NOTE: This implementation may encounter numerical stability issues...

# Read into the log-sum-exp trick OR use a method like: sklearn.linear\_model import LogisticRegression

def log\_reg\_loss(theta, X, y):

    # Size of batch

    B = X.shape[0]

    # Logistic regression model g(X \* theta)

    predictions = predict\_prob(X, theta)

    # NLL loss, 1/N sum [y\*log(g(X\*theta)) + (1-y)\*log(1-g(X\*theta))]

    error = predictions - y

    nll = -np.mean(y\*np.log(predictions) + (1 - y)\*np.log(1 - predictions))

    # Partial derivative for GD

    g = (1 / B) \* X.T.dot(error)

    # Logistic regression loss, NLL (binary cross entropy is another interpretation)

    return nll, g

# Breaks the matrix X and vector y into batches

def batchify(X, y, batch\_size, N):

    X\_batch = []

    y\_batch = []

    # Iterate over N in batch\_size steps, last batch may be < batch\_size

    for i in range(0, N, batch\_size):

        nxt = min(i + batch\_size, N + 1)

        X\_batch.append(X[i:nxt, :])

        y\_batch.append(y[i:nxt])

    return X\_batch, y\_batch

def gradient\_descent(loss\_func, theta0, X, y, N, \*args, \*\*kwargs):

    # Mini-batch GD. Stochastic GD if batch\_size=1.

    # Break up data into batches and work out gradient for each batch

    # Move parameters theta in that direction, scaled by the step size.

    # Options for total sweeps over data (max\_epochs),

    # and parameters, like learning rate and threshold.

    # Default options

    max\_epoch = kwargs['max\_epoch'] if 'max\_epoch' in kwargs else 200

    alpha = kwargs['alpha'] if 'alpha' in kwargs else 0.1

    epsilon = kwargs['tolerance'] if 'tolerance' in kwargs else 1e-6

    batch\_size = kwargs['batch\_size'] if 'batch\_size' in kwargs else 10

    # Turn the data into batches

    X\_batch, y\_batch = batchify(X, y, batch\_size, N)

    num\_batches = len(y\_batch)

    print("%d batches of size %d:" % (num\_batches, batch\_size))

    theta = theta0

    m\_t = np.zeros(theta.shape)

    trace = {}

    trace['loss'] = []

    trace['theta'] = []

    # Main loop:

    for epoch in range(1, max\_epoch + 1):

        # print("epoch %d\n" % epoch)

        loss\_epoch = 0

        for b in range(num\_batches):

            X\_b = X\_batch[b]

            y\_b = y\_batch[b]

            # print("epoch %d batch %d\n" % (epoch, b))

            # Compute NLL loss and gradient of NLL function

            loss, gradient = loss\_func(theta, X\_b, y\_b, \*args)

            loss\_epoch += loss

            # Steepest descent update

            theta = theta - alpha \* gradient

            # Terminating Condition is based on how close we are to minimum (gradient = 0)

            if np.linalg.norm(gradient) < epsilon:

                print("Gradient Descent has converged after {} epochs".format(epoch))

                break

        # Storing the history of the parameters and loss values per epoch

        trace['loss'].append(np.mean(loss\_epoch))

        trace['theta'].append(theta)

        # Also break epochs loop

        if np.linalg.norm(gradient) < epsilon:

            break

    return theta, trace

def create\_prediction\_score\_grid(theta, poly\_type):

    # Create coordinate matrices determined by the sample space; can add finer intervals than 100 if desired

    xx, yy = np.meshgrid(np.linspace(bounds\_X[0], bounds\_X[1], 200), np.linspace(bounds\_Y[0], bounds\_Y[1], 200))

    # Augment grid space with bias ones vector and basis expansion if necessary

    grid = np.c\_[xx.ravel(), yy.ravel()]

    grid\_aug = np.column\_stack((np.ones(200\*200), grid))

    if poly\_type == 'Q':

        grid\_aug = quadratic\_transformation(grid\_aug)

    # Z matrix are the predictions resulting from sigmoid on the provided model parameters

    Z = predict\_prob(grid\_aug, theta).reshape(xx.shape)

    return xx, yy, Z

def plot\_prediction\_contours(X, theta, ax, poly\_type):

    xx, yy, Z = create\_prediction\_score\_grid(theta, poly\_type)

    # Once reshaped as a grid, plot contour of probabilities per input feature (ignoring bias)

    cs = ax.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.55)

    ax.set\_xlim([bounds\_X[0], bounds\_X[1]])

    ax.set\_ylim([bounds\_Y[0], bounds\_Y[1]])

def plot\_decision\_boundaries(X, labels, theta, ax, poly\_type):

    # Plots original class labels and decision boundaries

    ax.plot(X[labels==0, 1], X[labels==0, 2], 'o', label="Class 0")

    ax.plot(X[labels==1, 1], X[labels==1, 2], '+', label="Class 1")

    xx, yy, Z = create\_prediction\_score\_grid(theta, poly\_type)

    # Once reshaped as a grid, plot contour of probabilities per input feature (ignoring bias)

    cs = ax.contour(xx, yy, Z, levels=1, colors='k')

    ax.set\_xlabel(r"$x\_1$")

    ax.set\_ylabel(r"$x\_2$")

    ax.set\_aspect('equal')

def report\_logistic\_classifier\_results(X, theta, labels, N\_labels, ax, poly\_type):

    """

    Report the probability of error and plot the classified data, plus predicted

    decision contours of the logistic classifier applied to the data given.

    """

    predictions = predict\_prob(X, theta)

    # Predicted decisions based on the default 0.5 threshold (higher probability mass on one side or the other)

    decisions = np.array(predictions >= 0.5)

    # True Negative Probability Rate

    ind\_00 = np.argwhere((decisions == 0) & (labels == 0))

    tnr = len(ind\_00) / N\_labels[0]

    # False Positive Probability Rate

    ind\_10 = np.argwhere((decisions == 1) & (labels == 0))

    fpr = len(ind\_10) / N\_labels[0]

    # False Negative Probability Rate

    ind\_01 = np.argwhere((decisions == 0) & (labels == 1))

    fnr = len(ind\_01) / N\_labels[1]

    # True Positive Probability Rate

    ind\_11 = np.argwhere((decisions == 1) & (labels == 1))

    tpr = len(ind\_11) / N\_labels[1]

    prob\_error = fpr\*priors[0] + fnr\*priors[1]

    print("The total error achieved with this classifier is {:.3f}\n".format(prob\_error))

    # Plot all decisions (green = correct, red = incorrect)

    ax.plot(X[ind\_00, 1], X[ind\_00, 2], 'og', label="Class 0 Correct", alpha=.25)

    ax.plot(X[ind\_10, 1], X[ind\_10, 2], 'or', label="Class 0 Wrong")

    ax.plot(X[ind\_01, 1], X[ind\_01, 2], '+r', label="Class 1 Wrong")

    ax.plot(X[ind\_11, 1], X[ind\_11, 2], '+g', label="Class 1 Correct", alpha=.25)

    # Draw the decision boundary based on whether its linear (L) or quadratic (Q)

    plot\_prediction\_contours(X, theta, ax, poly\_type)

    ax.set\_aspect('equal')

# Can also use: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html

def quadratic\_transformation(X):

    n = X.shape[1]

    phi\_X = X

    # Take all monic polynomials for a quadratic

    phi\_X = np.column\_stack((phi\_X, X[:, 1] \* X[:, 1], X[:, 1] \* X[:, 2], X[:, 2] \* X[:, 2]))

    return phi\_X

# Options for mini-batch gradient descent

opts = {}

opts['max\_epoch'] = 1000

opts['alpha'] = 1e-3

opts['tolerance'] = 1e-3

opts['batch\_size'] = 10

# def main():

######################################

###              MAIN              ###

######################################

mu0 = np.array([[3, 0],

                [0, 3]])  # Gaussian distributions means

sigma0 = np.array([[[2, 0],

                    [0, 1]],

                    [[1, 0],

                    [0, 2]]])  # Gaussian distributions covariance matrices

mu1 = np.array([[2,2]])

sigma1 = np.array([[[1,0],

                    [0,1]]])

# Ntrain = np.array([20,200,2000])

# Ntest = 10000

N = np.array([20,200,2000,10000])

# Class priors

priors = np.array([0.65, 0.35])  # Likelihood of each distribution to be selected

weights = np.array([.5,.5])

# Determine number of mixture components

C = len(priors)

data = []

labels = []

Nl = []

for i in range(len(N)):

    temp1,temp2,Nlabels = gen\_data(N[i])

    temp1 = np.column\_stack((np.ones(N[i]), temp1))

    data.append(temp1)

    labels.append(temp2)

    Nl.append(np.array((sum(labels[i] == 0), sum(labels[i] == 1))))

# Ny\_valid = np.array((sum(labels[3] == 0), sum(labels[3] == 1)))

# Use the validation set's sample space to bound the grid of inputs

# Work out bounds that span the input feature space (x\_1 and x\_2)

bounds\_X = np.array((floor(np.min(data[3][:,1])), ceil(np.max(data[3][:,1]))))

bounds\_Y = np.array((floor(np.min(data[3][:,2])), ceil(np.max(data[3][:,2]))))

# Plot the original data and their true labels

# Choose which data set to use [20,200,2000,10000]

# dataset = 3

fig\_plot, ax\_plot = plt.subplots(2, 2, figsize=(10, 10));

for dataset in range(len(N)):

    if dataset < 2:

        j =0

    else:

        j=1

    ax\_plot[dataset%2, j].plot(data[dataset][labels[dataset]==0, 1], data[dataset][labels[dataset]==0, 2], 'bo', label="Class 0")

    ax\_plot[dataset%2, j].plot(data[dataset][labels[dataset]==1, 1], data[dataset][labels[dataset]==1, 2], 'k+', label="Class 1")

    # plt.plot(Dtrain20[labels2000==0, 0], Dtrain20[labels2000==0, 1], 'bo', label="Class 0")

    # plt.plot(Dtrain20[labels2000==1, 0], Dtrain20[labels2000==1, 1], 'k+', label="Class 1")

    ax\_plot[dataset%2, j].legend()

    ax\_plot[dataset%2, j].set\_xlabel(r"$x\_1$")

    ax\_plot[dataset%2, j].set\_ylabel(r"$x\_2$")

    ax\_plot[dataset%2, j].set\_aspect('equal')

    ax\_plot[dataset%2, j].set\_title("Data and True Class for N={}".format(data[dataset].shape[0]))

    # plt.tight\_layout()

#    fig = plt.figure(figsize=(9, 9))

    # plt.plot(data[dataset][labels[dataset]==0, 1], data[dataset][labels[dataset]==0, 2], 'bo', label="Class 0")

    # plt.plot(data[dataset][labels[dataset]==1, 1], data[dataset][labels[dataset]==1, 2], 'k+', label="Class 1")

    # # plt.plot(Dtrain20[labels2000==0, 0], Dtrain20[labels2000==0, 1], 'bo', label="Class 0")

    # # plt.plot(Dtrain20[labels2000==1, 0], Dtrain20[labels2000==1, 1], 'k+', label="Class 1")

    # plt.legend()

    # plt.xlabel(r"$x\_1$")

    # plt.ylabel(r"$x\_2$")

    # plt.title("Data and True Class Labels")

    # plt.tight\_layout()

plt.show()

n = 2 #mu0.shape[0]

# Caculate threshold rule

Lambda = np.ones((C, C)) - np.identity(C)

gamma\_map = priors[0] / priors[1]

print(f'Threshold value: {gamma\_map}')

# u = np.random.rand(N[3])

L = np.array(range(C))

Nl = np.array([sum(labels[3] == l) for l in L])

# class\_conditional\_likelihoods = np.array([multivariate\_normal.pdf(data[3], mu[l], Sigma[l]) for l in L])

class\_conditional\_likelihoods = np.array([0.5\*multivariate\_normal.pdf(data[3][:, 1:], mu0[0], sigma0[0])+0.5\*multivariate\_normal.pdf(data[3][:, 1:], mu0[1], sigma0[1]),\

    multivariate\_normal.pdf(data[3][:, 1:], mu1[0], sigma1[0])])

discriminant\_score\_erm = np.log(class\_conditional\_likelihoods[1]) - np.log(class\_conditional\_likelihoods[0])

decisions\_map = discriminant\_score\_erm >= np.log(gamma\_map)

# Get indices and probability estimates of the four decision scenarios:

# (true negative, false positive, false negative, true positive)

# True Negative Probability

ind\_00\_map = np.argwhere((decisions\_map==0) & (labels[3]==0))

p\_00\_map = len(ind\_00\_map) / Nl[0]

# False Positive Probability

ind\_10\_map = np.argwhere((decisions\_map==1) & (labels[3]==0))

p\_10\_map = len(ind\_10\_map) / Nl[0]

# False Negative Probability

ind\_01\_map = np.argwhere((decisions\_map==0) & (labels[3]==1))

p\_01\_map = len(ind\_01\_map) / Nl[1]

# True Positive Probability

ind\_11\_map = np.argwhere((decisions\_map==1) & (labels[3]==1))

p\_11\_map = len(ind\_11\_map) / Nl[1]

# Probability of error for MAP classifier, empirically estimated

prob\_error\_erm = np.array((p\_10\_map, p\_01\_map)).dot(Nl.T / N[3])

print("Confusion Matrix MAP (rows: Predicted class, columns: True class):")

conf\_mat = confusion\_matrix(decisions\_map, labels[3])

print(conf\_mat)

correct\_class\_samples = np.sum(np.diag(conf\_mat))

print("Total Number of Misclassified Samples (MAP): {:d}".format(N[3] - correct\_class\_samples))

# Construct the ROC for ERM by changing log(gamma)

roc\_erm, \_, bestGamma, p\_error\_erm, bestEmpGamma = estimate\_roc(discriminant\_score\_erm, labels[3],N[3])

roc\_map = np.array((p\_10\_map, p\_11\_map))

fig\_roc, ax\_roc = plt.subplots(figsize=(10, 10))

ax\_roc.plot(roc\_erm[0], roc\_erm[1])

ax\_roc.plot(roc\_map[0], roc\_map[1], 'rx', label="Minimum P(Error) MAP", markersize=16)

ax\_roc.plot(bestEmpGamma[0], bestEmpGamma[1], 'gx', mfc='none', label="Minimum P(Error) Empirical", markersize=16)

ax\_roc.legend()

ax\_roc.set\_xlabel(r"Probability of false alarm $P(D=1|L=0)$")

ax\_roc.set\_ylabel(r"Probability of correct decision $P(D=1|L=1)$")

plt.grid(True)

plt.title('ROC Curve')

plt.show()

fig\_roc;

print('Gamma MAP (Theoretical): ', gamma\_map)

print('Probability of Error(MAP): ', prob\_error\_erm)

print('Best Gamma (ERM): ', bestGamma)

print('Probability of Error(Empirical): ', p\_error\_erm)

print()

X = data[3][:, 1:]

fig\_disc\_grid, ax\_disc = plt.subplots(figsize=(10, 10));

# plt.ion() # Re-activate "interactive" mode

horizontal\_grid = np.linspace(np.floor(np.min(data[3][:,1])), np.ceil(np.max(data[3][:,1])), 100)

vertical\_grid = np.linspace(np.floor(np.min(data[3][:,2])), np.ceil(np.max(data[3][:,2])), 100)

ax\_disc.plot(X[ind\_00\_map, 0], X[ind\_00\_map, 1], 'og', label="Correct Class 0");

ax\_disc.plot(X[ind\_10\_map, 0], X[ind\_10\_map, 1], 'or', label="Incorrect Class 0");

ax\_disc.plot(X[ind\_01\_map, 0], X[ind\_01\_map, 1], '+r', label="Incorrect Class 1");

ax\_disc.plot(X[ind\_11\_map, 0], X[ind\_11\_map, 1], '+g', label="Correct Class 1");

ax\_disc.legend();

ax\_disc.set\_xlabel(r"$x\_1$");

ax\_disc.set\_ylabel(r"$x\_2$");

ax\_disc.set\_title("MAP Decisions (RED incorrect)");

fig\_disc\_grid.tight\_layout();

# Generate a grid of scores that spans the full range of data

[h, v] = np.meshgrid(horizontal\_grid, vertical\_grid)

# Flattening to feed vectorized matrix in pdf evaluation

gridxy = np.array([h.reshape(-1), v.reshape(-1)])

likelihood\_grid\_vals = np.array([0.5\*multivariate\_normal.pdf(gridxy.T, mu0[0], sigma0[0]) \

    + 0.5\*multivariate\_normal.pdf(gridxy.T, mu0[1], sigma0[1]), \

    multivariate\_normal.pdf(gridxy.T, mu1[0], sigma1[0])])

# Where a score of 0 indicates decision boundary level

# print(likelihood\_grid\_vals.shape)

discriminant\_score\_grid\_vals = np.log(likelihood\_grid\_vals[1]) - np.log(likelihood\_grid\_vals[0]) - np.log(gamma\_map)

# Contour plot of decision boundaries

discriminant\_score\_grid\_vals = np.array(discriminant\_score\_grid\_vals).reshape(100, 100)

equal\_levels = np.array((0.3, 0.6, 0.9))

min\_DSGV = np.min(discriminant\_score\_grid\_vals) \* equal\_levels[::-1]

max\_DSGV = np.max(discriminant\_score\_grid\_vals) \* equal\_levels

contour\_levels = min\_DSGV.tolist() + [0] + max\_DSGV.tolist()

cs = ax\_disc.contour(horizontal\_grid, vertical\_grid, discriminant\_score\_grid\_vals.tolist(), contour\_levels, colors='k')

ax\_disc.clabel(cs, fontsize=16, inline=1)

plt.show()

fig\_disc\_grid;

# display(fig\_disc\_grid)

######################################

###             Part 2             ###

######################################

# Use ML parameter estimation to train from the 3 samples.

# Specify as min of negative log likeliehood (NLL)

# Use favorite optimization approach (GD, optimize.minimize in scipy)

# Determine how to use class-label-approx to classify sample to approx min P-error

# Apply to test data, estimate prob(Error) (use counts of decisions on validation set)

# (OPTIONAL: Generate plots of decision boundaries)

# PART B

# Repeat for logistic-quadratic-funtion-based approx of class label posterior functions given a sample

# Compare performancesof classifiers from part B to A.

# Starting point from to search for optimal parameters

theta0\_linear = np.random.randn(n+1)

theta0\_quadratic = np.random.randn(n+3+1)

fig\_decision, ax\_decision = plt.subplots(3, 2, figsize=(15, 15));

print("Training the logistic-linear model with GD per data subset...")

poly\_type = 'L'

for i in range(len(N)-1):

    shuffled\_indices = np.random.permutation(N[i])

    # Shuffle row-wise X (i.e. across training examples) and labels using same permuted order

    Xshuf = data[i][shuffled\_indices]

    yshuf = labels[i][shuffled\_indices]

    X\_quad = quadratic\_transformation(Xshuf)

    if poly\_type=='Q':

        theta\_gd, trace = gradient\_descent(log\_reg\_loss, theta0\_quadratic, X\_quad, yshuf, N[i], \*\*opts)

    else:

        theta\_gd, trace = gradient\_descent(log\_reg\_loss, theta0\_linear, Xshuf, yshuf, N[i], \*\*opts)

    print("Logistic-Linear N={} GD Theta: {}".format(N[i], theta\_gd))

    print("Logistic-Linear N={} NLL: {}".format(N[i], trace['loss'][-1]))

    # Convert our trace of parameter and loss function values into NumPy "history" arrays:

    theta\_hist = np.asarray(trace['theta'])

    nll\_hist = np.array(trace['loss'])

    if poly\_type=='Q':

        plot\_decision\_boundaries(X\_quad, yshuf, theta\_gd, ax\_decision[i, 0], poly\_type)

    else:

        plot\_decision\_boundaries(Xshuf, yshuf, theta\_gd, ax\_decision[i, 0], poly\_type)

    ax\_decision[i, 0].set\_title("Decision Boundary for \n Logistic-Linear Model N={}".format(Xshuf.shape[0]))

    # Linear: use validation data (10k samples) and make decisions in report results routine

    X\_valid\_quad = quadratic\_transformation(data[3])

    if poly\_type=='Q':

        report\_logistic\_classifier\_results(X\_valid\_quad, theta\_gd, labels[3], Nl, ax\_decision[i, 1], poly\_type)

    else:

        report\_logistic\_classifier\_results(data[3], theta\_gd, labels[3], Nl, ax\_decision[i, 1], poly\_type)

    ax\_decision[i, 1].set\_title("Classifier Decisions on Validation Set \n Logistic-Linear Model N={}".format(N[i]))

x1\_valid\_lim = (floor(np.min(data[3][:,1])), ceil(np.max(data[3][:,1])))

x2\_valid\_lim = (floor(np.min(data[3][:,2])), ceil(np.max(data[3][:,2])))

# Again use the most sampled subset (validation) to define x-y limits

plt.setp(ax\_decision, xlim=x1\_valid\_lim, ylim=x2\_valid\_lim)

# Adjust subplot positions

plt.subplots\_adjust(left=0.05,

                    bottom=0.05,

                    right=0.6,

                    top=0.95,

                    wspace=0.1,

                    hspace=0.3)

# Super plot the legends

handles, labels = ax\_decision[0, 1].get\_legend\_handles\_labels()

fig\_decision.legend(handles, labels, loc='lower center')

plt.show()

**PROBLEM2**

import matplotlib.pyplot as plt # For general plotting

import numpy as np

from scipy.stats import multivariate\_normal # MVN not univariate

from sklearn.metrics import confusion\_matrix

from modules import prob\_utils

from homework2 import hw2q2

from math import ceil, floor

from sklearn.preprocessing import PolynomialFeatures

np.set\_printoptions(suppress=True)

np.random.seed(7)      # seed 7 is really bad for quadratic

plt.rc('font', size=18)          # controls default text sizes

plt.rc('axes', titlesize=18)     # fontsize of the axes title

plt.rc('axes', labelsize=18)     # fontsize of the x and y labels

plt.rc('xtick', labelsize=14)    # fontsize of the tick labels

plt.rc('ytick', labelsize=14)    # fontsize of the tick labels

plt.rc('legend', fontsize=16)    # legend fontsize

plt.rc('figure', titlesize=18)   # fontsize of the figure title

# Breaks the matrix X and vector y into batches

def batchify(X, y, batch\_size, N):

    X\_batch = []

    y\_batch = []

    # Iterate over N in batch\_size steps, last batch may be < batch\_size

    for i in range(0, N, batch\_size):

        nxt = min(i + batch\_size, N + 1)

        X\_batch.append(X[i:nxt, :])

        y\_batch.append(y[i:nxt])

    return X\_batch, y\_batch

def gradient\_descent(loss\_func, theta0, X, y, N, \*args, \*\*kwargs):

    # Mini-batch GD. Stochastic GD if batch\_size=1.

    # Break up data into batches and work out gradient for each batch

    # Move parameters theta in that direction, scaled by the step size.

    # Options for total sweeps over data (max\_epochs),

    # and parameters, like learning rate and threshold.

    # Default options

    max\_epoch = kwargs['max\_epoch'] if 'max\_epoch' in kwargs else 200

    alpha = kwargs['alpha'] if 'alpha' in kwargs else 0.1

    epsilon = kwargs['tolerance'] if 'tolerance' in kwargs else 1e-6

    batch\_size = kwargs['batch\_size'] if 'batch\_size' in kwargs else 10

    # Turn the data into batches

    X\_batch, y\_batch = batchify(X, y, batch\_size, N)

    num\_batches = len(y\_batch)

    print("%d batches of size %d:" % (num\_batches, batch\_size))

    theta = theta0

    m\_t = np.zeros(theta.shape)

    trace = {}

    trace['loss'] = []

    trace['theta'] = []

    # Main loop:

    for epoch in range(1, max\_epoch + 1):

        # print("epoch %d\n" % epoch)

        loss\_epoch = 0

        for b in range(num\_batches):

            X\_b = X\_batch[b]

            y\_b = y\_batch[b]

            # print("epoch %d batch %d\n" % (epoch, b))

            # Compute NLL loss and gradient of NLL function

            loss, gradient = loss\_func(theta, X\_b, y\_b, \*args)

            loss\_epoch += loss

            # Steepest descent update

            theta = theta - alpha \* gradient

            # Terminating Condition is based on how close we are to minimum (gradient = 0)

            if np.linalg.norm(gradient) < epsilon:

                print("Gradient Descent has converged after {} epochs".format(epoch))

                break

        # Storing the history of the parameters and loss values per epoch

        trace['loss'].append(np.mean(loss\_epoch))

        trace['theta'].append(theta)

        # print(trace['loss'])

        # Also break epochs loop

        if np.linalg.norm(gradient) < epsilon:

            break

    return theta, trace

def cubic\_transformation(X):

    n = X.shape[1]

    phi\_X = X

    # Take all monic polynomials for a quadratic

    phi\_X = np.column\_stack((phi\_X, X[:, 1] \* X[:, 1],              X[:, 1] \* X[:, 2],              X[:, 2] \* X[:, 2],              \

                                    X[:, 1] \* X[:, 1] \* X[:, 1],    X[:, 1] \* X[:, 1] \* X[:, 2],    X[:, 1] \* X[:, 2] \* X[:, 2],    \

                                    X[:, 2] \* X[:, 2] \* X[:, 2]

                                    ))

    return phi\_X

def plot3(a, b, c, name="Training", mark="o", col="y"):

    # Adjusts the aspect ratio and enlarges the figure (text does not enlarge)

    fig = plt.figure()

    ax = fig.add\_subplot(111, projection='3d')

    ax.scatter(a, b, c, marker=mark, color=col)

    ax.set\_xlabel(r"$x\_1$")

    ax.set\_ylabel(r"$x\_2$")

    ax.set\_zlabel(r"$y$")

    plt.title("{} Dataset".format(name))

    # To set the axes equal for a 3D plot

    # ax.set\_prop\_cycle(color=['red', 'green', 'blue'])

    # ax.set\_box\_aspect((np.ptp(a), np.ptp(b), np.ptp(c)))

    # plt.show()

def lin\_reg\_loss(theta, X, y):

    # Size of batch

    B = X.shape[0]

    # Linear regression model X \* theta

    predictions = X.dot(theta)

    # Residual error (X \* theta) - y

    error = predictions - y

    # Loss function is MSE

    # print(error)

    # loss\_f = np.mean(error \*\* 2)

    # loss\_f = 0.5\*error.T.dot(error)

    loss\_f = (X.dot(theta)-y).T.dot(X.dot(theta)-y)

    # print(loss\_f)

    # Partial derivative for GD, X^T \* ((X \* theta) - y)

    g = (1 / B) \* X.T.dot(error)

    # g = (X.T.dot(error) - X.T.dot(y))

    return loss\_f, g

def MAP\_gamma(X,y,gamma):

    theta = np.linalg.inv(X.T.dot(X)+gamma\*np.identity(X.shape[1])).dot(X.T.dot(y))

    return theta

def mean\_square\_err(X,y,theta):

    y\_predict = X.dot(theta) #+ noiseV

    ### MSE

    mse = np.mean((y - y\_predict)\*\*2)

    return mse

"""

cubic polynomial y = c(x,theta) + v

           where v = Gauss(0,sigma\*\*2)

x = [1,x1,x2,x1x1,x1x2,x2x2,x1x1x1,x1x1x2,x1x2x2,x2x2x2]

10 terms (including bias)

"""

# Options for mini-batch gradient descent

opts = {}

opts['max\_epoch'] = 100

opts['alpha'] = 1e-6

opts['tolerance'] = 1e-3

opts['batch\_size'] = 10

def main():

    # mu = np.array([[0,0,0,0,0,0,0,0,0,0]])

    mu = np.zeros(10)

    sigma2 = 1

    sigma = np.identity(10)\*sigma2

    mu = 0

    sigma = 1

    Ntrain = 100

    Nvalidate = 1000

    # xTrain= hw2q2.generateData(Ntrain)

    # xVal = hw2q2.generateData(Nval)

    xTrain, yTrain, xValidate, yValidate = hw2q2.hw2q2()

    noiseT = multivariate\_normal.rvs(mu,sigma,Ntrain)

    noiseV = multivariate\_normal.rvs(mu,sigma,Nvalidate)

    # shuffled\_indices = np.random.permutation(N[i])

    # # Shuffle row-wise X (i.e. across training examples) and labels using same permuted order

    # Xshuf = data[i][shuffled\_indices]

    # yshuf = labels[i][shuffled\_indices]

    xAugT = np.column\_stack((np.ones(Ntrain), xTrain))

    yAug = np.column\_stack((np.ones(Ntrain), yTrain))

    X3train = cubic\_transformation(xAugT) #+ noiseT

    xAugV = np.column\_stack((np.ones(Nvalidate), xValidate))

    X3validate = cubic\_transformation(xAugV) #+ noiseT

    # poly = PolynomialFeatures(3)

    # X3 = poly.fit\_transform(xTrain)

    # # print(xTrain[0])

    # print(X3[0])

    nCubic = X3train.shape[1]

    theta0 = np.random.randn(nCubic)

    # theta0 = np.zeros(nCubic)

    theta\_gd, trace = gradient\_descent(lin\_reg\_loss, theta0, X3train, yTrain, Ntrain, \*\*opts)

    theta\_MAP = MAP\_gamma(X3train,yTrain,0)

    #Results

    print('theta start:')

    print(theta0)

    print('theta MLE:')

    print(theta\_gd)

    print('theta MAP:')

    print(theta\_MAP)

    print()

    # print("Mini-batch GD Theta: ", theta\_gd)

    # print("MSE: ", trace['loss'][-1])

    ### Now compare to test data and calc accuracy using Mean Square Error ###

    mse\_gd = mean\_square\_err(X3validate,yValidate,theta\_gd)

    mse\_MAP = mean\_square\_err(X3validate,yValidate,theta\_MAP)

    print('MSE GD:', mse\_gd)

    print('MSE MAP:', mse\_MAP)

    # y = c + noise

    # X\_gd = multivariate\_normal.rvs(theta\_gd,sigma,Nvalidate) + noiseV

    # X\_MAP = multivariate\_normal.rvs(theta\_MAP,sigma,Nvalidate) + noiseV

    # print()

    y\_MAP = X3validate.dot(theta\_MAP) + noiseV

    y\_gd = X3validate.dot(theta\_gd) + noiseV

    fig = plt.figure()

    ax = fig.add\_subplot(111, projection='3d')

    ax.scatter(xValidate[:, 0], xValidate[:, 1], yValidate, marker='o', color='b', label='True Data')

    ax.scatter(X3validate[:, 1], X3validate[:, 2], y\_gd, marker='o', color='y', label='ML Estimate')

    ax.scatter(X3validate[:, 1], X3validate[:, 2], y\_MAP, marker='o', color='r', label='MAP Estimate')

    ax.set\_xlabel(r"$x\_1$")

    ax.set\_ylabel(r"$x\_2$")

    ax.set\_zlabel(r"$y$")

    ax.legend()

    plt.show()

    #####################################

    ###         Varying gamma         ###

    #####################################

    trials = 10001

    gamma = np.linspace(0.0001,1000,trials)

    # print(gamma)

    mse\_range = []

    for i in range(trials):

        theta\_temp = MAP\_gamma(X3train,yTrain,gamma[i])

        mse\_range.append(mean\_square\_err(X3validate,yValidate,theta\_temp))

    plt.plot(gamma,mse\_range)

    plt.title("MSE vs gamma")

    plt.xlabel('gamma')

    plt.ylabel('MSE')

    # plt.xscale('log')

    # plt.yscale('log')

    plt.show()

    return

if \_\_name\_\_ == '\_\_main\_\_':

    main()