Q4. Implement logistic regression (LOR), LOR with L2-norm regularization, and LOR with L1-norm regularization models using BGD, SGD, and MBGD algorithms. The dataset in data_q4_q5.xlsx contains 30 features and one output. The class label ‘M’ stands for malignant, and ‘B’ is the Benign class. You must use hold-out cross-validation ((CV) with 70% as training, 10% as validation and 20% as testing) to evaluate training, validation, and testing instances for each model. Evaluate the performance of each model using accuracy, sensitivity, and specificity measures.

```
import pandas as pd
import math
import numpy as np
import matplotlib.pyplot as plt
```

```
from google.colab import drive, files
uploaded = files.upload()
```

Choose Files data_q4_q5.xlsx

- **data_q4_q5.xlsx**(application/vnd.openxmlformats-officedocument.spreadsheetml.sheet) - 108274 bytes, last modified: 10/1/2021 - 100% done

Saving data_q4_q5.xlsx to data_q4_q5 (1).xlsx

```
def sigmoid(z):
  z = z.astype(float)
  z_out = 1/(1 + np.exp(-z))
  return z_out
```

```
def hypothesis(X, wt):
  hyp = sigmoid(np.dot(X, wt.T))
  return hyp
```

```
# function for normalising data
def norm(data):
  # norm_data = data
  mean = np.mean(data, axis = 0)
  std = np.std(data, axis = 0)
  norm_data = (data-mean)/std
  return norm_data
```

```
# function for regularisation
def wt_regularisation(lamb, wt, reg):
  wt_reg = np.zeros(wt.shape)
  if reg == 0:
    wt_reg = 0
  elif reg == 1:
    wt_reg = (lamb/2)*np.sign(wt)
  elif reg == 2:
    wt_reg = lamb*wt
  return wt_reg
```

```
def wt_update(alpha, lamb, reg, X, y, wt):
  wt = wt + (alpha/len(y))*(np.dot(hypothesis(X, wt)-y, X) - wt_regularisation(reg, lamb, 
  return wt


def bgd(alpha, lamb, iters, X, y, reg):
  w = np.zeros(X.shape[1], dtype=np.longfloat)
  for i in range(iters):
    hyp = hypothesis(X, wt)
    w = w - (alpha/len(y))*(np.dot(hyp - y, X) - wt_regularisation(lamb, w, reg))
    # w = w*(1 - (alpha/len(y))*lamb) - (alpha/len(y))*np.dot(hyp - y, X)
  return w


def mbgd(alpha, lamb, iters, batch_size, X, y, reg):
  w = np.random.rand(X.shape[1])
  for i in range(iters):
    rand_ind = np.random.randint(len(y))
    X_batch = X[rand_ind:rand_ind + batch_size]
    y_batch = y[rand_ind:rand_ind + batch_size]
    hyp = hypothesis(X_batch, wt)
    w = w - (alpha/len(y))*(np.dot(hyp-y_batch, X_batch) - wt_regularisation(lamb, w, reg)
  return w


def sgd(alpha, lamb, iters, X, y):
  w = np.random.rand(X.shape[1])
  for i in range(iters):
    rand_ind = np.random.randint(len(y))
    X_ind = X[rand_ind:rand_ind + 1]
    y_ind = y[rand_ind:rand_ind + 1]
    hyp = hypothesis(X_ind, wt)
    # print(hyp.shape)
    # print(y_ind.shape)
    w = w - (alpha/len(y))*(np.dot(hyp - y_ind, X_ind) - wt_regularisation(lamb, w, reg))
  return w


def classification(X_ts, wt):
  m = X_ts.shape[0]
  y_sig = hypothesis(X_ts, wt)
  print(y_sig)
  y_pred = np.zeros(m)
  for i in range(m):
    if y_sig[i]>0.5:
      y_pred[i] = 2
    elif y_sig[i]<=0.5:
      y_pred[i] = 1
  return y_pred


def confusion_mat(y_predicted, y_testing):
  a, b, c, d = 0 , 0 , 0 , 0
  for i in range(len(y_testing)):
    if y_testing[i] == 1 :
```

```
      if y_predicted[i] == 1 :
        a += 1
      if y_predicted[i] == 2 :
        b += 1
    if y_testing[i] == 2 :
      if y_predicted[i] == 1 :
        c += 1
      if y_predicted[i] == 2 :
        d += 1
  acc = (a+d)/(a+b+c+d)
  sens = (10*a)/(a+b)
  spec = (d)/(d+c)
  return print(f"Sensitivity: {sens}\nSpecificity: {spec}\nAccuracy: {acc}")


# extracting the data and separating
data = pd.read_excel("data_q4_q5.xlsx")
data = np.asarray(data)


#·splitting·into·input·and·output
X·=·data[:,·:-1]··#input
y·=·data[:,-1]····#output
for·i·in·range(len(y)):
··if·y[i]·==·'M':
····y[i]·=·1
··elif·y[i]·==·'B':
····y[i]·=·2
#·print(y)

#·normalizing·X·and·y
X·=·X.astype(float)
X·=·norm(X)
# y = norm(y)
# print(X)
# print(y)


# splitting the data into training, testing and validation
rowsX = X.shape[0]
X_tr = X[0:int(rowsX*0.7)]
y_tr = y[0:int(rowsX*0.7)]

X_val = X[int(rowsX*0.7):int(rowsX*0.9)]
y_val = y[int(rowsX*0.7):int(rowsX*0.9)]

X_ts = X[int(rowsX*0.9):rowsX+1]
y_ts = y[int(rowsX*0.9):rowsX+1]

print(X_tr.shape, X_ts.shape, X_val.shape)
print(y_tr.shape, y_ts.shape, y_val.shape)



# defining X for regression model
m = X_tr.shape[0]
one_tr = np.ones([m,1])
```

```
one_tr = np.ones([m,1])
X_tr = np.append(one_tr, X_tr, axis = 1)

m = X_val.shape[0]
one_val = np.ones([m,1])
X_val = np.append(one_val, X_val, axis = 1)

m = X_ts.shape[0]
one_ts = np.ones([m,1])
X_ts = np.append(one_ts, X_ts, axis = 1)

print(X_tr.shape, X_ts.shape, X_val.shape)
```

```
(398, 30) (57, 30) (114, 30)
(398,) (57,) (114,)
(398, 31) (57, 31) (114, 31)
```

```
reg = 2

# defining hyperparameters for BGD
alpha = 0.005
iters = 1000
lamb = 0.01
# BGD results
wt_bgd = bgd(alpha, lamb, iters, X_tr, y_tr, reg)
print(wt_bgd)

# testing the algorithm
# m = X_ts.shape[0]
# one_ts = np.ones([m,1])
# X_ts = np.append(one_ts, X_ts, 1)

y_predsig = hypothesis(X_ts, wt_bgd)
y_pred = np.zeros(m)
print(y_pred)
for i in range(m):
  if y_predsig[i]>0.5:
    y_pred[i] = 2
  elif y_predsig[i]<=0.5:
    y_pred[i] = 1
print(y_ts, '\n\n', y_pred)

confusion_mat(y_pred, y_ts)
```

```
[4.881676469350999295 0.102313425657281870306 -0.5841711604445999801
 0.130595216902323335824 0.116921034835298803 0.3218977602910331789
 0.509028489886085 6108 0.579614958931286 5682 0.43011014870128 74023
 0.605252267091176 9203 0.468495283819074 15637 0.278224603214925 2738
 -0.067065500075341523356 0.265730890921450 02368 0.177126197470701 2068
 0.163510155396306 2271 0.729230876069013 0076 0.765997182790356 04443
 0.594931255694791 61533 0.706363526189722 633 0.760270581336452 4663
 0.150413425886012 71704 -0.544036761417745 79017 0.162685629939692 52253
 0.171771007863697 79854 0.144555576899021 75423 0.425294257271654 59943
 0.456429771830421 09319 0.354982846767921 3155 0.541914553326897 965
 0.42123709179011154072]
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1 2 1 2 1 1 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 1 2 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 2]

 [2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 1. 2. 2. 2. 2. 1. 2. 2. 2. 2. 1. 2. 2. 2.
 2. 2. 1. 2. 2. 2. 1. 1. 2. 1. 1. 2. 1. 1. 1. 2. 1. 2. 1. 2. 1. 1. 2. 2.
 2. 1. 2. 2. 2. 2. 2. 2. 1.]
Sensitivity: 0.0
Specificity: 0.6046511627906976
Accuracy: 0.45614035087719296
```

```
# BGD with L1
alpha_vals = np.linspace(0.001, 1, 20)
l_vals = np.linspace(0.001, 1, 20)
l_vals = np.logspace(-3,0,num=20)
err_bgd = 1000000
a_opt = 0
l_opt = 0
for a in alpha_vals:
  l = 0.001
  wt = bgd(a, l, 100, X_tr, y_tr, 1)
  # print(wt)
  temp_err = (0.5/len(y_val))*np.sum((y_val - hypothesis(X_val, wt))**2)
  # print(wt)
  # print(mse_err)
  if temp_err < err_bgd:
    # wt_bgd1 = wt
    a_opt = a
    err_bgd = temp_err
    # print(wt_bgd, '\n\n')

for l in l_vals:
  wt = bgd(a_opt, l, 100, X_tr, y_tr, 1)
  # print(wt)
  temp_err = (0.5/len(y_val))*np.sum((y_val - hypothesis(X_val, wt))**2)
  # print(wt)
  # print(mse_err)
  if temp_err < err_bgd:
    # wt_bgd1 = wt
    l_opt = l
    err_bgd = temp_err
    # print(wt_bgd, '\n\n')

wt_bgd1 = bgd(a_opt, l_opt, 500, X_tr, y_tr, 1)
y_pred = classification(X_ts, wt_bgd1)

print('BGD LOR with L1 regularisation')
confusion_mat(y_pred, y_ts)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: overf
  This is separate from the ipykernel package so we can avoid doing imports until
[1.00000000e+00 9.99999423e-01 6.56390441e-01 9.99485301e-01
 1.00000000e+00 1.00000000e+00 1.00000000e+00 1.00000000e+00
 1.00000000e+00 1.00000000e+00 2.73157338e-13 1.00000000e+00
```

```
 3.64646682e-01 9.99996253e-01 9.99997937e-01 2.35736354e-03
 1.00000000e+00 9.99983860e-01 9.99551627e-01 9.97636541e-01
 4.27512424e-05 1.00000000e+00 9.99957824e-01 1.00000000e+00
 1.00000000e+00 1.00000000e+00 4.09188036e-05 1.00000000e+00
 1.00000000e+00 1.00000000e+00 6.72621483e-04 6.28985042e-12
 9.65435877e-01 7.89733554e-04 3.17898153e-07 9.99999968e-01
 2.26859349e-05 2.18444609e-03 6.40043111e-13 9.99999755e-01
 3.64539277e-14 9.81569702e-01 3.68778621e-08 4.78114927e-01
 3.51804388e-05 9.85512394e-14 1.00000000e+00 9.99472034e-01
 9.99512801e-01 4.73220924e-28 1.00000000e+00 1.00000000e+00
 1.00000000e+00 1.00000000e+00 9.99999730e-01 1.00000000e+00
 5.31211674e-18]
BGD LOR with L1 regularisation
Sensitivity: 10.0
Specificity: 0.4418604651162791
Accuracy: 0.5789473684210527
```

```
# BGD with L2
wt_bgd2 = np.random.rand(X_tr.shape[1])
alpha_vals = np.logspace(-4,-2,num=20)
l_vals = np.logspace(-3,0,num=20)
err_bgd = 1000000
n = 0
for a in alpha_vals:
  for l in l_vals:
    wt = bgd(a, l, 500, X_tr, y_tr, 2)
    # print(wt)
    temp_err = (0.5/len(y_val))*np.sum((y_val - hypothesis(X_val, wt))**2)
    # print(wt)
    # print(mse_err)
    if temp_err < err_bgd:
      wt_bgd2 = wt
      err_bgd = temp_err
      # print(wt_bgd, '\n\n')

print(wt_bgd2)
y_pred = classification(X_ts, wt_bgd2)
# print(y_ts,·'\n\n',·y_pred)·

print('BGD LOR with L2 regularisation')
confusion_mat(y_pred, y_ts)
```

```
BGD LOR with L2 regularisation
Sensitivity: 0.42857142857142855
Specificity: 1.0
Accuracy: 0.7894736842105263
```

```
# BGD without regularisation
wt_bgd = np.random.rand(X_tr.shape[1])
alpha_vals = np.linspace(0.001, 1, 20)
l_vals = np.linspace(0.001, 1, 20)
err_bgd = 100000
n = 0
for a in alpha_vals:
  for l in l_vals:
```

```
    wt = bgd(a, l, 100, X_tr, y_tr, 0)
    # print(wt)
    temp_err = (1/len(y_val))*np.sum((y_val - hypothesis(X_val, wt))**2)
    # print(wt)
    # print(mse_err)
    if temp_err < err_bgd:
      wt_bgd = wt
      err_bgd = temp_err
      # print(wt_bgd, '\n\n')

print(wt_bgd)
y_pred = classification(X_ts, wt_bgd2)
# print(y_ts, '\n\n', y_pred)

print('BGD·LOR·without·regularisation')
confusion_mat(y_pred, y_ts)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:3: RuntimeWarning: overf
  This is separate from the ipykernel package so we can avoid doing imports until
[5.593375343255326676 -1.2325053801043730356 -1.4785859926342731843
 -1.2508165625878236299 -1.1816373854857486689 -0.5525851437939096472
 -0.9941444440928419446 -1.0302591855910626928 -1.2084231381093733793
 -0.2747541325614866279 0.0508512044577675670018 -0.8404041180295345609
 -0.19868145378882985132 -0.85436538182538327574 -0.8585356788519363236
 0.12656621491945847183 -0.29099941199668072703 -0.1835819915483014736
 -0.50735354970188554363 0.537759417205986405 0.12356406699411703366
 -1.3011255551794154856 -1.5009404331439740732 -1.322195633243759133
 -1.20175262350530037 -0.82840873292714164305 -1.0325325024282088274
 -1.101768561325546826 -1.344519225242194263 -0.43869413054550190217
 -0.601201411615441203]
BGD LOR without regularisation
Sensitivity: 0.7142857142857143
Specificity: 1.0
Accuracy: 0.7719298245614035
```

✓ 0s completed at 10:28 PM