# ECEN 765 – 600

# PROGRAMMING ASSIGNMENT -2

**SUBMITTED BY:**

**SAMBANDH BHUSAN DHAL**

**UIN: 726006080**

Q1.

**1. Implement the perceptron algorithm for logistic regression** (30pts):

1. Use the data set with binary classes (`bclass-train` and `bclass-test`): Each row is one example with the first column as the class label $\{-1, 1\}$ and the rest as feature variables) You can replace the class label $-1$ by 0). You can find the data set from eCampus.
2. Run your logistic regression using the raw data, data that has been normalized to have unit $l_1$ norm, and data that has been normalized to have unit $l_2$ norm. Which seems to work best?
3. Plot error rates for the training, and test data as a function of iteration (both the raw predictions and the normalized predictions).
4. Repeat the previous tasks by modifying the perceptron algorithm to use the averaged weights instead of the original updated weights over the iterations. The averaging extension of the perceptron algorithm is to maintain all weight vectors during the iterations and the final weight vector is the accumulative average weight vector.
5. Compare two versions of the perceptron algorithms. Discuss the differences.

Ans 1. **Python code to run the perceptron algorithm:**

```
import os
from numpy import linalg as LA
import numpy as np
import matplotlib.pyplot as plt

#definition of sigmoid function
def sigmoid(inX):
  return 1.0/(1+np.exp(-1*inX))

#steps to preprocess data
def DataSet(norm):
  test=open("/home/downloads/assignment/2_lm_opt/bclass/bclass/bclass-train",'r')
  inputData=np.zeros((200,34))
  labels=np.zeros((200))
  for i in range(0,200):
    row=test.readline()
    intArray=list(map(float,row.split()))
    end=np.shape(intArray)[0]
    arr=np.asarray(intArray)
    inputFeature=arr[1:end]
    l1Norm=LA.norm(inputFeature,norm)
    normalizedFeature=inputFeature/l1Norm
    if norm!=0:
      inputData[i,:]=normalizedFeature
    else:
      inputData[i,:]=inputFeature
    labels[i]=arr[0]
  test.close()
  return inputData,labels
```

```
#definition of perceptron algorithm
def perceptron(inputData,labels,maxCycles):
    [m,n]=np.shape(inputData)
    weights=np.ones((n))
    for i in range(maxCycles):
        for k in range(0,m):
            row=inputData[k,:]
            x=np.matmul(row,weights)
            h=sigmoid(x)
            if h>0.5:
                h=1
            else:
                h=-1
            if int(labels[k])!=h:
                temp=np.array(np.transpose(row))
                temp1=labels[k]*np.array(temp)
                for z in range(34):
                    weights[z]=weights[z]+temp1[z]
        k=0

    return weights

#Running perceptron algorithm over raw data
inputData,labels=DataSet(0)
Iterations=[10,50,100,150,200,250,300,500,750,1000,2500,5000,7500,10000,20000,50000,75000,10000
0]
rawDataTestError=np.zeros((18))
for j in range(0,18):
    print("Iteration",Iterations[j])
    inputData,labels=DataSet(0)
    weights=perceptron(inputData,labels,Iterations[j])
    ftest=open("/home/downloads/assignment/2_lm_opt/bclass/bclass/bclass-train",'r')
    inputData=np.zeros((1,34))
    error=0
    for i in range(0,76):
        testInput=ftest.readline()
        intArray=list(map(float,testInput.split()))
        arr=np.asarray(intArray)
        end=np.shape(intArray)[0]
        inputFeature=arr[1:end]
        l2Norm=LA.norm(inputFeature,2)
        normalizedFeature=inputFeature
        testOutput=sigmoid(np.matmul(normalizedFeature,weights))
        if testOutput>=0.5:
            testOutput=1
        else:
            testOutput=-1
        if testOutput!=int(arr[0]):
```

```
        rawDataTestError[j]=rawDataTestError[j]+1
rawDataTestError=rawDataTestError/76
print(rawDataTestError)
```

Output:

```
Iteration 10
Iteration 50
Iteration 100
Iteration 150
Iteration 200
Iteration 250
Iteration 300
Iteration 500
Iteration 750
Iteration 1000
Iteration 2500
Iteration 5000
Iteration 7500
Iteration 10000
Iteration 20000
Iteration 50000
Iteration 75000
Iteration 100000
[ 0.44736842  0.27631579  0.15789474  0.19736842  0.19736842  0.15789474
  0.15789474  0.21052632  0.21052632  0.17105263  0.18421053  0.19736842
  0.23684211  0.19736842  0.19736842  0.19736842  0.25        0.26315789]
```

**#Running perceptron algorithm over test data with l1 normalization**

```
inputData,labels=DataSet(1)
Iterations=[10,50,100,150,200,250,300,500,750,1000,2500,5000,7500,10000,20000,50000,75000,10000
0]
l1DataTestError=np.zeros((18))
for j in range(0,18):
   inputData,labels=DataSet(1)
   weights=perceptron(inputData,labels,Iterations[j])
   ftest=open("/home/downloads/assignment/2_lm_opt/bclass/bclass/bclass-test",'r')
   inputData=np.zeros((1,34))
   error=0
   for i in range(0,76):
      testInput=ftest.readline()
      intArray=list(map(float,testInput.split()))
      arr=np.asarray(intArray)
      end=np.shape(intArray)[0]
      inputFeature=arr[1:end]
      l1Norm=LA.norm(inputFeature,1)
      normalizedFeature=inputFeature/l1Norm
      testOutput=sigmoid(np.matmul(normalizedFeature,weights))
      if testOutput>=0.5:
         testOutput=1
      else:
         testOutput=-1
```

```
        if testOutput!=int(arr[0]):
            l1DataTestError[j]=l1DataTestError[j]+1
l1DataTestError=l1DataTestError/76
print(l1DataTestError)
```

**Output:**

```
[ 0.19736842  0.22368421  0.19736842  0.21052632  0.19736842  0.19736842
  0.18421053  0.19736842  0.19736842  0.18421053  0.19736842  0.18421053
  0.17105263  0.23684211  0.18421053  0.18421053  0.18421053  0.18421053]
```

**#Running perceptron algorithm over training data with l1 normalization**

```
inputData,labels=DataSet(1)
Iterations=[10,50,100,150,200,250,300,500,750,1000,2500,5000,7500,10000,20000,50000,75000,10000
0]
#Iterations=[10,50,100,150,200,250,300,500,750,1000,2500,5000,7500,10000]
l1DataTrainError=np.zeros((18))
for j in range(0,18):
    inputData,labels=DataSet(1)
    weights=perceptron(inputData,labels,Iterations[j])
    ftest=open("/home/downloads/assignment/2_lm_opt/bclass/bclass/bclass-train",'r')
    inputData=np.zeros((1,34))
    error=0
    for i in range(0,200):
        testInput=ftest.readline()
        intArray=list(map(float,testInput.split()))
        arr=np.asarray(intArray)
        end=np.shape(intArray)[0]
        inputFeature=arr[1:end]
        l1Norm=LA.norm(inputFeature,1)
        normalizedFeature=inputFeature/l1Norm
        testOutput=sigmoid(np.matmul(normalizedFeature,weights))
        if testOutput>=0.5:
            testOutput=1
        else:
            testOutput=-1
        if testOutput!=int(arr[0]):
            l1DataTrainError[j]=l1DataTrainError[j]+1
l1DataTrainError=l1DataTrainError/200
print(l1DataTrainError)
```

**Output:**

```
[ 0.115  0.1    0.08   0.065  0.05   0.055  0.07   0.1    0.065  0.05   0.09
  0.06   0.055  0.09   0.06   0.05   0.055  0.065]
```

**#Running perceptron algorithm over testing data with l2 normalization**

```
inputData,labels=DataSet(2)
Iterations=[10,50,100,150,200,250,300,500,750,1000,2500,5000,7500,10000,20000,50000,75000,10000
0]
```

```
l2DataTestError=np.zeros((18))
for j in range(0,18):
   inputData,labels=DataSet(2)
   weights=perceptron(inputData,labels,Iterations[j])
   ftest=open(""/home/downloads/assignment/2_lm_opt/bclass/bclass/bclass-test",'r')
   inputData=np.zeros((1,34))
   error=0
   for i in range(0,76):
      testInput=ftest.readline()
      intArray=list(map(float,testInput.split()))
      arr=np.asarray(intArray)
      end=np.shape(intArray)[0]
      inputFeature=arr[1:end]
      l2Norm=LA.norm(inputFeature,2)
      normalizedFeature=inputFeature/l2Norm
      testOutput=sigmoid(np.matmul(normalizedFeature,weights))
      if testOutput>=0.5:
         testOutput=1
      else:
         testOutput=-1
      if testOutput!=int(arr[0]):
         l2DataTestError[j]=l2DataTestError[j]+1
l2DataTestError=l2DataTestError/76
print(l2DataTestError)
```

**Output:**

```
[ 0.23684211  0.19736842  0.22368421  0.18421053  0.22368421  0.19736842
  0.17105263  0.18421053  0.17105263  0.17105263  0.17105263  0.17105263
  0.19736842  0.17105263  0.19736842  0.18421053  0.18421053  0.18421053]
```

**#Running perceptron algorithm over training data with l2 normalization**

```
inputData,labels=DataSet(2)
Iterations=[10,50,100,150,200,250,300,500,750,1000,2500,5000,7500,10000,20000,50000,75000,10000
0]
#Iterations=[10,50,100,150,200,250,300,500,750,1000,2500,5000,7500,10000]
l2DataTrainError=np.zeros((18))
for j in range(0,18):
   inputData,labels=DataSet(2)
   weights=perceptron(inputData,labels,Iterations[j])
   ftest=open("/home/downloads/assignment/2_lm_opt/bclass/bclass/bclass-train",'r')
   inputData=np.zeros((1,34))
   error=0
   for i in range(0,200):
      testInput=ftest.readline()
      intArray=list(map(float,testInput.split()))
      arr=np.asarray(intArray)
      end=np.shape(intArray)[0]
```

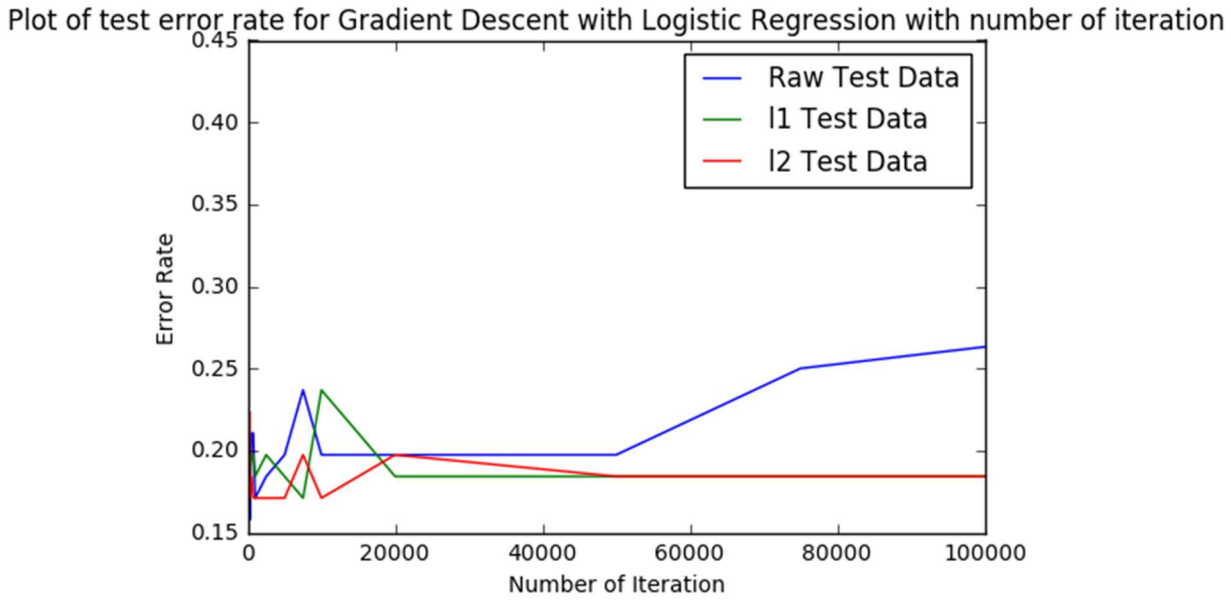
```
        inputFeature=arr[1:end]
        l2Norm=LA.norm(inputFeature,2)
        normalizedFeature=inputFeature/l2Norm
        testOutput=sigmoid(np.matmul(normalizedFeature,weights))
        if testOutput>=0.5:
            testOutput=1
        else:
            testOutput=-1
        if testOutput!=int(arr[0]):
            l2DataTrainError[j]=l2DataTrainError[j]+1
l2DataTrainError=l2DataTrainError/200
print(l2DataTrainError)
```

**Output:**

```
[ 0.135  0.09   0.085  0.065  0.085  0.06   0.055  0.06   0.055  0.055
  0.065  0.06   0.05   0.065  0.07   0.08   0.065  0.05 ]
```

**#Plotting testing error rates of raw data, data with l1 normalization and data with l2 normalization for perceptron algorithm**

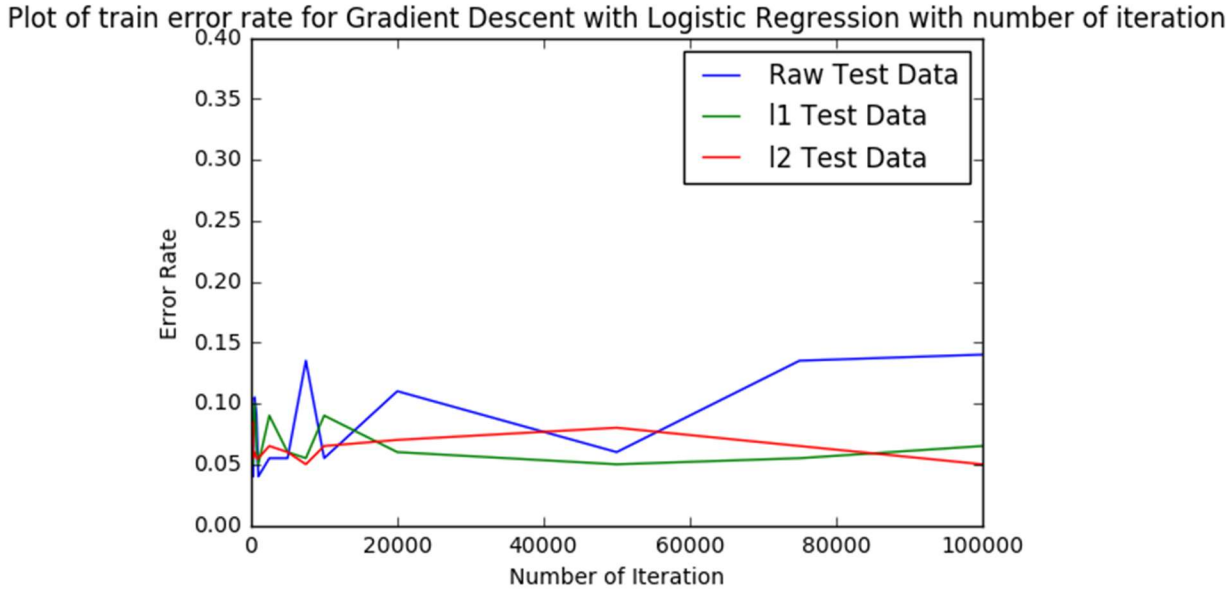
```
plt.plot(Iterations,l1DataTestError,label="l1 Test Data")
plt.plot(Iterations,l2DataTestError,label="l2 Test Data")
plt.ylabel('Error Rate')
plt.xlabel('Number of Iteration')
plt.title('Plot of test error rate for Gradient Descent with Logistic Regression with number of iteration')
plt.legend()
plt.show()
```



**The error rate for raw test data increases with the number of iterations where as for l1 and l2 test data, the error rate decreases and then becomes constant.**

**#Plotting training error rates of raw data, data with l1 normalization and data with l2 normalization for perceptron algorithm**

```
plt.plot(Iterations,rawDataTrainError,label="Raw Test Data")
plt.plot(Iterations,l1DataTrainError,label="l1 Test Data")
plt.plot(Iterations,l2DataTrainError,label="l2 Test Data")
plt.ylabel('Error Rate')
plt.xlabel('Number of Iteration')
plt.title('Plot of train error rate for Gradient Descent with Logistic Regression with number of iteration')
plt.legend()
plt.show()
```



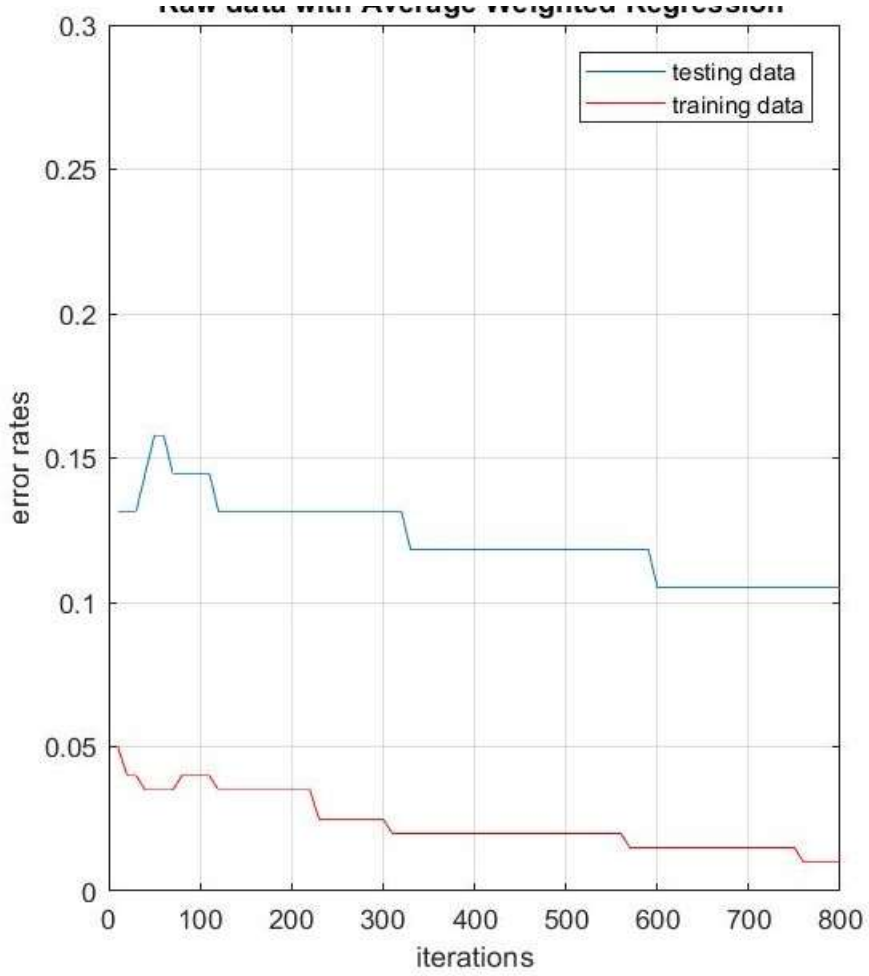

**The testing error rate for raw test data increases with the number of iterations. The logistic regression algorithm works best on l2 test data where the error rate decreases to less than 0.05 for over 100000 iterations.**

**#Running logistic regression with weighted perceptron**

```
def perceptron(inputData,labels,maxCycles):
    [m,n]=np.shape(inputData)
    weights=np.ones((n))
    for i in range(maxCycles):
        old_weights=weights
        for k in range(m):
            row=inputData[k,:]
            x=np.matmul(row,weights)
            h=sigmoid(x)
            if h>0.5:
```

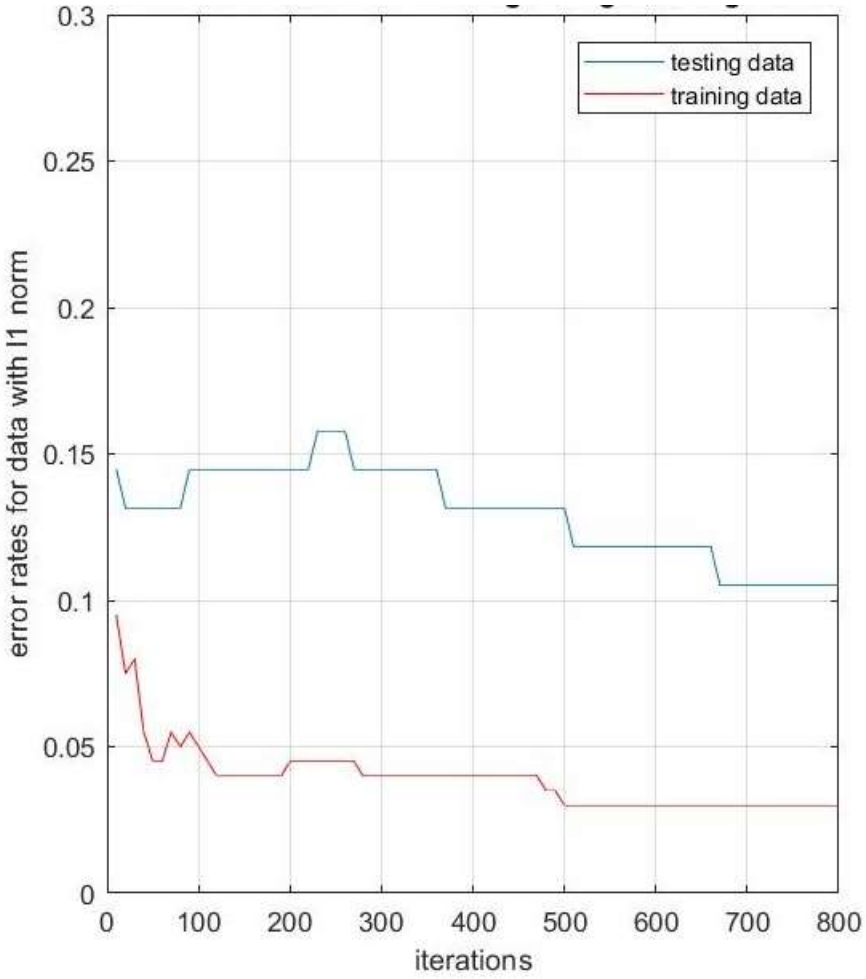
```
            h=1
         else:
            h=-1
         if int(labels[k])!=h:
            temp=np.array(np.transpose(row))
            temp1=labels[k]*np.array(temp)
            for z in range(34):
               weights[z]=weights[z]+temp1[z]
      if i>0:
         weights=(i*np.array(old_weights)+np.array(weights)/(i+1)

   return weights

inputData,labels=DataSet(0)
Iterations=[10,50,100,150,200,250,300,500,750,1000,2500,5000,7500,10000,20000,50000,75000,10000
0]
rawDataTestError=np.zeros((18))
for j in range(0,18):
   print("Iteration",Iterations[j])
   inputData,labels=DataSet(0)
   weights=perceptron(inputData,labels,Iterations[j])
   ftest=open("/home/downloads/assignment/2_lm_opt/bclass/bclass/bclass-test",'r')
   inputData=np.zeros((1,34))
   error=0
   for i in range(0,76):
      testInput=ftest.readline()
      intArray=list(map(float,testInput.split()))
      arr=np.asarray(intArray)
      end=np.shape(intArray)[0]
      inputFeature=arr[1:end]
      l2Norm=LA.norm(inputFeature,2)
      normalizedFeature=inputFeature
      testOutput=sigmoid(np.matmul(normalizedFeature,weights))
      if testOutput>=0.5:
         testOutput=1
      else:
         testOutput=-1
      if testOutput!=int(arr[0]):
         rawDataTestError[j]=rawDataTestError[j]+1
rawDataTestError=rawDataTestError/76
print(rawDataTestError)

#Plotting error rate for weighted logistic regression on raw data
plt.plot(Iterations,rawDataTestError,label="Testing Data")
plt.plot(Iterations,rawDataTrainError,label="Training Data")
plt.ylabel('Error Rate')
plt.xlabel('Iterations')
plt.title('Raw data with average weighted regression')
```

```
plt.legend()
plt.show()
```



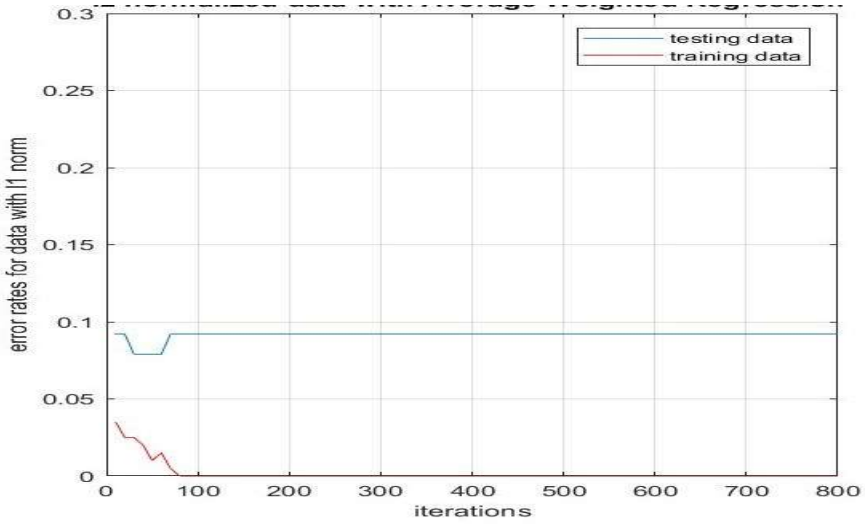

**#Plotting error rate for weighted logistic regression on l1 test data**

```
plt.plot(Iterations,l1DataTestError,label="testing data")
plt.plot(Iterations,l1DataTrainError,label="training data")
plt.ylabel('error rates for data with L1 norm')
plt.xlabel('Iterations')
plt.legend()
plt.show()
```




**#Plotting error rate for weighted logistic regression on l2 test data**

```
plt.plot(Iterations,l2DataTestError,label="testing data")
plt.plot(Iterations,l1DataTrainError,label="training data")
plt.ylabel('error rates for data with l1 norm')
plt.xlabel('Iterations')
plt.legend()
plt.show()
```




**In average perceptron, the weight vector is updated each time but the final weight vector which is used for testing the data is the mean of all the weight vectors used in the training data.**
**In vanilla perceptron, the weight vector is updated each time and the final updated vector is used to test the data.**
**The average perceptron algorithm works best on testing data with l2 norm compared to both l1 and raw data. The error rate is 0.083 for over 800 iterations.**
**Thus, we can conclude that the average perceptron algorithm works best on this set of data when compared to the vanilla perceptron algorithm.**

Q2. Implement the gradient descent/ascent algorithm for logistic regression (30pts):

1. Use the same data set as in 1. Run your logistic regression using the raw data, data that has been normalized to have unit l1 norm, and data that has been normalized to have unit l2 norm. Which seems to work best?

2. Plot error rates for the training, and test data as a function of iteration (both the raw predictions and the normalized predictions).

3. Compare the results with the ones from the perceptron algorithm. Which one is better? Why?

Ans:

1 . **Python Code to run gradient descent/ascent algorithm:**

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

```
from sklearn import metrics

#steps to pre-process data
df_train = pd.read_table('/home/downloads/assignment/2_lm_opt/bclass/bclass/bclass-train',delimiter='\t',header=None)
df_test = pd.read_table('/home/downloads/assignment/2_lm_opt/bclass/bclass/bclass-test',delimiter='\t',header=None)

#Converting -1 to 0
df_train.head()
df_train.loc[df_train[0] == -1,0] = 0
df_test.loc[df_test[0] == -1,0] = 0

#Splitting as feature vectors and label vectors
X_train = df_train.loc[:,1:]
y_train = df_train.loc[:,0]
X_test = df_test.loc[:,1:]
y_test = df_test.loc[:,0]

#implementing gradient descent function on the data
from sklearn import metrics, preprocessing
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import learning_curve
def logistic_func(theta,x):
   return float(1) / (1 + math.e**(-x.dot(theta)))
def log_gradient(theta, x, y):
   first_calc = logistic_func(theta, x) - np.squeeze(y)
   final_calc = first_calc.T.dot(x)
   return final_calc

def cost_func(theta, x, y):
   log_func_v = logistic_func(theta,x)
   y = np.squeeze(y)
   step1 = y * np.log(log_func_v)
   step2 = (1-y) * np.log(1 - log_func_v)
   final = -step1 - step2
   return np.mean(final)

def pred_values(theta, X, hard=True):
   pred_prob = logistic_func(theta, X)
   pred_value = np.where(pred_prob >= .5, 1, 0)
   if hard:
```

```
        return pred_value
    return pred_prob
def grad_desc(theta_values, X_train, y_train, X_test, y_test, lr=.001, converge_change=.00001,n_iter =
100000):

    cost_iter = []
    train_pred_acc = []
    test_pred_acc = []
    cost = cost_func(theta_values, X_train, y_train)
    train_acc = metrics.accuracy_score(y_train, pred_values(theta_values, X_train))
    test_acc = metrics.accuracy_score(y_test, pred_values(theta_values, X_test))

    cost_iter.append([0, cost])
    train_pred_acc.append([0,train_acc])
    test_pred_acc.append([0,test_acc])
    change_cost = 1
    i = 1

    while(change_cost > converge_change and i < n_iter):
        old_cost = cost
        theta_values = theta_values - (lr * log_gradient(theta_values, X_train, y_train))
        cost = cost_func(theta_values, X_train, y_train)
        train_acc = metrics.accuracy_score(y_train, pred_values(theta_values, X_train))
        test_acc = metrics.accuracy_score(y_test, pred_values(theta_values, X_test))

        cost_iter.append([i, cost])
        train_pred_acc.append([i, train_acc])
        test_pred_acc.append([i,test_acc])
        change_cost = old_cost - cost
        i+=1

return theta_values, np.array(cost_iter), np.array(train_pred_acc), np.array(test_pred_acc)

        X_train_m = X_train.as_matrix()
        y_train_m = y_train.as_matrix()
        X_test_m = X_test.as_matrix()
        y_test_m = y_test.as_matrix()

        shape = X_train_m.shape[1]
        betas = np.zeros(shape)
fitted_values, cost_iter, train_pred_accuracy,test_pred_accuracy = grad_desc(betas, X_train_m,
y_train_m,X_test_m, y_test_m)
```

2. **Python code to plot error rates for training and testing data as a function of iteration:**

**#Plotting error rates on training and testing data as a function of iteration on raw data**

```
fig = plt.figure(figsize=(20, 8))
ax1= fig.add_subplot(1,2,1)
plt.style.use('ggplot')
ax1.plot(cost_iter[:,0], cost_iter[:,1],linewidth=2.0)

ax1.set_ylabel("Cost")
ax1.set_xlabel("Iteration")
ax1.set_ylim(0,0.9)

ax2 = fig.add_subplot(1,2,2)
ax2.plot(train_pred_accuracy[:,0], train_pred_accuracy[:,1],linewidth=2.0,label="Training Accuracy")
ax2.plot(test_pred_accuracy[:,0], test_pred_accuracy[:,1],linewidth=2.0,label="Testing Accuracy")
ax2.set_ylabel("Accuracy")
ax2.set_xlabel("Iteration")
ax2.legend()
```

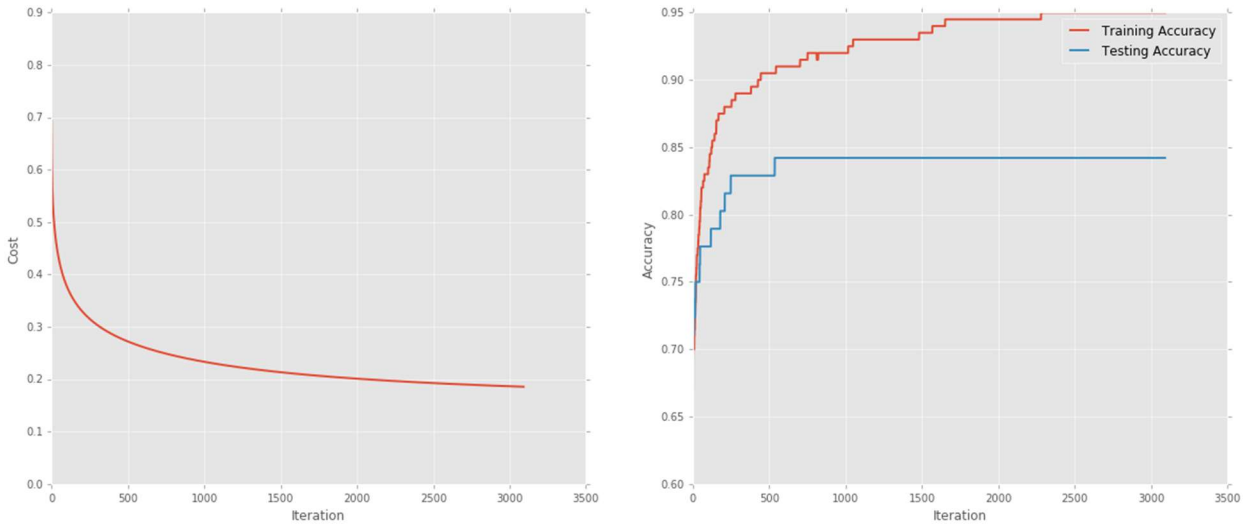



**Fig 1: Plot showing training and testing accuracy on raw data**
**The training accuracy was found to be 0.95 and the testing accuracy for the data was found to be 0.832**

**#Plotting error rates on training and testing data as a function of iteration on data normalized to have unit l1 norm**

```
X_train_norm_l1 = preprocessing.normalize(X_train_m, norm='l1')
```

```
X_test_norm_l1 = preprocessing.normalize(X_test_m, norm='l1')

shape = X_train_norm_l1.shape[1]
betas = np.zeros(shape)
fitted_values, cost_iter, train_pred_accuracy,test_pred_accuracy = grad_desc(betas, X_train_norm_l1,
y_train_m,X_test_norm_l1, y_test_m)

print train_pred_accuracy[-1,1]
print test_pred_accuracy[-1,1]

fig = plt.figure(figsize=(20, 8))
ax1= fig.add_subplot(1,2,1)
plt.style.use('ggplot')
ax1.plot(cost_iter[:,0], cost_iter[:,1],linewidth=2.0)

ax1.set_ylabel("Cost")
ax1.set_xlabel("Iteration")
ax1.set_ylim(0,0.9)

ax2 = fig.add_subplot(1,2,2)
ax2.plot(train_pred_accuracy[:,0], train_pred_accuracy[:,1],linewidth=2.0,label="Training Accuracy")
ax2.plot(test_pred_accuracy[:,0], test_pred_accuracy[:,1],linewidth=2.0,label="Testing Accuracy")
ax2.set_ylabel("Accuracy")
ax2.set_xlabel("Iteration")
ax2.legend()
```

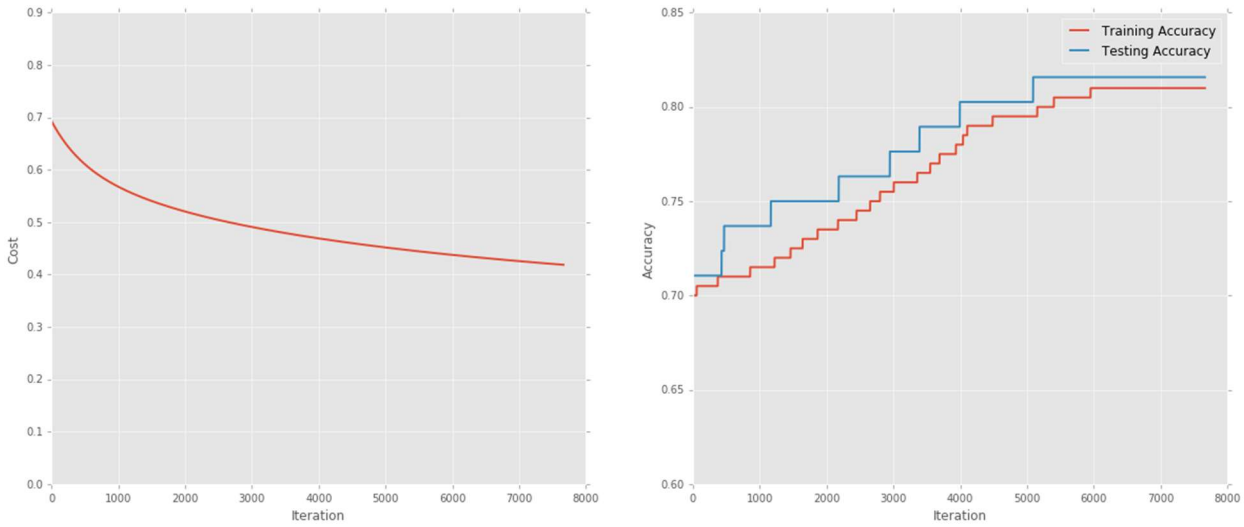



**Fig 2: Plot showing training and testing accuracy as a function of iteration on data normalized to have unit l1 norm**

**The training accuracy was found to be 0.81 and testing accuracy was found to be 0.815789473684**

**#Plotting error rates on training and testing data as a function of iteration on data normalized to have unit l2 norm**

```
X_train_norm_l2 = preprocessing.normalize(X_train_m, norm='l2')
X_test_norm_l2 = preprocessing.normalize(X_test_m, norm='l2')

shape = X_train_norm_l1.shape[1]
betas = np.zeros(shape)
fitted_values, cost_iter, train_pred_accuracy,test_pred_accuracy = grad_desc(betas, X_train_norm_l2,
y_train_m,X_test_norm_l2, y_test_m)

print train_pred_accuracy[-1,1]
print test_pred_accuracy[-1,1]

fig = plt.figure(figsize=(20, 8))
ax1= fig.add_subplot(1,2,1)
plt.style.use('ggplot')
ax1.plot(cost_iter[:,0], cost_iter[:,1],linewidth=2.0)

ax1.set_ylabel("Cost")
ax1.set_xlabel("Iteration")
ax1.set_ylim(0,0.9)

ax2 = fig.add_subplot(1,2,2)
ax2.plot(train_pred_accuracy[:,0], train_pred_accuracy[:,1],linewidth=2.0,label="Training Accuracy")
ax2.plot(test_pred_accuracy[:,0], test_pred_accuracy[:,1],linewidth=2.0,label="Testing Accuracy")
ax2.set_ylabel("Accuracy")
ax2.set_xlabel("Iteration")
ax2.legend()
```

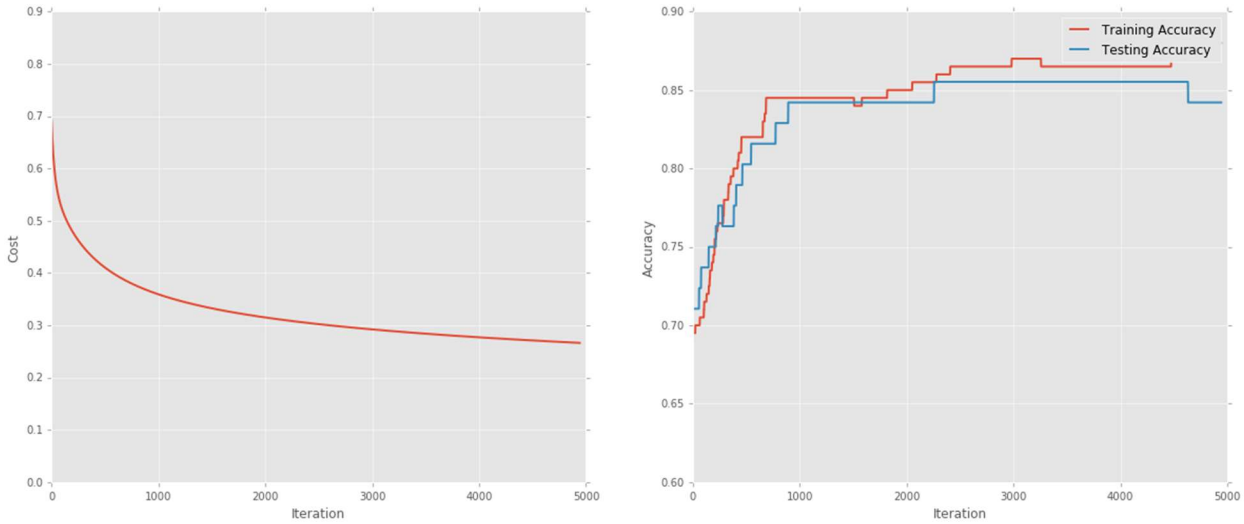



**Fig 2: Plot showing training and testing accuracy as a function of iteration on data normalized to have unit l2 norm**

**The training accuracy was found to be 0.88 and testing accuracy was found to be 0.842**

**Thus, from the inferences stated above we can conclude that the data normalized to have unit l2 norm has the highest accuracy ( accuracy of 0.86 after 2100 observations) when compared to l1 norm ( accuracy of 0.81) and the raw data.**

**The logistic regression algorithm using weighted perceptron method gives an accuracy of about 0.92 on l2 norm testing data where as, in case of gradient ascent/descent algorithm, the highest accuracy that we get is 0.86( after 2100 observations). Thus, we can conclude that the weighted perceptron method works better on this data set.**

Q3.

**3. Locally Weighted Logistic Regression** (40pts):

Implement a locally-weighted version of logistic regression, where we weight different training examples differently according to the query point. The locally weighted logistic regression problem is to maximize

$$l(\beta) = \sum_{i=1}^{N} w^i \Big\{ y^i \log f_\beta(x^i) + (1 - y^i) \log \left[1 - f_\beta(x^i)\right] \Big\} - \lambda \beta^T \beta,$$

where the last term is the regularization term as we discussed for the linear regression in class. (You can also implement the regularized logistic regression for **2**, which often can give more stable results using either gradient descent or Newton's method.) You can set $\lambda = 0.001$; Or you can use the development data included in the data set to pick the best performing $\lambda$:

1. Compute the gradient $\nabla_\beta l(\beta)$ and the Hessian matrix $H = \nabla_\beta [\nabla_\beta l(\beta)]$;
2. Given a new test data point $x$, we compute the weight by

$$w^i = \exp(-\frac{\| x - x^i \|_{l_2}^2}{2\tau^2}),$$

where $\tau$ is the bandwidth. Use the same data set with binary classes as above (`bclass-train` and `bclass-test`) to implement **Newton's** method for this locally weighted logistic regression. Vary $\tau = \{0.01, 0.05, 0.1, 0.5, 1.0, 5.0\}$ to: (a) compute $w^i$'s for each development/test sample using the formula above, (b) maximize $l(\beta)$ to learn $\beta$, (c) predict $y$ based on $f_\beta(x)$ ($y = 1$ when $f_\beta(x) \geq 0.5$), and finally (d) plot the error rates with respect to $\tau$ and compare them with the ones obtained in **1**.

Ans :

1 .**Python Code to run locally weighted logistic regression algorithm:**

```
import numpy as np
import pandas as pd

#steps to pre-process data
df_train = pd.read_table('/home/downloads/assignment/2_lm_opt/bclass/bclass/bclass-train',delimiter='\t',header=None)
df_test = pd.read_table('/home/downloads/assignment/2_lm_opt/bclass/bclass/bclass-test',delimiter='\t',header=None)

#Converting -1 to 0
df_train.head()
df_train.loc[df_train[0] == -1,0] = 0
df_test.loc[df_test[0] == -1,0] = 0
df_train = df_train.drop(2, 1) #CAREFUL
df_test = df_test.drop(2,1) #CAREFUL LABEL NAMES
```

**#Splitting data into features and label vectors**

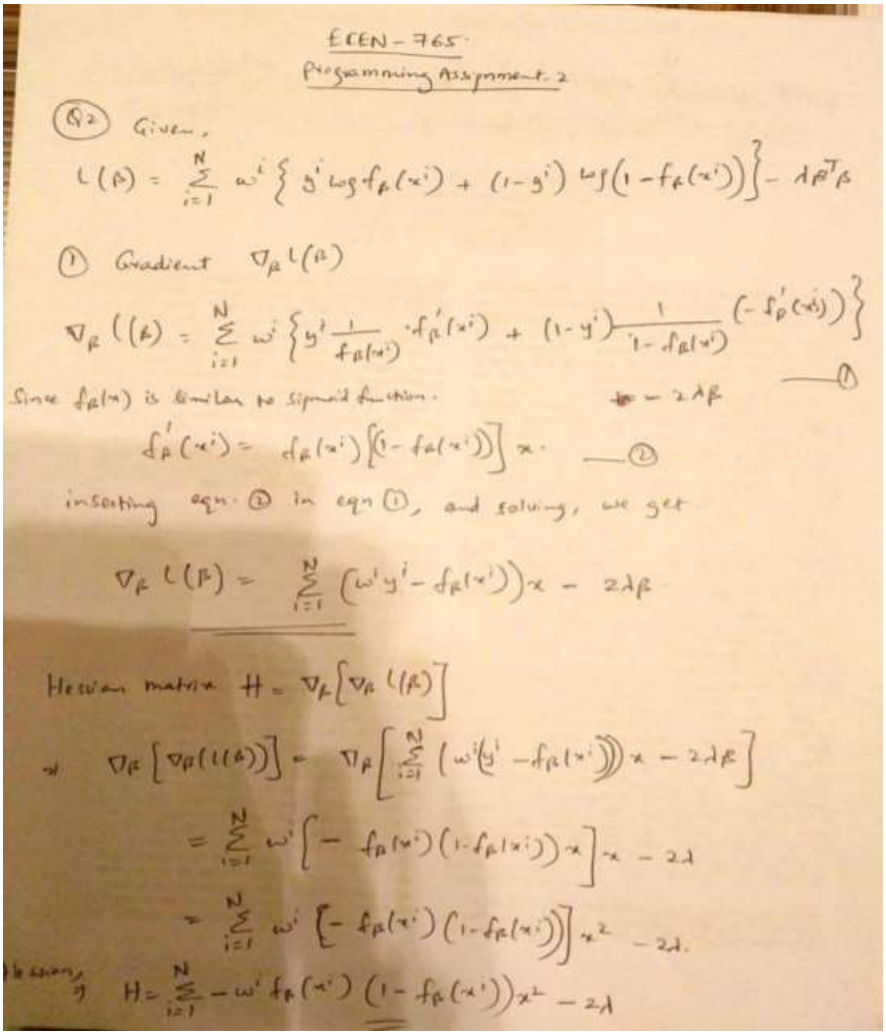
```
X_train = df_train.ix[:,1:]
y_train = df_train[0]
X_test = df_test.loc[:,1:]
y_test = df_test.loc[:,0]
```

**# Code to compute the gradient and Hessian matrix**

```
lambda_value = 0.001

def sigmoid(z):
   return 1 / (1 + np.e**(-z))

def hypothesis(theta, x):
   return sigmoid(np.dot(x, theta))

def gaussian(point1, point2, tau):
      return np.exp(- (np.linalg.norm(a-b)**2) / (2 * tau**2))

def weights(data, new_point, tau):
   assert (new_point.shape[0] == 1) #new point should be 1xn, where n is the number of features
   new_point_repeat = np.repeat(new_point, data.shape[0], axis = 0)
   norm_squared = np.linalg.norm(data - new_point_repeat, axis = 1) ** 2
   weights = np.exp(-norm_squared / (2 * tau ** 2))
   return weights.reshape(len(weights), 1)

def lw_logreg(x_train, y_train, x, tau):
   x_train_aug = pd.concat([pd.Series(np.ones(len(x_train.index))), x_train], axis = 1)
   x_aug = np.concatenate([np.asarray([[1]]), x], axis = 1)

   theta_length = x_train_aug.shape[1]
   theta = np.zeros([theta_length, 1])
   gradient = np.ones([33,1])

   dist = 1
   i = 0
   while dist > 0.0001 and i < 100:
      weights = getWeights(x_train_aug, x_aug, tau)
      y_train = y_train.reshape(200,1)
      z = weights * (y_train - hypothesis(theta, x_train_aug))
      gradient = np.dot(x_train_aug.transpose(), z) -  lambda_value * theta
```

```
    diags = -weights * hypothesis(theta, x_aug) * (1 - hypothesis(theta, x_aug))
    D = np.diag(diags[:, 0])
H = np.dot(np.dot(x_train_aug.transpose(), D), x_train_aug) - lambda_value *
np.eye(x_train_aug.shape[1]) #computes the Hessian matrix

    theta_new = theta - np.dot(np.linalg.inv(H), gradient)

    dist = np.linalg.norm(theta_new - theta)
    theta = theta_new
    i +=1

  return theta
```

**#Theoretical computation of Gradient and Hessian matrix**



EECEN - 765

Programming Assignment 2

(Q2) Given,

$$L(\beta) = \sum_{i=1}^{N} w^i \left\{ y^i \log f_\beta(x^i) + (1-y^i) \log\left(1 - f_\beta(x^i)\right) \right\} - \lambda \beta^T \beta$$

① Gradient $\nabla_\beta L(\beta)$

$$\nabla_\beta L(\beta) = \sum_{i=1}^{N} w^i \left\{ y^i \frac{1}{f_\beta(x^i)} \cdot f'_\beta(x^i) + (1-y^i) \frac{1}{1 - f_\beta(x^i)} \left(-f'_\beta(x^i)\right) \right\} \quad \text{---①}$$

Since $f_\beta(x)$ is similar to sigmoid function. $-2\lambda\beta$

$$f'_\beta(x^i) = f_\beta(x^i)\left[(1 - f_\beta(x^i))\right] x. \quad \text{---②}$$

inserting eqn ② in eqn ①, and solving, we get

$$\nabla_\beta L(\beta) = \sum_{i=1}^{N} \left(w^i y^i - f_\beta(x^i)\right) x - 2\lambda\beta$$

Hessian matrix $H = \nabla_\beta \left[\nabla_\beta L(\beta)\right]$

$$\Rightarrow \nabla_\beta \left[\nabla_\beta (L(\beta))\right] = \nabla_\beta \left[ \sum_{i=1}^{N} \left( w^i (y^i - f_\beta(x^i)) \right) x - 2\lambda\beta \right]$$

$$= \sum_{i=1}^{N} w^i \left[ - f_\beta(x^i)\left(1 - f_\beta(x^i)\right) x \right] x - 2\lambda$$

$$= \sum_{i=1}^{N} w^i \left[ - f_\beta(x^i)\left(1 - f_\beta(x^i)\right) \right] x^2 - 2\lambda.$$

Hessian, $H = \sum_{i=1}^{N} - w^i f_\beta(x^i)\left(1 - f_\beta(x^i)\right) x^2 - 2\lambda$

**Python code to implement training accuracy, testing accuracy, training and testing error rate of the Locally weighted regression algorithm**

```
from sklearn import metrics

# code to find training accuracy
pred = []
train_accs = []
taus = [0.01, 0.05, 0.1, 1.0, 5.0]
for tau in taus:
    pred = []
    for i in xrange(200):
        x = X_train.iloc[[i]]
        final_theta = lw_logreg(X_train, y_train, x, tau)
        x_aug = np.concatenate([np.asarray([[1]]), x], axis = 1)

        if hypothesis(final_theta,x_aug) > 0.5:
            pred.append(1)
        else:
            pred.append(0)

    train_acc = metrics.accuracy_score(y_train,pred)
    print train_acc
    train_accs.append(train_acc)
```

Output:

```
1.0         1.0         1.0      0.665       0.665
```

**# code to find testing accuracy**

```
pred = []
test_accs = []
taus = [0.01, 0.05, 0.1, 1.0, 5.0]
for tau in taus:
    pred = []
    for i in xrange(X_test.shape[0]):
        x = X_test.iloc[[i]]
        final_theta = lw_logreg(X_train, y_train, x, tau)
        x_aug = np.concatenate([np.asarray([[1]]), x], axis = 1)

        if hypothesis(final_theta,x_aug) > 0.5:
            pred.append(1)
        else:
```

```
        pred.append(0)

    test_acc = metrics.accuracy_score(y_test,pred)
    print test_acc
    test_accs.append(test_acc)
```

Output:

```
0.328947368421 0.552631578947 0.815789473684 0.565789473684 0.723684210526
```

**#code to find the training and testing error**

```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(8, 8))
plt.style.use('ggplot')

width = 0.35
ind = np.arange(5)

ax1 = fig.add_subplot(1,1,1)
rects1 = ax1.bar(ind, train_accs, width, color='#fc8d62',label='Training')
ax1.set_ylabel("Accuracy")
ax1.set_xlabel("Tau Values")
ax1.set_xticks(ind+width)
ax1.set_xticklabels(('0.01', '0.05', '0.1', '1', '5'))
ax1.set_title("Training vs Testing errors")

rects2 = ax1.bar(ind+width, test_accs, width, color='#66c2a5',label='Testing')
ax1.legend()
plt.show()
```

## Interpretation:

### Training Error:

**For smaller tau values, the locally weighted logistic regression classifier is overfitting the data. Hence, it gets 100% accuracy on the training data set. In such a case, we can expect the decision boundary to be highly complex. As tau value increases, the classifier is able to generalize better and the decision boundary becomes much smoother. Hence, we clearly see a drop in the accuracy with the training data set**

### Testing Error:

**As expected, we see that the testing accuracy when tau = 0.01 (the case when overfitting occured) is extremely low. After that, as the boundary becomes smoother, the accuracy on the test data set begins to increase. The dip maybe because of this particular data set but we could do such tests to determine which tau value is the best for a particular data set.**

**Upon comparing error plots from Locally Weighted Logistic Regression and the previous plots from question 1, it can be concluded that as the value of Tau increases the error rates also increase for both training and test data. In comparison with Q1, at low values of Tau, the error rate performance is better in Locally Weighted Logistic Regression. At higher values of Tau, L-2 and Raw Data are giving better error performance.**