Implementing Logistic regression with delta learning rule using Newton's method and compare the results with LOG(using gradient descent)

Solution:

Step 1. Reading the dataset:

I am using pandas for reading the dataset and further classifying them into training and test set.

```
df = pd.read_csv('data_exam.csv')
```

```
X = df.iloc[:,0:2]
Y = df.iloc[:,-1]
```

```
df.head()
```

	X	Y	label
0	34.623660	78.024693	0
1	30.286711	43.894998	0
2	35.847409	72.902198	0
3	60.182599	86.308552	1
4	79.032736	75.344376	1

Data Normalization:

I have used mean shifting and variance scaling for normalising my dataset. After normalising the data, a bias term has been added to the data.

```
X = (X - np.mean(X))/np.std(X)
```

```
X.insert(loc = 0,column = 'bias',value=np.ones(X.shape[0]))
```

Splitting into training set and test set:

For testing the model performance, I am training my model on 70 % of the data and it's performance has been tested on rest of the 30 % of the data.

```
X_train, X_test,Y_train,Y_test = train_test_split(X,Y,test_size = 0.3)
```

Weight Initialisation:

Weights has been initialised from a normal distribution of mean 0 and variance 1.

```
w = np.random.normal(0,1,X.shape[1])
```

Creating the Model.

Sigmoid function:

Sigmoid function is used for converting any value in the range of 0 and 1. It is also known as activation function for the logistic regression. It can be defined as follows.

```
def sigmoid(x):
    return 1/(1 + np.exp(-x))
```

Also for logistic regression we use log loss function for calculating the cost and we minimize this log loss function by using gradient Descent. Loss can be defined as

```
def loss(y,hx):
    return ((-y * np.log(hx)) - (1-y) * np.log(1-hx)).mean()
```

Newton's method:
In constrast to classical gradient descent algorithm, newton's method tries to find the minima of a function by taking variable sized step towards minima of a function. Update rules for newton's method is given as:

$$\theta^{(t+1)} = \theta^{(t)} - H^{-1}\nabla_\theta J$$

In logistic regression, the gradient and hessian are given as:

$$\nabla_\theta J = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x^{(i)}$$

$$H = \frac{1}{m}\sum_{i=1}^{m}\left[h_\theta(x^{(i)})\left(1 - h_\theta(x^{(i)})\right)x^{(i)}\left(x^{(i)}\right)^T\right]$$

The above formula is written in the vectorised form. My implementation of the above approach in python is as follows:

```
def newton(w,epochs):
    weights = []
    errors = []
    w = w.reshape(-1,1)
    for i in range(epochs):
        pred = np.dot(X_train,w)
        print(pred.shape)
        h = sigmoid(pred)
        #h = h.reshape(-1,1)
        a = h*(1-h)*X_train
        error = loss(y_train,h)
        grad = np.dot(X_train.T,h- y_train)
        hess = a.T @ X_train
        weights.append(w)
        errors.append(error)
        w = w - np.matmul(np.linalg.pinv(hess),grad);
    return weights,errors
```

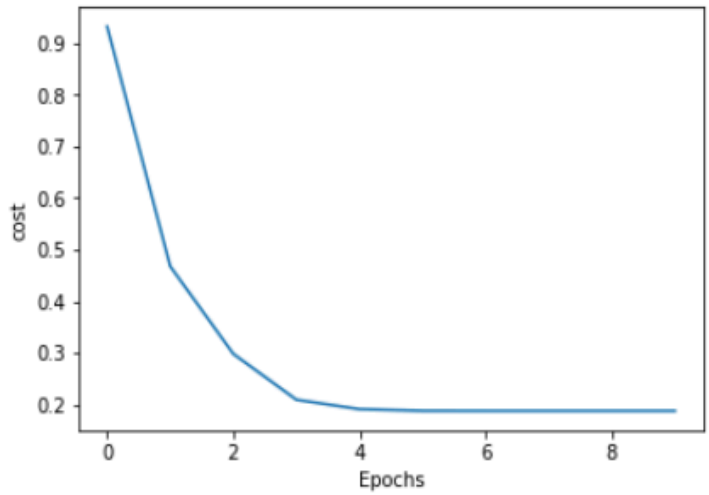
The function newton can be called as:

```
weights,error = newton(w,10)
```

At every iteration, value of weight is stored in weights[] list. We can use last value of weight for our prediction.

Cost vs. Epochs plot:

```
plt.plot(error)
plt.xlabel('Epochs')
plt.ylabel('cost')
plt.show()
```



Observation: It can be seen that np. of epochs required for convergence using newton's method is quite less. It converged in only 4 iterations.

Predicting Values:

By taking the sigmoid of the dot product, we can calculate the predicted value.

```
def pred(data):
    return sigmoid(np.dot(data,weights))
```

```
a = pred(X_test)
```

```
a = a >= 0.5
```

```
pred = pd.DataFrame(data = {"label":a}).astype(int)
```

Calculating the accuracy:

Accuracy for logistic regression is defined as ratio of correctly classified points to the ratio of total points.

```
accuracy = 1 - np.sum(abs(predicted_data['label'] - predicted_data['Actual_label']))/predicted_data.shape[0]
```

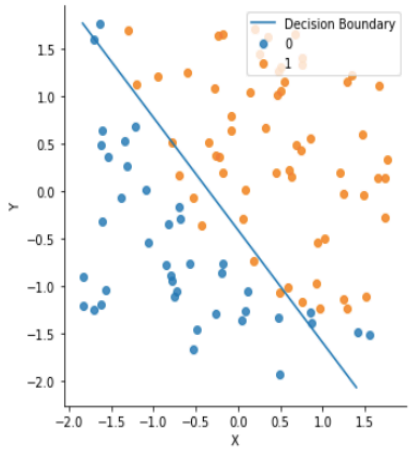
```
print("Accuracy of the model without any parametersc tunning is :",accuracy)
```

```
Accuracy of the model without any parametersc tunning is : 0.8666666666666667
```

Plotting the Decision Boundary:

```
import seaborn as sns
sns.lmplot( x="X", y="Y", data=plot_data, fit_reg=False,hue = 'label' ,legend=False)
x_0 = min(plot_data['X'])
x_1 = max(plot_data['X'])
plt.plot([x_0,1 * -(weights[0] + weights[1]* x_0)/weights[2]], [ x_1,1 * -(weights[0] + weights[1]* x_1)/weights[2] ],l
plt.legend(loc='upper right')
```

```
<matplotlib.legend.Legend at 0x1a2b8debe0>
```



Using regularisation:

We can modify our newton's method a bit to add a regulariser parameter to penalise our higher order polynomials.

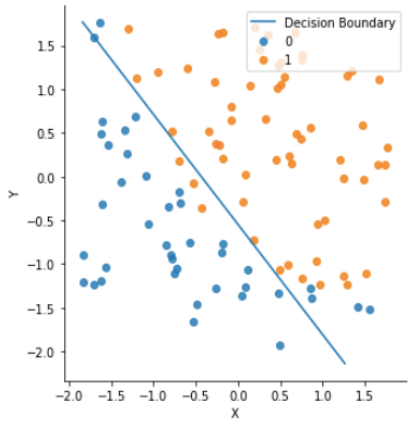
Regularised Newton's method:

```
def newton(w,epochs,lamda):
    weights = []
    errors = []
    w = w.reshape(-1,1)
    for i in range(epochs):
        pred = np.dot(X_train,w)
        print(pred.shape)
        h = sigmoid(pred)
        #h = h.reshape(-1,1)
        a = h*(1-h)*X_train
        error = loss(Y_train.values.reshape(-1,1),h)
        grad = np.dot(X_train.T,h- Y_train.values.reshape(-1,1)) + lamda * w
        hess = a.T @ X_train + lamda
        weights.append(w)
        errors.append(error)
        w = w - np.matmul(np.linalg.pinv(hess),grad);
    return weights,errors
```

Once obtaining the weight values from newton's method, we can predict our values as per the steps defined above and calculate the accuracy.

```
import seaborn as sns
sns.lmplot( x="X", y="Y", data=plot_data, fit_reg=False,hue = 'label' ,legend=False)
x_0 = min(plot_data['X'])
x_1 = max(plot_data['X'])
plt.plot([x_0,1 * -(weights[0] + weights[1]* x_0)/weights[2]], [ x_1,1 * -(weights[0] + weights[1]* x_1)/weights[2] ],la
plt.legend(loc='upper right')
```

```
<matplotlib.legend.Legend at 0x1a18dcb8d0>
```



Comparison: As compared to gradient descent algorithm, cost function converges very fast in case of Newton's method. However it doesn't effect much on the accuracy of the model.