1. Using the data set of two examination results design a predictor using logistic regression for predicting whether a student can get an admission in the institution. Use regularizer to further tune the parameters. Use 70 % data for training and rest 30% data for testing your predictor and calculate the efficiency of the predictor/hypothesis.

   Hints: 1. You can pre process the data for convenience

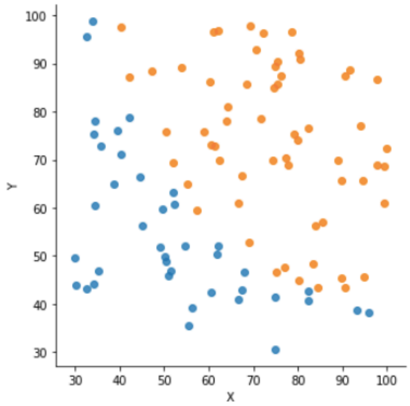
   2. You must use Python program for evaluating parameters using batch gradient descent algorithm (GDA). No function should be used for GDA.

Answer:

As the data is given in the word form, first we need to convert the data into specified format. I have converted it into csv format and read the data in the form of data frame.

Data visualization:

```
34]: import seaborn as sns
     # Use the 'hue' argument to provide a factor variable
     sns.lmplot( x="X", y="Y", data=df, fit_reg=False,hue = 'label' ,legend=False)
     plt.show()
```



It can be seen that data can be separated using a linear curve.

Normalisation: Mean shifting and variance scaling has been used for normalisation.

```
[4]: X = (X - np.mean(X))/np.std(X)
```

Adding bias term and initialising weights.

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

```
[6]: X_train, X_test,Y_train,Y_test = train_test_split(X,Y,test_size = 0.3)
```

```
[7]: w = np.random.normal(0,1,3)
```

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()
```

Gradient Descent algorithm for Logistic regression:

Gradient descent algorithm is as earlier except different cost functions and it's derivative.

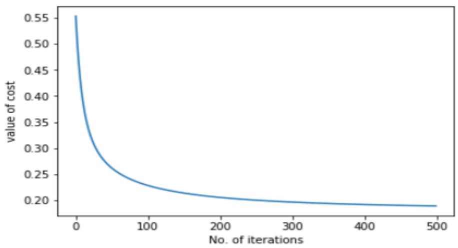
```
def gradient_descent(w,alpha,num_iters):
    theta = []
    cost = []
    for i in range(num_iters):
        pred = np.dot(X_train,w)
        h = sigmoid(pred)
        error = loss(Y_train,h)
        grad = np.dot(X_train.T,h- Y_train)/Y_train.size
        theta.append(w)
        cost.append(error)
        w = w - alpha * grad
    return cost,theta
```

```
cost, theta = gradient_descent(w,0.6,100)
```

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

Cost vs. Number of iteration curve.

```
13]: plt.plot(cost)
     plt.xlabel("No. of iterations")
     plt.ylabel("value of cost")
13]: Text(0,0.5,'value of cost')
```



Predicting the class:

By taking sigmoid of the dot product, we can predict the class.

```
21]: def pred(data):
         return sigmoid(np.dot(data,theta))
```

```
22]: a = pred(X_test)
```

```
23]: 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]
```

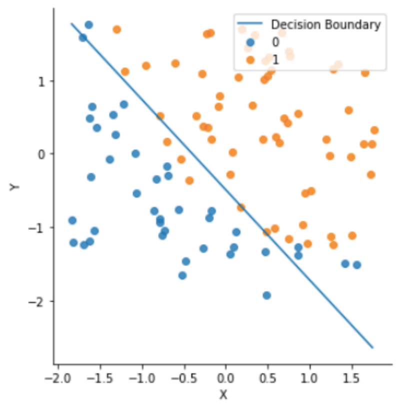
```
print("Accuracy of the model without any parametersc tunning is :",accuracy)
```

```
Accuracy of the model without any parameter 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 * -(theta[0] + theta[1]* x_0)/theta[2]], [ x_1,1 * -(theta[0] + theta[1]* x_1)/theta[2] ],label = "Deci
plt.legend(loc='upper right')
```

```
<matplotlib.legend.Legend at 0x1a1541c828>
```



Using Regularisation

For using regularisation, we need to add some polynomial features first in order to create a non-linear classifier. Three extra features has been added which is X^2, Y^2, X_Y. After adding these extra features we can try to re-run out algorithm.

Below Code has been implemented by me to add polynomial features.

```
#Adding polynomial features
X['x^2'] = X['X'] ** 2
```

```
X['Y^2'] = X['Y'] ** 2
X['X_Y'] = X['X'] * X['Y']
```

```
X.head(
```

	X	Y	x^2	Y^2	X_Y
0	34.623660	78.024693	1198.797805	6087.852690	2701.500406
1	30.286711	43.894998	917.284849	1926.770807	1329.435094
2	35.847409	72.902198	1285.036716	5314.730478	2613.354893
3	60.182599	86.308552	3621.945269	7449.166166	5194.273015
4	79.032736	75.344376	6246.173368	5676.775061	5954.672216

Gradient descent algorithm can be modified a bit for regularisation. Below implementation has been used for calculation.

```
In [63]: lemda = 0.001
```

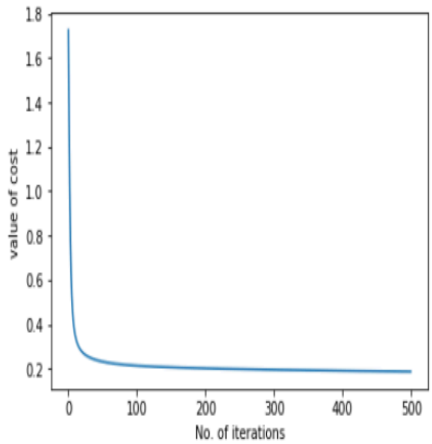
```
In [64]: def gradient_descent(w,alpha,num_iters,lemda):
             theta = []
             cost = []
             lembda_mat = lemda * np.identity(X.shape[1])
             lembda_mat[0][0] = 0
             for i in range(num_iters):
                 pred = np.dot(X_train,w)
                 h = sigmoid(pred)
                 error = loss(Y_train,h) + lemda * np.dot(w.T, w)
                 grad = (np.dot(X_train.T,h- Y_train) + np.matmul(lembda_mat,w))/Y_train.size
                 theta.append(w)
                 cost.append(error)
                 w = w - alpha * grad
             return cost,theta
```

```
In [85]: cost, theta = gradient_descent(w,0.3,500,lemda)
```

Cost vs number of Iterations:

```
[19]: plt.plot(cost)
      plt.xlabel("No. of iterations")
      plt.ylabel("value of cost")
```

```
:[19]: Text(0,0.5,'value of cost')
```



Calculating the accuracy:

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

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

```
[28]: accuracy
```

```
:[28]: 0.9333333333333333
```

Observation:

We can see that accuracy has improved from 0.86667 to 0.93333 using regularisation

Plotting the Decision boundary:

[60]:
```
import seaborn as sns
# Use the 'hue' argument to provide a factor variable
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 * -(theta[0] + theta[1]* x_0)/theta[2]], [ x_1,1 * -(theta[0] + theta[1]* x_1)/theta[2] ],label = "Deci
# Move the legend to an empty part of the plot
plt.legend(loc='best')
```

[60]:
```
<matplotlib.legend.Legend at 0x118910c88>
```