1. Using the data set of two quality test results of a microchip product, design a predictor using logistic regression which will predict the acceptance or rejection of the microchip given the two test results. 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.

Solution:

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.

Preprocessing:

After looking at few of the provided data, it shows that none of the data is varying much so there's no point of normalising the data.

[4]:
```
df = pd.read_csv('chip_data.csv')
```

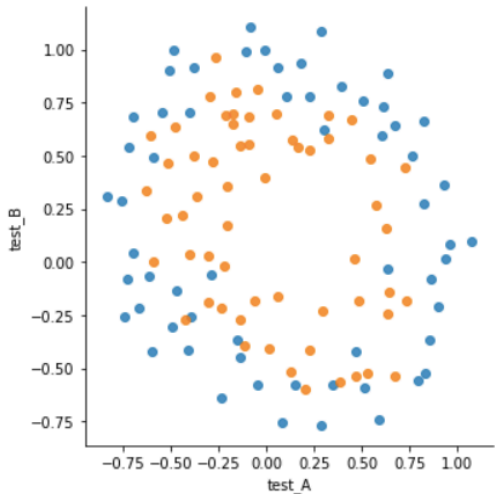
[5]:
```
df.head()
```

[5]:

	test_A	test_B	label
0	0.051267	0.69956	1
1	-0.092742	0.68494	1
2	-0.213710	0.69225	1
3	-0.375000	0.50219	1
4	-0.513250	0.46564	1

We can also plot a data for better intuition. Scatter plot of the data is shown as below.

42]:
```
import seaborn as sns
# Use the 'hue' argument to provide a factor variable
sns.lmplot( x="test_A", y="test_B", data=df, fit_reg=False,hue = 'label' ,legend=False)
plt.show()
```



It seems like we can not separate the data using a simple hyperplane. If we try to separate the data our accuracy will be very poor.

I have separated the data into training set and test set by the ratio of 0.30. I will be training the data on 70 % of the data and output will be checked on rest of the 30 % of the data.

```
#Training and labels
X = df.ilpc[:,0:2]
Y = df.iloc[:,-1]
```

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

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

Initialising the weights:

I have initialised the weights by generating 3 Gaussian distributed points with mean of 0 and variance of 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()
```

Gradient Descent algorithm for Logistic regression:

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

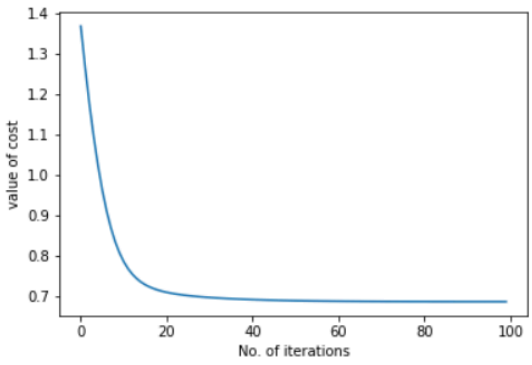
```
def gradient_descent(w,alpha,num_iters):
    theta = []
    cost = []
    for i in range(num_iters):
        pred = np.dot(X_train,w)
        h = sigmoid(pred)
        error = loss(Y_train,h)
        grad = np.dot(X_train.T,h- Y_train)/Y_train.size
        theta.append(w)
        cost.append(error)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

```
In [22]: plt.plot(cost)
         plt.xlabel("No. of iterations")
         plt.ylabel("value of cost")
Out[22]: Text(0,0.5,'value of cost')
```



Predicting Values:

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

```
In [23]: def pred(data):
             return sigmoid(np.dot(data,theta))

In [24]: a = pred(X_test)

In [25]: a = a >= 0.5

In [26]: 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.

```
In [33]: accuracy = 1 - np.sum(abs(predicted_data['label'] - predicted_data['Actual_label']))/predicted_d

In [35]: print("Accuracy of the model without any parameter tunning is :",accuracy)
         Accuracy of the model without any parameter tunning is : 0.4444444444444444
```

Observation:

We can see that the accuracy is quite poor while trying to classifying the data points without using any polynomial feature.

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.

In [7]:
```
#Adding polynomial features
X['test_A^2'] = X['test_A'] ** 2
X['test_B^2'] = X['test_B'] ** 2
X['testA_testB'] = X['test_A'] * X['test_B']
```

In [8]:
```
X.head()
```

Out[8]:

	bias	test_A	test_B	test_A^2	test_B^2	testA_testB
0	1.0	0.051267	0.69956	0.002628	0.489384	0.035864
1	1.0	-0.092742	0.68494	0.008601	0.469143	-0.063523
2	1.0	-0.213710	0.69225	0.045672	0.479210	-0.147941
3	1.0	-0.375000	0.50219	0.140625	0.252195	-0.188321
4	1.0	-0.513250	0.46564	0.263426	0.216821	-0.238990

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

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

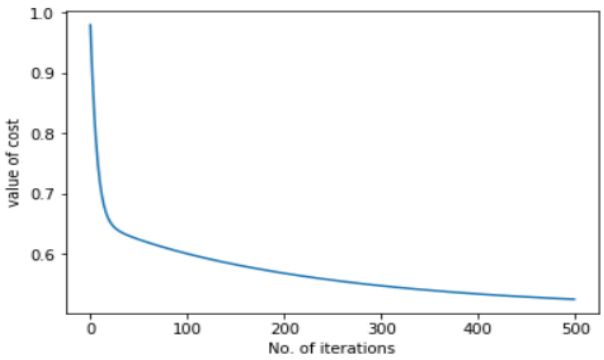
In [64]:
```
def gradient_descent(w,alpha,num_iters,lemda):
    theta = []
    cost = []
    lembda_mat = lemda * np.identity(X.shape[1])
    lembda_mat[0][0] = 0
    for i in range(num_iters):
        pred = np.dot(X_train,w)
        h = sigmoid(pred)
        error = loss(Y_train,h) + lemda * np.dot(w.T,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

In [86]:
```
plt.plot(cost)
plt.xlabel("No. of iterations")
plt.ylabel("value of cost")
```

Out[86]:
```
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.

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

```
print("Accuracy of the model with regularization is :",accuracy)
```

```
Accuracy of the model with regularization is : 0.8888888888888888
```

Observation:

It can be seen that by adding the polynomial features, our accuracy has become almost double.

Plotting the Decision Boundary:

```
h = .005 # step size in the mesh
x_min, x_max = X[:, 1].min() - .1, X[:, 1].max() + .1
y_min, y_max = X[:, 2].min() - .1, X[:, 2].max() + .1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
np.arange(y_min, y_max, h))

t = np.insert(np.c_[xx.ravel(), yy.ravel()],0,1,axis=1)
T = np.insert(t,3,t[:,1]**2,axis=1)
T = np.insert(T,4,t[:,2]**2,axis=1)
T=np.insert(T,5,t[:,1] * t[:,2],axis = 1)
Z = (T @ theta.T) > 0
Z = Z.reshape(xx.shape)
plt.scatter(X[:,1], X[:,2],c=Y)
plt.contour(xx, yy, Z)
```

```
<matplotlib.contour.QuadContourSet at 0x1a1ed9f898>
```