In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
```

Q1_a

In [8]:

```
np.random.seed(24787)
a = np.random.randint(low = 0, high = 8, size = (3,4,4))
#printing the array and its shape
print(a,"\n")
print(a.shape)
```

```
[[[2 6 4 1]
  [0 4 4 3]
  [6 6 1 2]
  [7 0 6 5]]

 [[1 3 3 7]
  [4 7 2 5]
  [0 4 6 7]
  [5 5 7 1]]

 [[7 2 4 5]
  [6 7 7 0]
  [6 2 0 4]
  [2 0 7 6]]]

(3, 4, 4)
```

In [9]:

```
four_indices = np.where(a == 4)
print("row_indices:",four_indices[1],"column_indices",four_indices[2])
#four_indices[2] denotes the depth index of value 4
```

```
row_indices: [0 1 1 1 2 0 2] column_indices [2 1 2 0 1 2 3]
```

Q1_b, using tile to make a (3,8,8) array

In [15]:
```
b = np.tile(a,(1,2,2))
#print(b)
print(b.shape)
```

```
(3, 8, 8)
```

Q1_c, calculating sum along depth of b

In [16]:
```
c = np.sum(b,axis=0)
print(c)
print(c.shape)
```

```
[[10 11 11 13 10 11 11 13]
 [10 18 13  8 10 18 13  8]
 [12 12  7 13 12 12  7 13]
 [14  5 20 12 14  5 20 12]
 [10 11 11 13 10 11 11 13]
 [10 18 13  8 10 18 13  8]
 [12 12  7 13 12 12  7 13]
 [14  5 20 12 14  5 20 12]]
(8, 8)
```

In [30]:
```
import time
np.random.seed(24787)
a = np.random.randint(0,8,(1000,1000))
b = np.random.randint(0,8,(1000,1000))
#declaring array to hold the result
c = np.zeros_like(b)

def matmul(a,b):
    start = time.time()
    for i in range(a.shape[0]):
        for j in range(b.shape[1]):
            c[i,j] = np.dot(a[i],b[:,j].T)
    time_taken = time.time() - start
    return c,time_taken
```

In [31]:
```
c, time_taken_manual = matmul(a,b)
```

```python
start = time.time()
C = a@b
time_taken_inbuilt = time.time() - start
print("Output by matmul function: \n",c)
print("Output by @ operator: \n", C)
print("Difference between matmul and @: \n", C - c)
print("Time taken for matmul: ",time_taken_manual)
print("Time taken for @: ",time_taken_inbuilt)
```

```
Output by matmul function:
 [[12146 12253 12302 ... 12123 12415 12239]
 [12251 12131 12180 ... 12691 12396 12497]
 [11434 11864 12043 ... 12348 11960 12207]
 ...
 [11774 11945 12276 ... 12339 12178 12059]
 [11627 12167 12254 ... 11929 11958 12078]
 [11560 12145 12077 ... 12210 12124 12031]]
Output by @ operator:
 [[12146 12253 12302 ... 12123 12415 12239]
 [12251 12131 12180 ... 12691 12396 12497]
 [11434 11864 12043 ... 12348 11960 12207]
 ...
 [11774 11945 12276 ... 12339 12178 12059]
 [11627 12167 12254 ... 11929 11958 12078]
 [11560 12145 12077 ... 12210 12124 12031]]
difference between matmul and @:
 [[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]]
Time taken for matmul:  4.92743706703186
Time taken for @:  1.779259204864502
```

The difference between C and c = 0, which shows the correctness of our method. The matmul takes 4.92s almost 3 times for @ operator which takes 1.78s

The reason for faster performance of @ operator, is beacuse in our method the loop runs for 1000000 times. While @ operator runs multiplication of various rows of a and columns of b parallelly

```
In [ ]:
```

Q_2 Linear Regression

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:

```
error_list = []
```

In [3]:

```
#function to calculate gradients
def compute(train_data,b0,b1):
    x_values = train_data[:,0]
    y_values = train_data[:,1]
    y_calc = b1*x_values + b0
    diff = y_calc - y_values
    sum_error = np.sum(diff*diff)
    error_list.append(sum_error/(2*x_values.shape[0]))
    gradient_b0 = np.sum(diff)/x_values.shape[0]
    gradient_b1 = np.sum(diff*x_values)/x_values.shape[0]
    return gradient_b0, gradient_b1
```

In [4]:

```
#fucntion to update the weights
def weight_update(train_data,b0,b1,learning_rate,no_of_iterations):
    for i in range(no_of_iterations):
        #print(b0,b1,"\n")
        gradient_b0, gradient_b1 = compute(train_data,b0,b1)
        b0 -= learning_rate*gradient_b0
        b1 -= learning_rate*gradient_b1

    return b0,b1
```

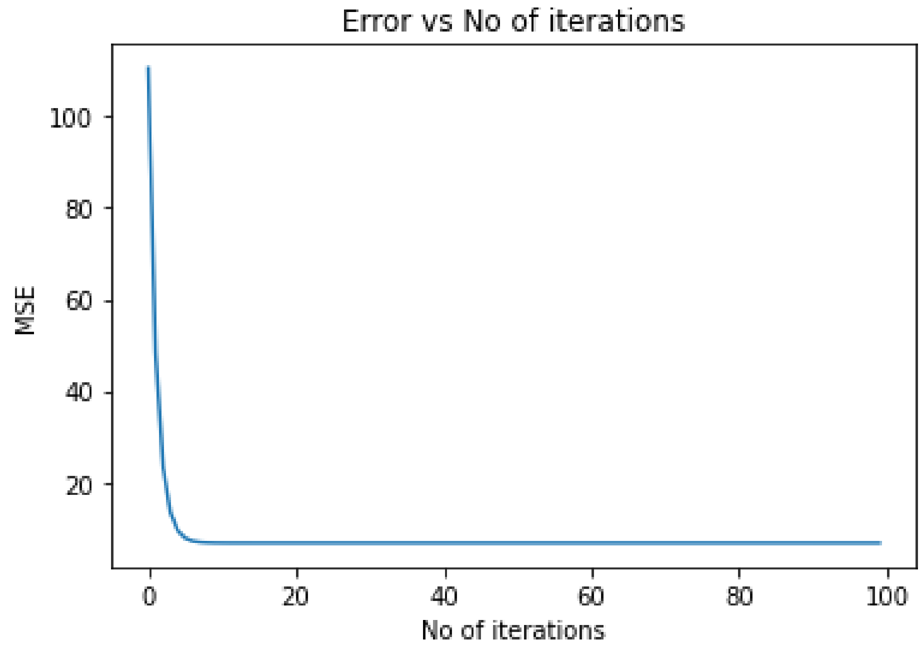
Learning rate .0001

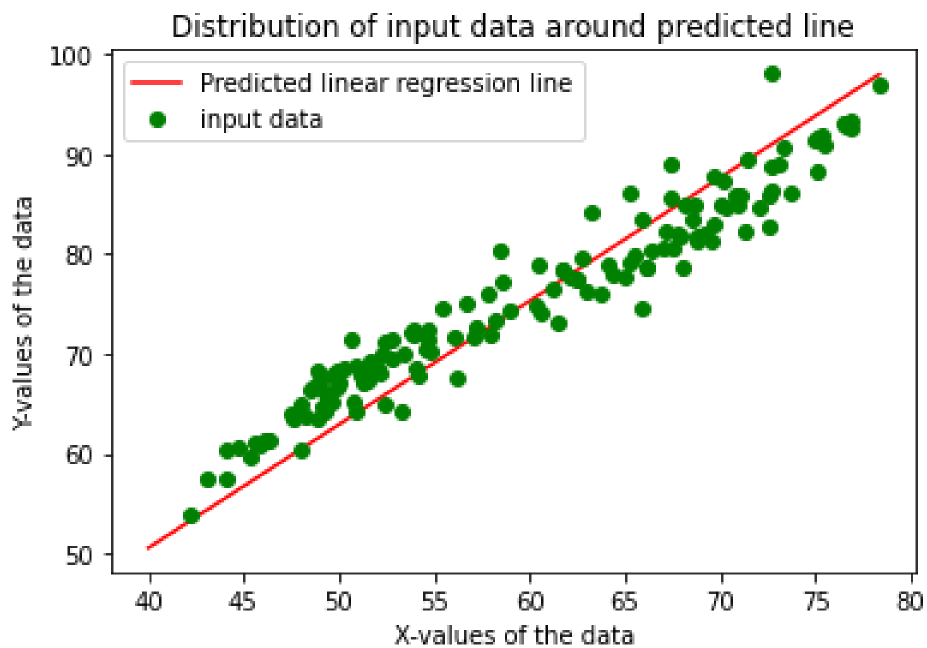
In [5]:

```
#declaring hyper parameters
```

```
error_list = []
b0, b1 = np.random.randint(20,size=2)
learning_rate = 0.0001
data = np.load("data-2.npy")
no_of_iterations = 100
b0, b1 = weight_update(data,1,1,learning_rate,no_of_iterations)

x_data = data[:,0]
y_data = data[:,1]
x_values = np.arange(40,np.max(x_data),.001)
y_values = b0 + b1 * x_values
iteration_list = np.arange(0,no_of_iterations,1)
#printing b0,b1
print("Final intercept:",np.round(b0,4),"and slope: ",np.round(b1,4))
print("Final error: ",error_list[-1])

#plotting various fraphs
fig, ax1 = plt.subplots(1)
ax1.plot(iteration_list,error_list)
ax1.set_xlabel("No of iterations")
ax1.set_ylabel("MSE")
ax1.set_title("Error vs No of iterations")
fig, ax = plt.subplots(1)
#plotting the obtained line

ax.plot(x_values,y_values,'r',label="Predicted linear regression line")
ax.plot(x_data,y_data,'og',label="input data")
ax.set_xlabel("X-values of the data")
ax.set_ylabel("Y-values of the data")
ax.set_title("Distribution of input data around predicted line")
ax.legend()
plt.show()
```

```
Final intercept: 1.0082 and slope:  1.2376
Final error:  6.7802409829839005
```





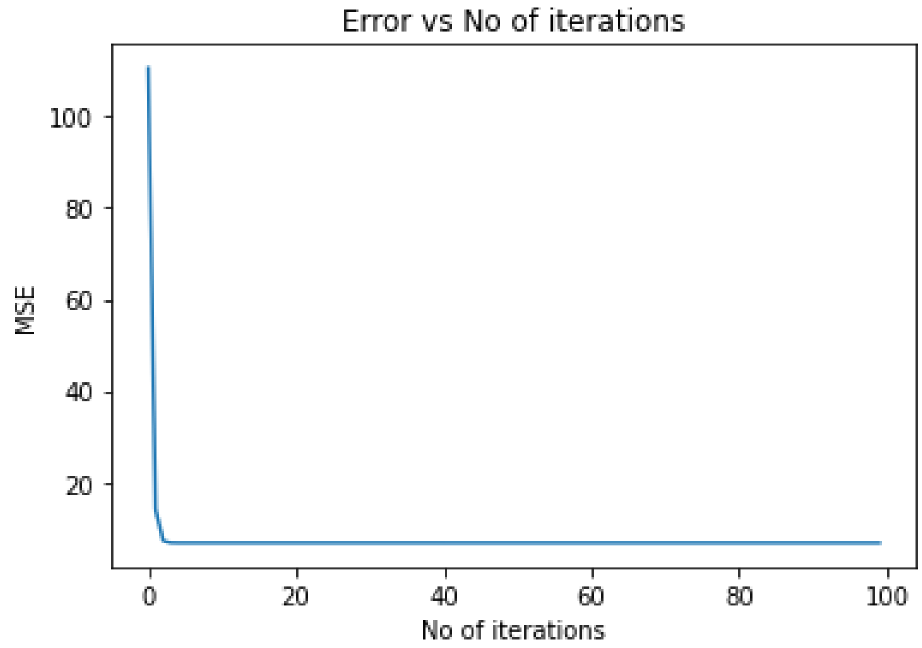
Learning rate 0.0002

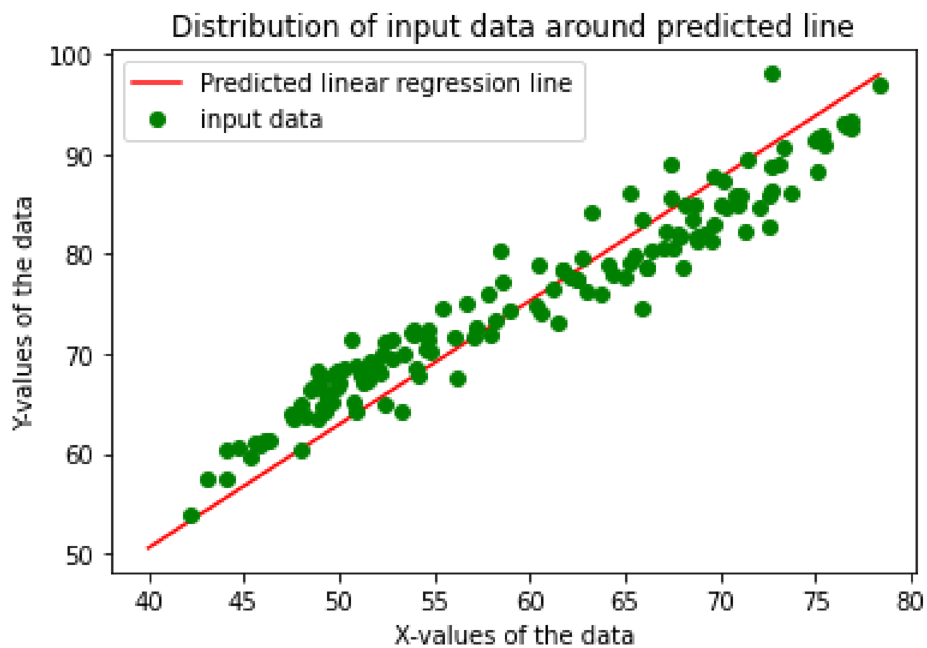
In [6]:

```
error_list = []
b0, b1 = 1,1
```

```
learning_rate = 0.0002
data = np.load("data-2.npy")
no_of_iterations = 100
b0, b1 = weight_update(data,1,1,learning_rate,no_of_iterations)
iteration_list = np.arange(0,no_of_iterations,1)

print("Final intercept: ",np.round(b0,4),"and slope: ",np.round(b1,4))
print("Final error",error_list[-1])

fig, ax1 = plt.subplots(1)
ax1.plot(iteration_list,error_list)
ax1.set_xlabel("No of iterations")
ax1.set_ylabel("MSE")
ax1.set_title("Error vs No of iterations")
fig, ax = plt.subplots(1)
#plotting the obtained line

ax.plot(x_values,y_values,'r',label="Predicted linear regression line")
ax.plot(x_data,y_data,'og',label="input data")
ax.set_xlabel("X-values of the data")
ax.set_ylabel("Y-values of the data")
ax.set_title("Distribution of input data around predicted line")
ax.legend()
plt.show()
```

```
Final intercept:  1.0125 and slope:  1.2375
Final error 6.778377807023614
```





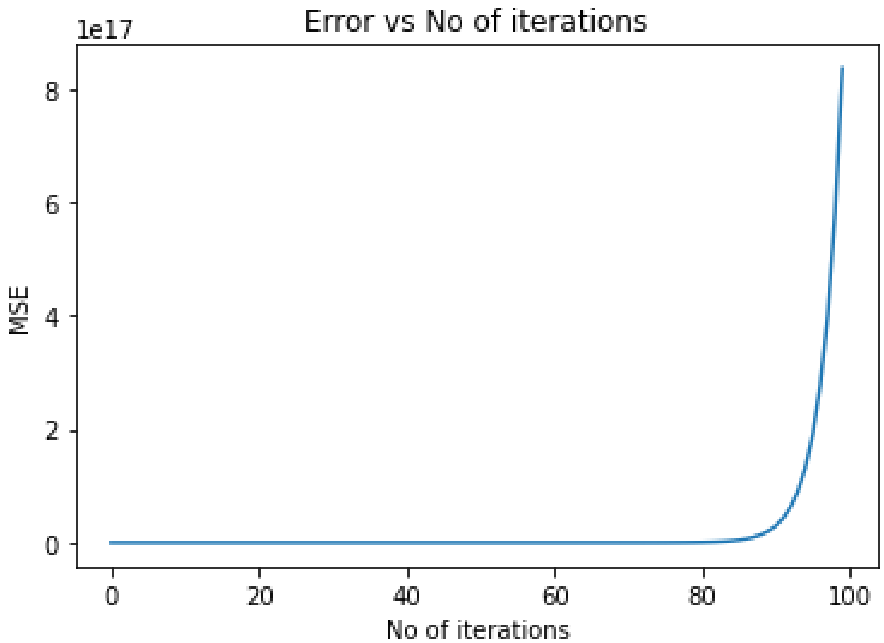
Learning rate 0.0006

In [7]:
```
error_list = []
b0, b1 = 1,1
```

```
learning_rate = 0.0006
data = np.load("data-2.npy")
no_of_iterations = 100
b0, b1 = weight_update(data,1,1,learning_rate,no_of_iterations)
iteration_list = np.arange(0,no_of_iterations,1)

print("Final intercept: ",np.round(b0,4),"and slope: ",np.round(b1,4))
print("Final error",error_list[-1])

fig, ax1 = plt.subplots(1)
ax1.plot(iteration_list,error_list)
ax1.set_xlabel("No of iterations")
ax1.set_ylabel("MSE")
ax1.set_title("Error vs No of iterations")
```

```
Final intercept:  -418375.976 and slope:  -25679229.1794
Final error 8.365249706795523e+17
```

Out[7]: `Text(0.5, 1.0, 'Error vs No of iterations')`



Learning rate 0.01

In [8]:

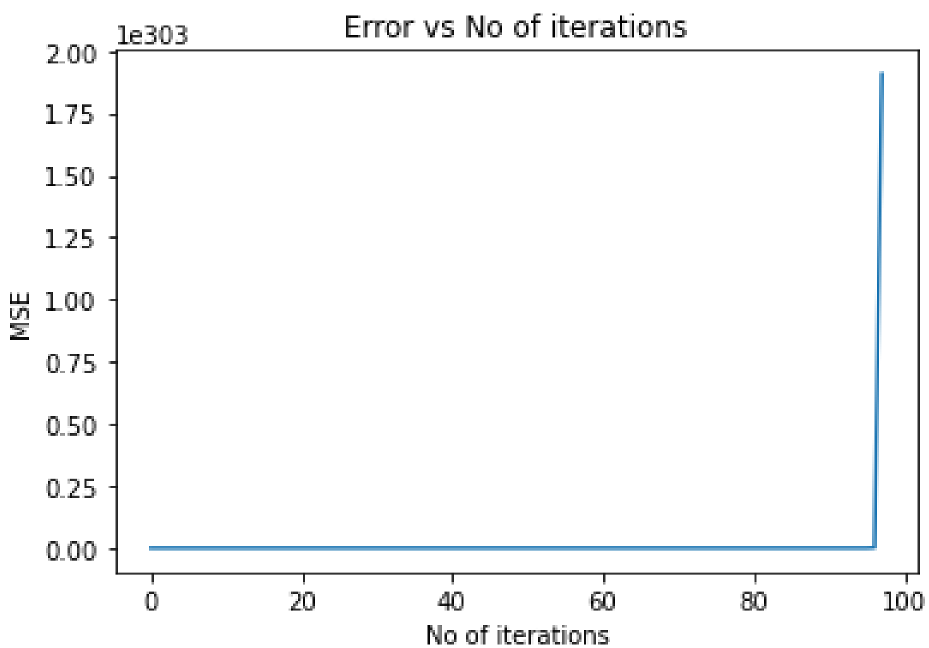
```
error_list = []
```

```python
b0, b1 = 1,1
learning_rate = 0.01
data = np.load("data-2.npy")
no_of_iterations = 100
b0, b1 = weight_update(data,1,1,learning_rate,no_of_iterations)
iteration_list = np.arange(0,no_of_iterations,1)

print("Final intercept: ",np.round(b0,4),"and slope: ",np.round(b1,4))
print("Final error",error_list[-1])

fig, ax1 = plt.subplots(1)
ax1.plot(iteration_list,error_list)
ax1.set_xlabel("No of iterations")
ax1.set_ylabel("MSE")
ax1.set_title("Error vs No of iterations")
```

```
Final intercept:  -7.568487598507173e+152 and slope:  -4.6454019747292825e+154
Final error inf
```

```
/home/akshay/.local/lib/python3.8/site-packages/numpy/core/fromnumeric.py:86: RuntimeWarning: over
flow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
<ipython-input-3-e36a763e0022>:7: RuntimeWarning: overflow encountered in multiply
  sum_error = np.sum(diff*diff)
```

Out[8]:
```
Text(0.5, 1.0, 'Error vs No of iterations')
```



Learning rate 10

In [9]:

```
error_list = []
b0, b1 = 1,1
learning_rate = 10
no_of_iterations = 100
b0, b1 = weight_update(data,1,1,learning_rate,no_of_iterations)
iteration_list = np.arange(0,no_of_iterations,1)

print("Final intercept:",np.round(b0,4),"and slope: ",np.round(b1,4))
print("Final error",error_list[-1])

fig, ax1 = plt.subplots(1)
ax1.plot(iteration_list,error_list)
ax1.set_xlabel("No of iterations")
ax1.set_ylabel("MSE")
ax1.set_title("Error vs No of iterations")
```

```
Final intercept: nan and slope:  nan
Final error nan
```

```
<ipython-input-3-e36a763e0022>:7: RuntimeWarning: overflow encountered in multiply
  sum_error = np.sum(diff*diff)
/home/akshay/.local/lib/python3.8/site-packages/numpy/core/fromnumeric.py:86: RuntimeWarning: over
flow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
<ipython-input-3-e36a763e0022>:10: RuntimeWarning: overflow encountered in multiply
  gradient_b1 = np.sum(diff*x_values)/x_values.shape[0]
<ipython-input-4-e3c348689dc5>:6: RuntimeWarning: invalid value encountered in double_scalars
  b0 -= learning_rate*gradient_b0
<ipython-input-4-e3c348689dc5>:7: RuntimeWarning: invalid value encountered in double_scalars
  b1 -= learning_rate*gradient_b1
```

Out[9]:

```
Text(0.5, 1.0, 'Error vs No of iterations')
```

Best Learning rate according to me is either .0001 or 0.0002, they both converge 5-6 epochs, as you can see above .0006,.1,10 diverges. The final MSE of 0.0001 and 0.0002 are comparable. .0001 offers more smooth curve compared to .0002

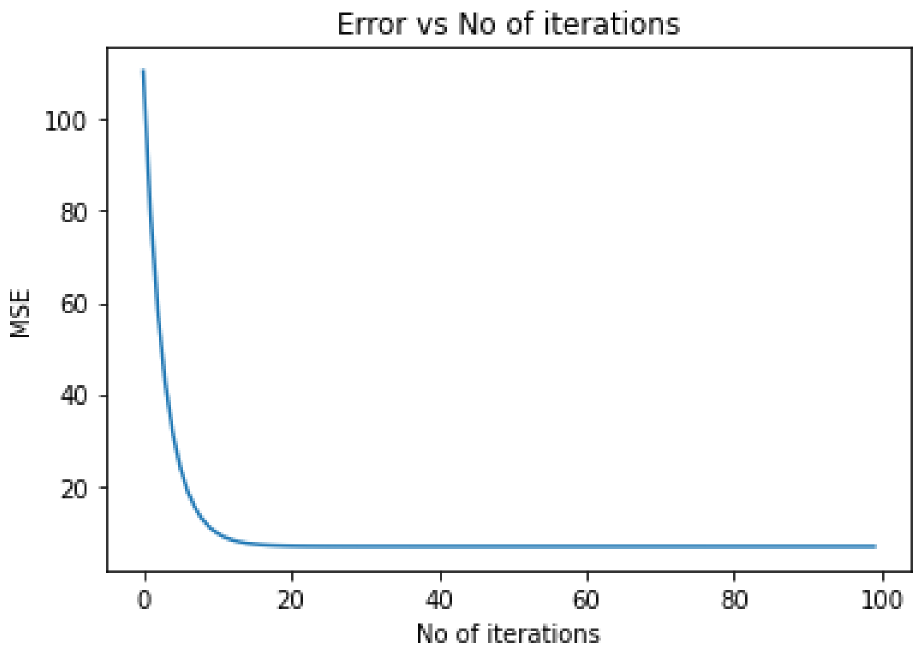
Showing Proof for the upper bound , obtained upper bound 0.0005

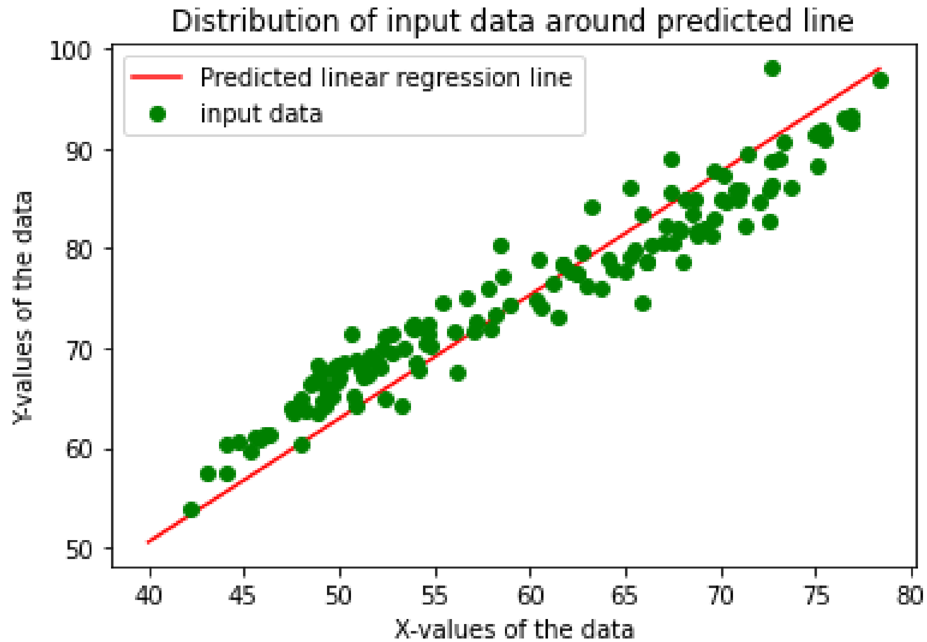
For 0.0005 below cell

In [10]:

```
error_list = []
b0, b1 = 1,1
learning_rate = 0.0005
no_of_iterations = 100
b0, b1 = weight_update(data,1,1,learning_rate,no_of_iterations)
iteration_list = np.arange(0,no_of_iterations,1)

fig, ax1 = plt.subplots(1)
ax1.plot(iteration_list,error_list)
```

```
ax1.set_xlabel("No of iterations")
ax1.set_ylabel("MSE")
ax1.set_title("Error vs No of iterations")
fig, ax = plt.subplots(1)
#plotting the obtained line
ax.plot(x_values,y_values,'r',label="Predicted linear regression line")
ax.plot(x_data,y_data,'og',label="input data")
ax.set_xlabel("X-values of the data")
ax.set_ylabel("Y-values of the data")
ax.set_title("Distribution of input data around predicted line")
ax.legend()
plt.show()
```





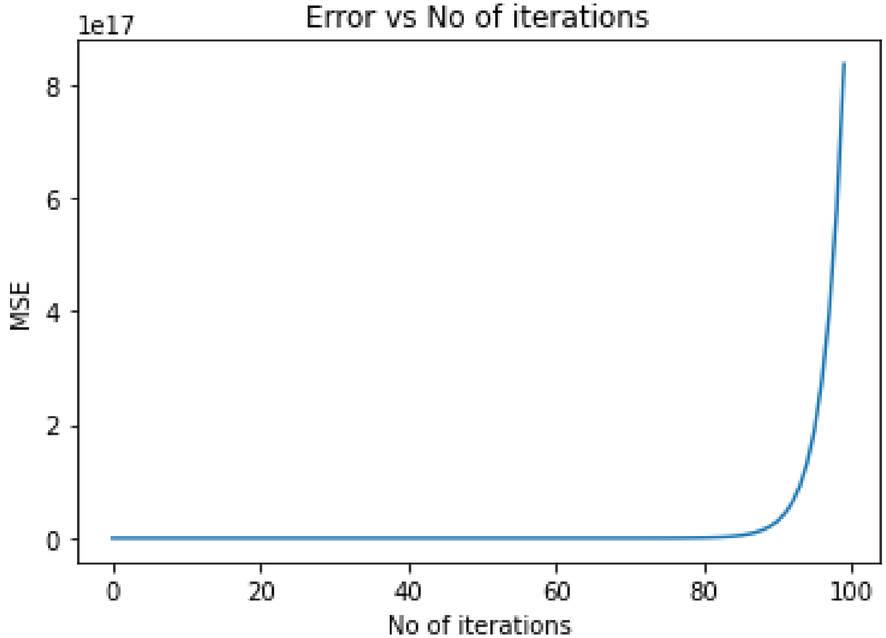
For 0.0006 the below diagram diverges

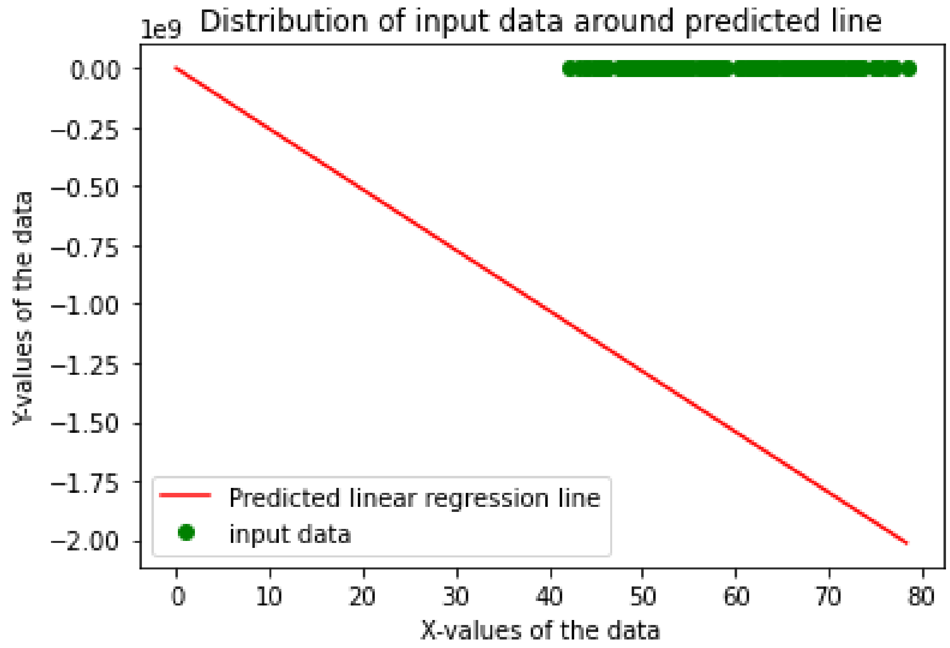
In [11]:

```
error_list = []
b0, b1 = 1,1
learning_rate = 0.0006
no_of_iterations = 100
b0, b1 = weight_update(data,1,1,learning_rate,no_of_iterations)
x_values = np.arange(0,np.max(x_data),.001)
y_values = b0 + b1 * x_values
iteration_list = np.arange(0,no_of_iterations,1)

print("Final intercept: ",np.round(b0,4),"and slope: ",np.round(b1,4))
print("Final error",error_list[-1])

fig, ax1 = plt.subplots(1)
ax1.plot(iteration_list,error_list)
ax1.set_xlabel("No of iterations")
ax1.set_ylabel("MSE")
ax1.set_title("Error vs No of iterations")
fig, ax = plt.subplots(1)
ax.plot(x_values,y_values,'r',label="Predicted linear regression line")
ax.plot(x_data,y_data,'og',label="input data")
```

```
ax.set_xlabel("X-values of the data")
ax.set_ylabel("Y-values of the data")
ax.set_title("Distribution of input data around predicted line")
ax.legend()
plt.show()
```

```
Final intercept:  -418375.976 and slope:  -25679229.1794
Final error 8.365249706795523e+17
```





Conclusion

The error increases when lr is .0006, when the lr is 0.0006 the regression line obtained diverges from the data points so much

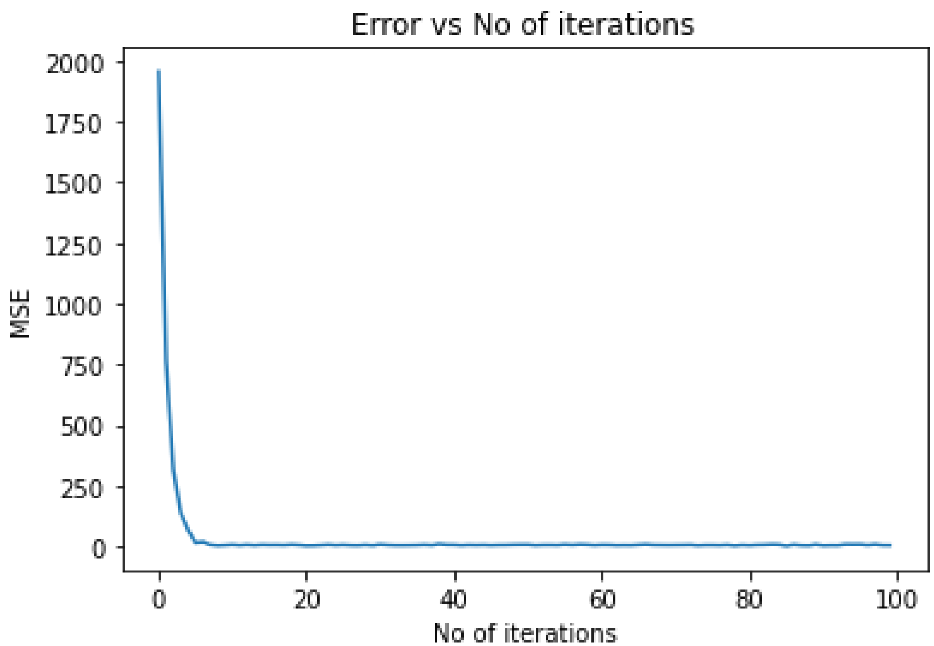
In [12]:

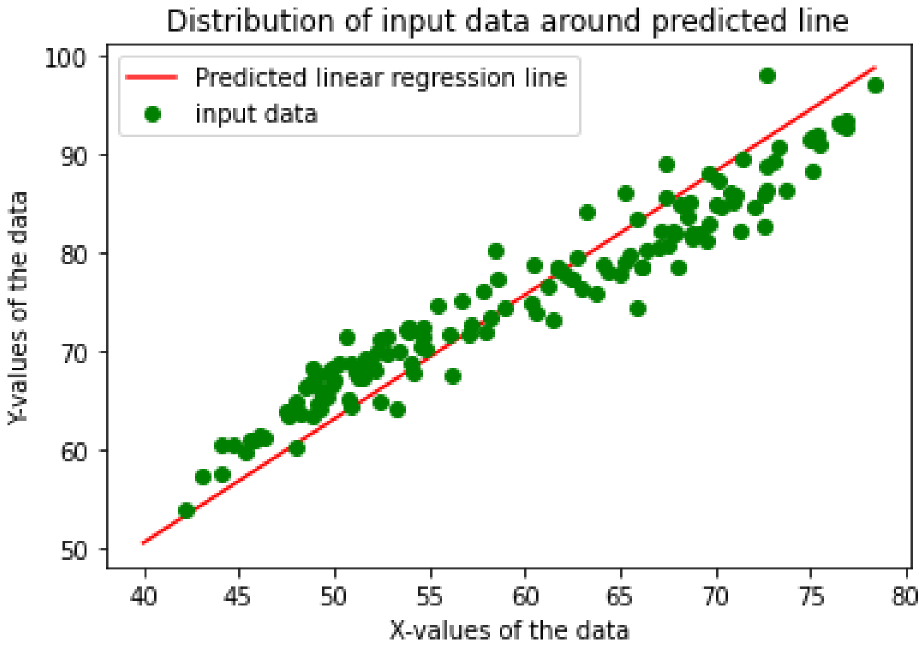
```
#funtion used for mini batch gradient
def batch_weight_update(train_data,b0,b1,learning_rate,no_of_iterations,batch_size):
    for i in range(no_of_iterations):
        np.random.shuffle(train_data)
        gradient_b0, gradient_b1 = compute(train_data[:batch_size],b0,b1)
        b0 -= learning_rate*gradient_b0
        b1 -= learning_rate*gradient_b1
    return b0,b1
```

In [13]:

```
error_list = []
b0, b1 = np.random.rand(2,1)
learning_rate = 0.0001
no_of_iterations = 100
```

```
batch_size = 20
b0, b1 = batch_weight_update(data,b0,b1,learning_rate,no_of_iterations,batch_size)
#plotting the obtained line
x_values = np.arange(40,np.max(x_data),.001)
y_values = b0 + b1 * x_values
iteration_list = np.arange(0,no_of_iterations)

fig, ax1 = plt.subplots(1)
ax1.plot(iteration_list,error_list)
ax1.set_xlabel("No of iterations")
ax1.set_ylabel("MSE")
ax1.set_title("Error vs No of iterations")
fig, ax = plt.subplots(1)
ax.plot(x_values,y_values,'r',label="Predicted linear regression line")
ax.plot(x_data,y_data,'og',label="input data")
ax.set_xlabel("X-values of the data")
ax.set_ylabel("Y-values of the data")
ax.set_title("Distribution of input data around predicted line")
ax.legend()
plt.show()
```





Minibatch converges more with less epochs and less data. And the final MSE we got is around 5.81 is lower than batch gradient descent, and also the error oscillates a little bit for minibatch gradient

Verifying with sklearn

In [14]:

```
import sklearn
from sklearn.linear_model import LinearRegression
x_values = data[:,0].reshape(-1,1)
y_values = data[:,1].reshape(-1,1)
reg = LinearRegression().fit(x_values,y_values)
print(reg.coef_,reg.intercept_)
y_pred = reg.predict(x_values)
sklearn.metrics.mean_squared_error(y_pred,y_values)
```

```
[[0.961082]] [17.98049938]
```

Out[14]:

```
6.195654144451672
```

In [ ]:

```

```

HW2-Q2-Normal-Equation-Derivation

Akshay Antony

October 2, 2021

Normal equations:
$J : cost(meansquarederror)$
θ^T weights
$Y : labels$
$X : input - data$

$h_\theta(x) = \theta^T X$
$J = (1/2m) * (\theta^T X - Y)^T (\theta^T X - Y)$
$J = (1/2m) * (((\theta^T X)^T.\theta^T X) - Y^T \theta^T X - (\theta^T X)^T Y + Y^T.Y)$
Neglecting the constant 1/2m
$J = X^T \theta \theta^T X - Y^T \theta^T X - X^T \theta Y + Y^T Y$ Taking derivative wrt θ
$\partial J/\partial\theta = 2X^T X\theta - X^T Y - X^T Y$
equating to 0
$2X^T X\theta = 2X^T Y$
$\theta = (X^T X)^{-1} X^T Y$

Gradient Descent and Update:
$y' = b_1 * x + b_0$
where y' is the predicted value
Sum of squared error is: $\sum_{i=1}^{m}(y - (b_1 x + b_0))^2$

Taking the Mean squared error as:
$(1/2m) * \sum_{i=1}^{m}(y - (b_1 x + b_0))^2$
To minimize the MSE, we take the differential w.r.t b_1, b_0

Taking derivative w.r.t b_1
$= (1/m)(\sum_{i=1}^{m} 2(y - (b_1 x + b_0)) * b_1$ —(1)

$\frac{\partial J}{\partial b_1} = (1/m)(\sum_{i=1}^{m}(y - (b_1 x + b_0)) * b_1$

Similarly
$\frac{\partial J}{\partial b_0} = (1/m) * (\sum + i = 1^m(y - (b_1 x + b_0)))$ —(2)

Applying gradient descent to update b_1, b_0
$b_1 = b_1 - \frac{\partial J}{\partial b_1} * \alpha$ where α is the learning rate
$b_0 = b_0 - \frac{\partial J}{\partial b_0} * \alpha$
There are mainly 3 types of gradient descent:
1. Batch gradient descent: Here gradient descent is done after all the data in a batch is passed once. That means the parameters are updated only after the whole training set is passed
2. Mini batch gradient descent: Here the whole training set is divided into mini-batches and passed to the model. The parameters are updated by gradient descent after each mini-batch is processed.
3. Stochastic Gradient Descent: Here the parameters are updated after each training example in the training data. That means for the whole data set, this update happens m times according to equations 1 and 2.

Question 3: Logistic Regression

In [1]:

```
#Import all the required libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Load the data

In [2]:

```
# load the data
data_input_0 = pd.read_csv("/home/akshay/Downloads/MAIL/Assignment 2/class0-input.csv")
data_input_1 = pd.read_csv("/home/akshay/Downloads/MAIL/Assignment 2/class1-input.csv")
data_labels = pd.read_csv("/home/akshay/Downloads/MAIL/Assignment 2/labels.csv")

# Perform important operations on the data
X = pd.concat([data_input_0,data_input_1],axis=0)
X = X.to_numpy()
X = np.float64(X)
Y = data_labels.to_numpy()
Y = np.float64(Y)
```

Check the shape

In [3]:

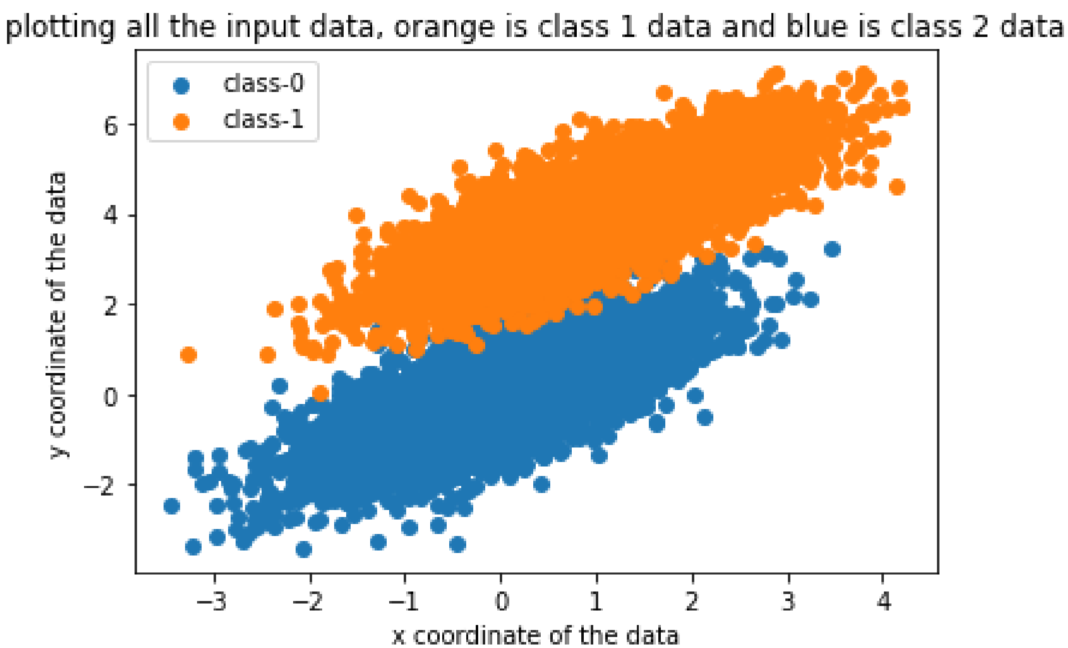
```
# Shape of X
print(X.shape)
# Shape of Y
print(Y.shape)
```

```
(10000, 2)
(10000, 1)
```

Visualize the data

In [4]:
```
# Use different colors for each class
# Use plt.scatter
fig, ax = plt.subplots(1)
ax.scatter(X[:5000,0],X[:5000,1],label="class-0")
ax.scatter(X[5000:10000,0],X[5000:10000,1],label="class-1")
# Dont forget to add axes titles, graph title, legend
ax.set_xlabel("x coordinate of the data")
ax.set_ylabel("y coordinate of the data")
ax.set_title("plotting all the input data, orange is class 1 data and blue is class 2 data")
ax.legend()
```

Out[4]:
```
<matplotlib.legend.Legend at 0x7f442b303760>
```



Define the required functions

In [5]:
```
# Pass in the required arguments
# Implement the sigmoid function
def sigmoid(x):
    sig_x = 1/(1 + np.exp(-x))
    return sig_x
```

In [6]:
```
# Pass in the required arguments
# The function should return the gradients
def calculate_gradients(Y,X,sig_x):
    grad_x1 = (Y - sig_x).squeeze()*X[:,0]
    grad_x2 = (Y - sig_x).squeeze()*X[:,1]
    grad_x0 = (Y - sig_x)
    current_grads = np.asarray([[np.sum(grad_x0)/Y.shape[0]],[np.sum(grad_x1)/Y.shape[0]]
                                ,[np.sum(grad_x2)/Y.shape[0]]])
    #print(current_grads)
    return current_grads
```

In [7]:
```
# Update the weights using gradients calculated using above function and learning rate
# The function should return the updated weights to be used in the next step
def update_weights(prev_weights, current_grads, learning_rate):
    prev_weights += learning_rate*current_grads
    return prev_weights
```

In [8]:
```
# Use the implemented functions in the main function
# 'main' fucntion should return weights after all the iterations
# Dont forget to divide by the number of datapoints wherever necessary!
# Initialize the intial weigths randomly
def main(X, Y, weights, learning_rate = 0.0005, num_steps = 50000):
    updated_weights = weights
    for j in range(num_steps):
        sig_x = sigmoid(X@updated_weights[1:3] + updated_weights[0])
        #predicted = np.where(sig_x<=0.5,0,1)
        current_grads = calculate_gradients(Y,X,sig_x)
        updated_weights = update_weights(updated_weights,current_grads,learning_rate)

    return updated_weights
```

In [9]:
```
# Pass in the required arguments (final weights and input)
# The function should return the predictions obtained using sigmoid function.
def predict(final_weights,X):
    sig_x = sigmoid(X@final_weights[1:3] + final_weights[0])
    return sig_x
```

Visualize the misclassification

In [10]:

```
# Use the final weights to perform prediction using predict funtion
# Convert the predictions to '0' or '1'
# Calculate the accuracy using predictions and labels
#initial_weights = np.random.rand(3,1)
initial_weights = np.asarray([[0.],[0.],[0.]])
final_weights = main(X,Y,weights=initial_weights)
predicted = predict(final_weights,X)
predicted = np.where(predicted<=0.5,0,1)
accuracy = np.sum(predicted == Y)/Y.shape[0]
print("Accuracy: ",accuracy,"Intercept: ",final_weights[0],"Coefficients: ",final_weights[1],fina
```

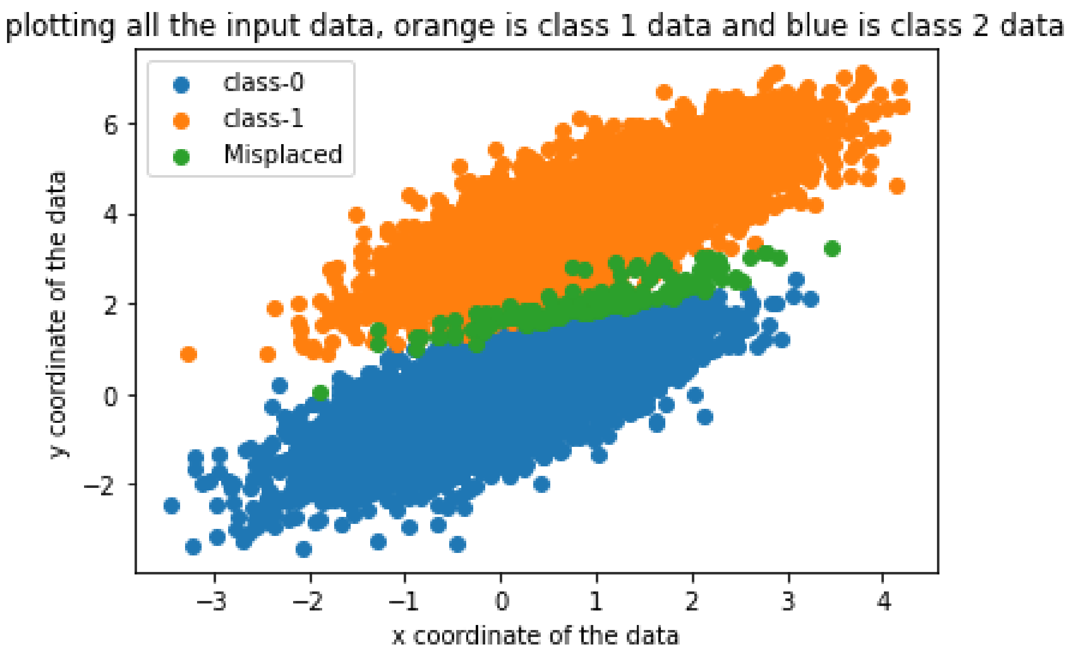
```
Accuracy:  0.984 Intercept:  [-2.20981838] Coefficients:  [-0.59123076] [1.55533894]
```

In [11]:

```
# Use different colors for class 0, class 1 and misclassified datapoints
# Use plt.scatter
# Dont forget to add axes titles, graph title, legend
class0_x,class0_y,class1_x, class1_y, mis_class_x, mis_class_y = [], [], [], [], [], []
for i in range(predicted.shape[0]):
    if(predicted[i] == Y[i] == 0):
        class0_x.append(X[i,0])
        class0_y.append(X[i,1])

    elif(predicted[i] == Y[i] == 1):
        class1_x.append(X[i,0])
        class1_y.append(X[i,1])

    else:
        mis_class_x.append(X[i,0])
        mis_class_y.append(X[i,1])
#print(len(mis_class_x))
fig, ax = plt.subplots(1)
ax.scatter(class0_x,class0_y,label="class-0")
ax.scatter(class1_x,class1_y,label="class-1")
ax.scatter(mis_class_x,mis_class_y,label="Misplaced")
# Dont forget to add axes titles, graph title, legend
ax.set_xlabel("x coordinate of the data")
ax.set_ylabel("y coordinate of the data")
```

```python
ax.set_title("plotting all the input data, orange is class 1 data and blue is class 2 data")
ax.legend()
```

Out[11]: <matplotlib.legend.Legend at 0x7f442914d2b0>



Compare the results with sklearn's Logistic Regression

In [12]:

```python
# import sklearn and necessary libraries
# Print the accuracy obtained by sklearn and your model
```

In [13]:

```python
import sklearn
from sklearn.linear_model import LogisticRegression

Y = Y.reshape(10000)

log_reg_obj = LogisticRegression()
log_reg_obj.fit(X,Y)
print(log_reg_obj.coef_,log_reg_obj.intercept_,log_reg_obj.score(X, Y))
```

```
[[-3.92166117  6.5756403 ]] [-11.2220325] 0.9948
```

Accuracy given by my model: .984

SKLearn Accuracy: .9948

Both accuracies are equal. To increase the accuracy

1. We can pass in the data in batches
2. Optimizing the hyperparameters including lr

In [ ]:

```

```

In [ ]:

```

```