Logistic Regression Report

In this implementation of Logistic Regression, we use only numpy and pandas library to perform binary classification on a dataset with 4 features. We then define a function to perform a random train-test-split to split the data in a 70-30 fashion as required. Also, as required in Logistic Regression, we use the sigmoid function defined as:

$$S(z) = \frac{1}{1 + e^{-z}}$$

We also write a function for the cost function as:

$$J(\theta) = \frac{1}{m} \cdot \left(-y^T \log(h) - (1 - y)^T \log(1 - h)\right)$$

We have separate functions for comuting both of them. These are used in both gradient descent as well as stochastic gradient descent algortihms

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

#LR loss function
def loss(h , y) :
    return (-y * np.log(h) - (1 - y) * np.log(1 - h)).sum()
```

We then have functions for Gradient Descent as well as Stochastic Gradient Descent. In both of these we initialise the weight vectors as zeros. In every iteration, we update our weight vectors according to the gradient of the cost function, (which is the same as linear regression). After updating the weight vector, we calculate the loss of that

particular weight vector and append it to a list of accuracies which we then use to plot graphs for each of the three learning rates. The functions we wrote for gradient descent and stochastic gradient descent

```
#LR using gradient descent

def LogisticRegression(lr  , iterations  , X  , y) :

    rows = X.shape[0] #rows x cols
    features = X.shape[1]
    weights = np.zeros(features)  #initialising weights
    bias = 0  #initialising bias
    y = y.reshape((y.shape[0],))

    l = []
    a = []
    #Gradient descent
    for i in range(iterations):
        w = (X@weights) + bias    #weight vector

        y_pred = sigmoid(w)    #predicting class

        #calculating weight and bias errors
        delta_w = (X.T) @ (y_pred - y)
        delta_b = np.sum(y_pred - y)

        #updating weights and bias
        weights = weights - lr*(delta_w)
        bias = bias - lr*(delta_b)

        binary_pred = [1 if i>0.5 else 0 for i in y_pred]     #classifying class

        if i%50 == 0 :
            l.append(loss(y_pred , y))     #appending loss
            a.append(((binary_pred == y).sum()/len(y)))     #appending accuracy

    return weights,bias,l,a
```

```
#LR using stochastic gradient descent

def LogisticRegressionSGD(lr, iterations  , X  , y) :
    l=[]
    a=[]
    rows , features  = X.shape #rows x cols
    weights = np.zeros(features)    #initialising weights
    bias = 0    #initialising bias
    y = y.reshape((y.shape[0],))
```

```
    iter_no = 0

    #Stochastic Gradient descent
    for i in range(iterations):

        #chosing a random entry
        random_number = random.randint(0, len(y)-1)
        x_b, y_b = X[random_number], y[random_number]

        w = np.dot(x_b, weights) + bias    #weight vector

        y_pred = 1/(1+np.exp(-w))    #predicting class

        #calculating weight and bias errors
        delta_w = np.dot(x_b.T , y_pred - y_b)
        delta_b = np.sum(y_pred - y_b)

        #updating weights and bias
        weights = weights - lr*(delta_w)
        bias = bias - lr*(delta_b)

        binary_pred = predict(weights,bias,X)    #classifying class

        iter_no+=1

        if iter_no%50==0:
            l.append(loss(y_pred , y_b))     #appending loss
            tp,fp,tn,fn = evaluate(binary_pred , y)   #calculating true/false positives/negatives
            a.append((tp+tn)/(tp+tn+fp+fn))     #appending accuracy

    return weights,bias,l,a
```

we also have a function to compute the values of false positives, false negatives, true positives, true negatives that we use to calculate f-score, precision, accuracy etc.

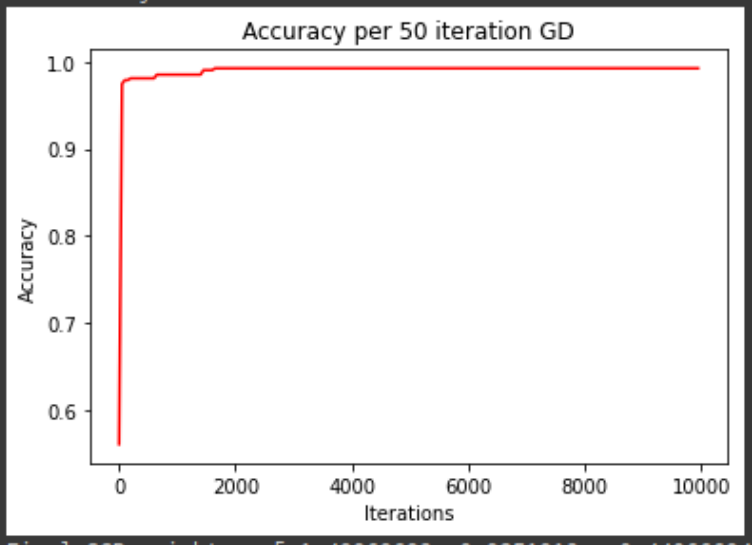
```
#calculating true/false positives/negatives:
def evaluate(y_pred , y):
    k=0
    tp=0
    fp=0
    fn=0
    tn=0
    for x in y_pred :
        if x==1 and y[k]==1:
          tp+=1
        elif x==1 and y[k]!=1:
          fp+=1
        else:
            if y[k] == 1:
```

```
                fn+=1
            else:
                tn+=1
        k+=1
    return tp,fp,tn,fn
```

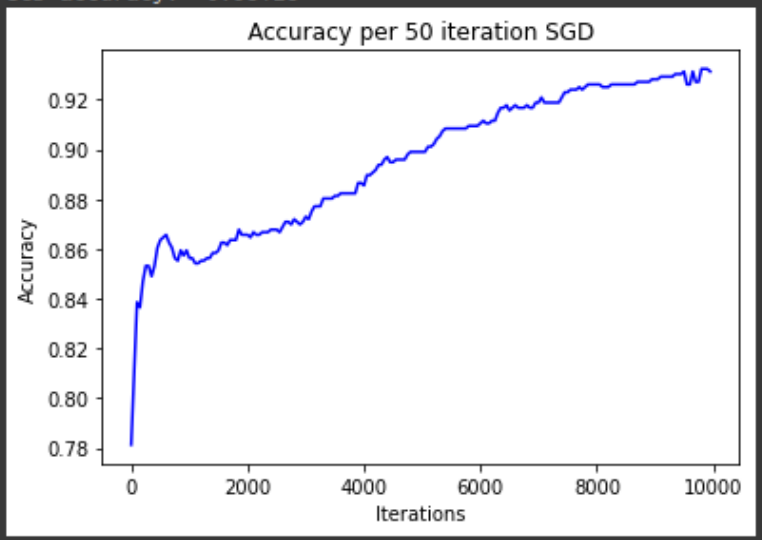
The following are the plots for the different learning rates for both gradient and stochastic gradient descent:

```
-----------------Learning Rate : 0.001 --------------------

Final GD weights:  [-11.25174407 -12.53453735 -11.44191673  -0.25134856]
Final GD bias:  -5.358592217634708
GD accuracy:  0.9927083333333333
```

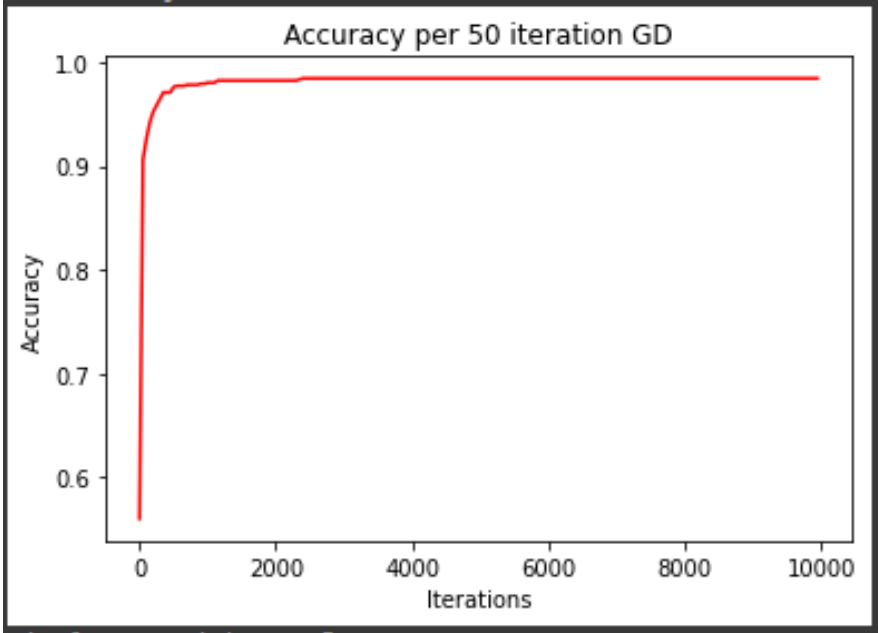



```
Final SGD weights:  [-1.49969693 -0.9251913  -0.44066694 -0.05624126]
Final SGD bias:  -0.2699145817666991
SGD accuracy:  0.93125
```

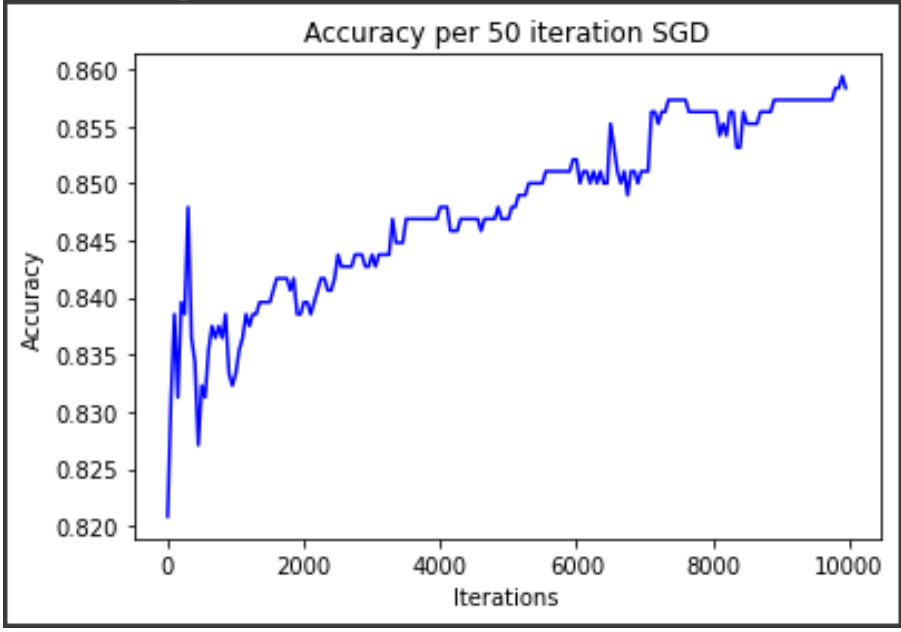


```
-----------------Learning Rate : 0.0001 --------------------

Final GD weights:  [-6.05151127 -6.47279084 -5.90261635  0.28801085]
Final GD bias:  -2.0871082483180405
GD accuracy:  0.984375
```

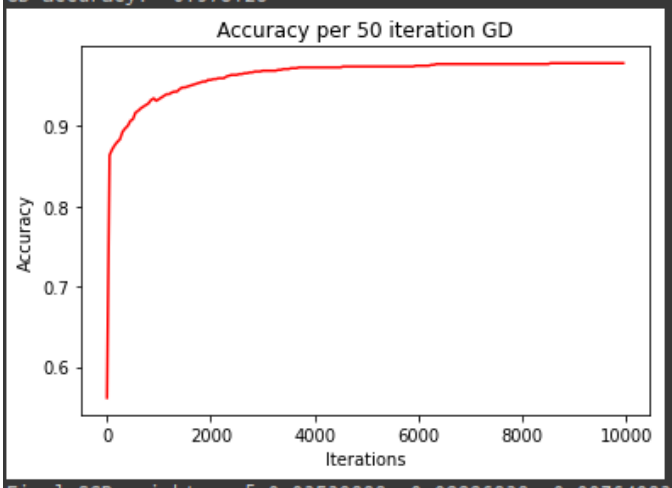



```
Final SGD weights:  [-0.30773083 -0.1865584   0.03683134 -0.00486592]
Final SGD bias:  -0.04772272097583271
SGD accuracy:  0.8583333333333333
```

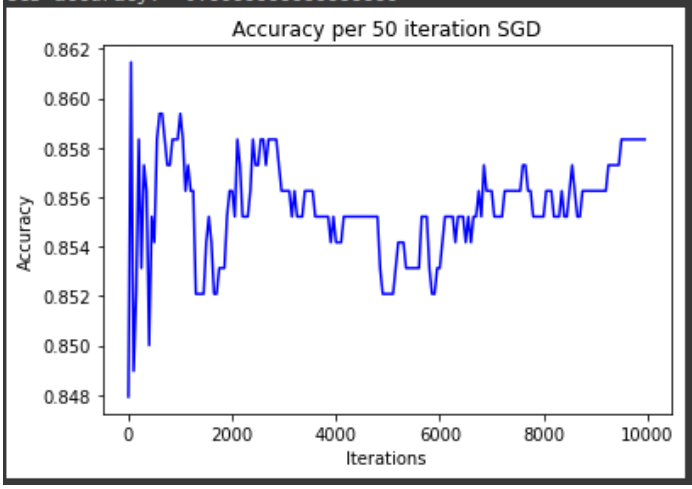


```
-----------------Learning Rate : 1e-05 ---------------------

Final GD weights:  [-3.47914802 -2.89892197 -2.65435106  0.20611138]
Final GD bias:  -0.6991186489435021
GD accuracy:  0.978125
```




```
Final SGD weights:  [-0.03539888 -0.02226838  0.00764083 -0.00120196]
Final SGD bias:  -0.006398546686490999
SGD accuracy:  0.8583333333333333
```



As for which feature is the most important, we can see that in the weight vector with learning rate 0.001, the largest weight vector by magnitude is feature 1. Therefore we can say that the model gives highest importance to feature 1 and it is the most important feature.