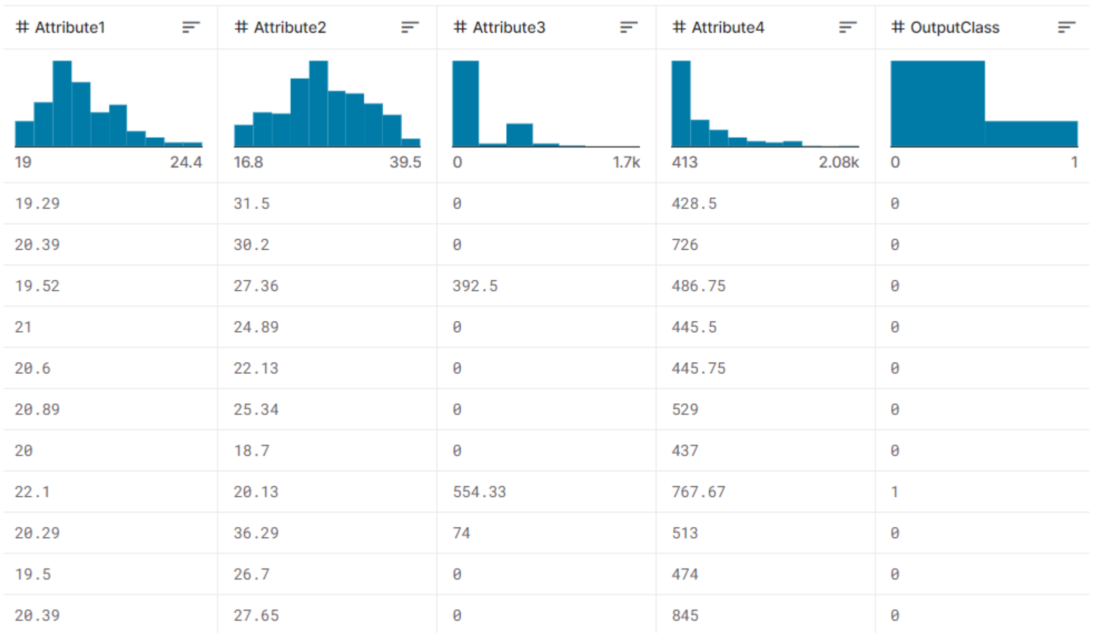
REPORT FOR LOGISTIC REGRESSION

This is a logistic regression model for binary classification. It reads a CSV file containing input data with two attributes and a target class label, and pre-processes the data by removing unwanted columns and splitting it into training and test sets.

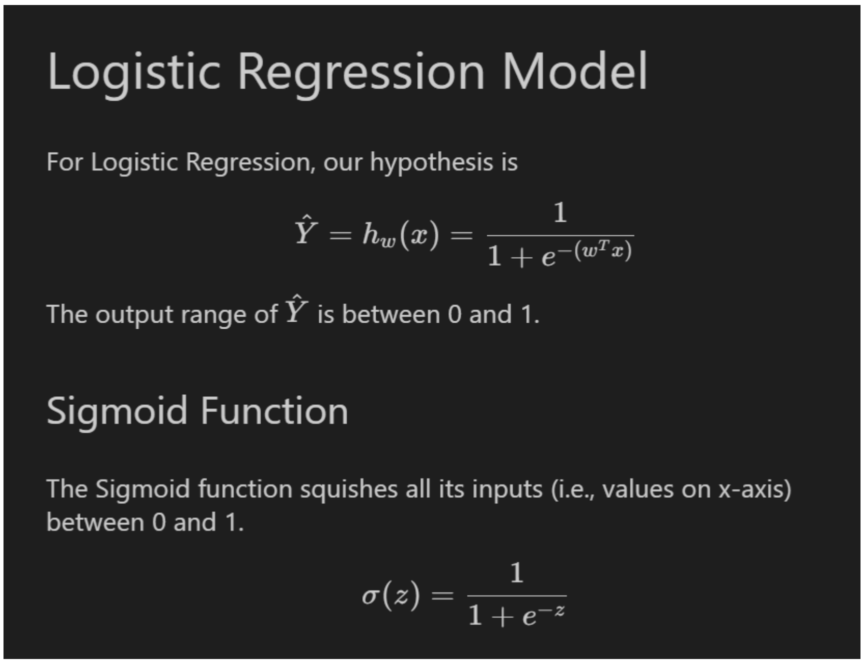

19
24.4
16.8
39.5
0
1.7k
413
2.08k
0
1

# Attribute1	# Attribute2	# Attribute3	# Attribute4	# OutputClass
19.29	31.5	0	428.5	0
20.39	30.2	0	726	0
19.52	27.36	392.5	486.75	0
21	24.89	0	445.5	0
20.6	22.13	0	445.75	0
20.89	25.34	0	529	0
20	18.7	0	437	0
22.1	20.13	554.33	767.67	1
20.29	36.29	74	513	0
19.5	26.7	0	474	0
20.39	27.65	0	845	0

The model then trains on the training set using gradient descent to optimize the logistic regression loss function. Once trained, it predicts the class label of test data and calculates the accuracy of the model. Finally, it plots the decision boundary and the loss vs. epoch graph.

The model contains the following functions:

1. sigmoid(z): computes the sigmoid of an input z.



Logistic Regression Model

For Logistic Regression, our hypothesis is

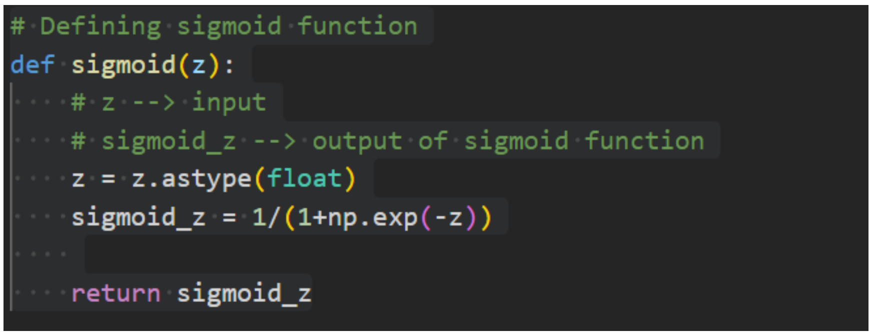
$$\hat{Y} = h_w(x) = \frac{1}{1 + e^{-(w^T x)}}$$

The output range of $\hat{Y}$ is between 0 and 1.

Sigmoid Function

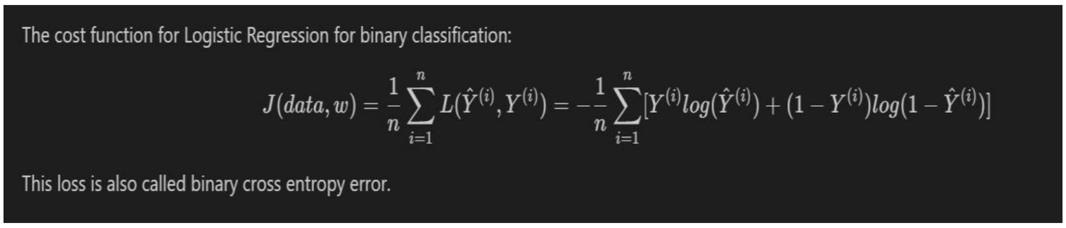
The Sigmoid function squishes all its inputs (i.e., values on x-axis) between 0 and 1.

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



```python
# Defining sigmoid function
def sigmoid(z):
    # z --> input
    # sigmoid_z --> output of sigmoid function
    z = z.astype(float)
    sigmoid_z = 1/(1+np.exp(-z))
    
    return sigmoid_z
```

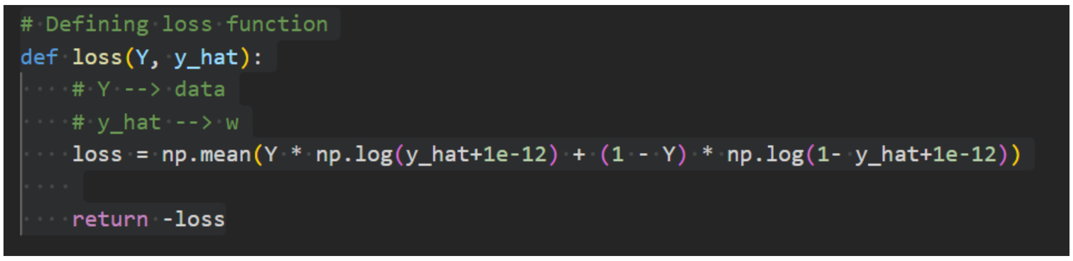
2. loss(Y, y_hat): computes the logistic regression loss.



The cost function for Logistic Regression for binary classification:

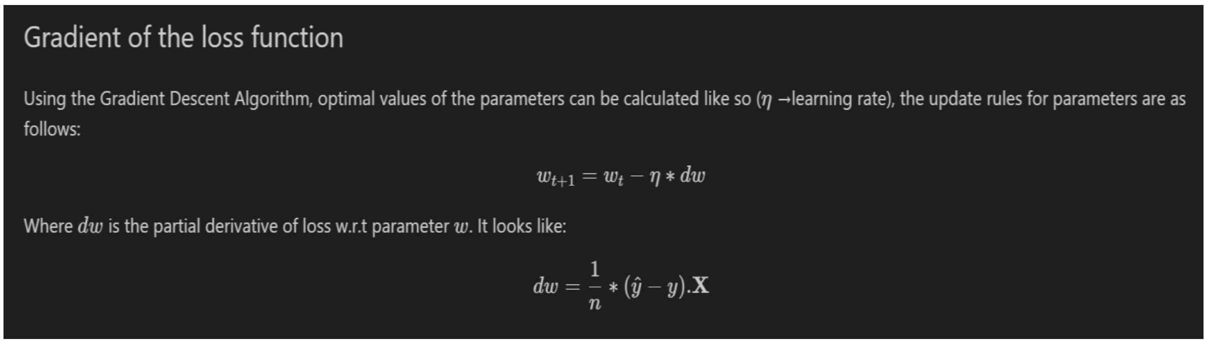
$$J(data, w) = \frac{1}{n}\sum_{i=1}^{n} L(\hat{Y}^{(i)}, Y^{(i)}) = -\frac{1}{n}\sum_{i=1}^{n}[Y^{(i)}log(\hat{Y}^{(i)}) + (1 - Y^{(i)})log(1 - \hat{Y}^{(i)})]$$

This loss is also called binary cross entropy error.



```python
# Defining loss function
def loss(Y, y_hat):
    # Y --> data
    # y_hat --> w
    loss = np.mean(Y * np.log(y_hat+1e-12) + (1 - Y) * np.log(1- y_hat+1e-12))
    
    return -loss
```

3. gradients(X, Y, y_hat): computes the gradients of the loss w.r.t the weights.



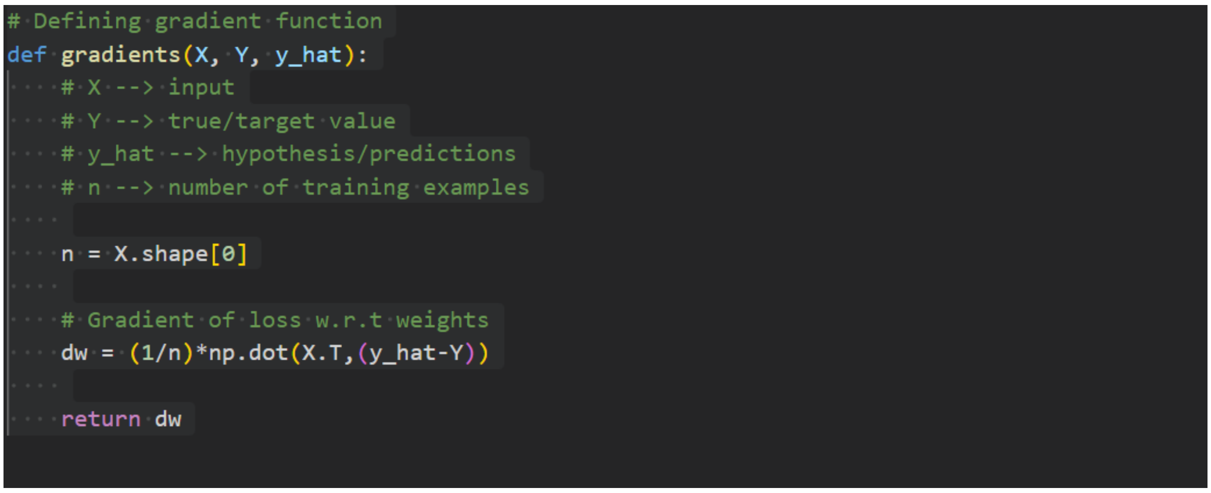
Gradient of the loss function

Using the Gradient Descent Algorithm, optimal values of the parameters can be calculated like so (η →learning rate), the update rules for parameters are as follows:

$$w_{t+1} = w_t - \eta * dw$$

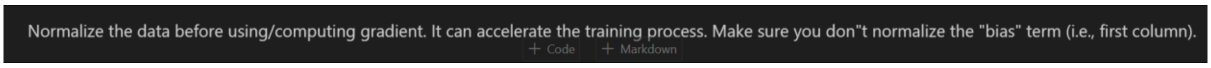
Where dw is the partial derivative of loss w.r.t parameter w. It looks like:

$$dw = \frac{1}{n} * (\hat{y} - y).\mathbf{X}$$

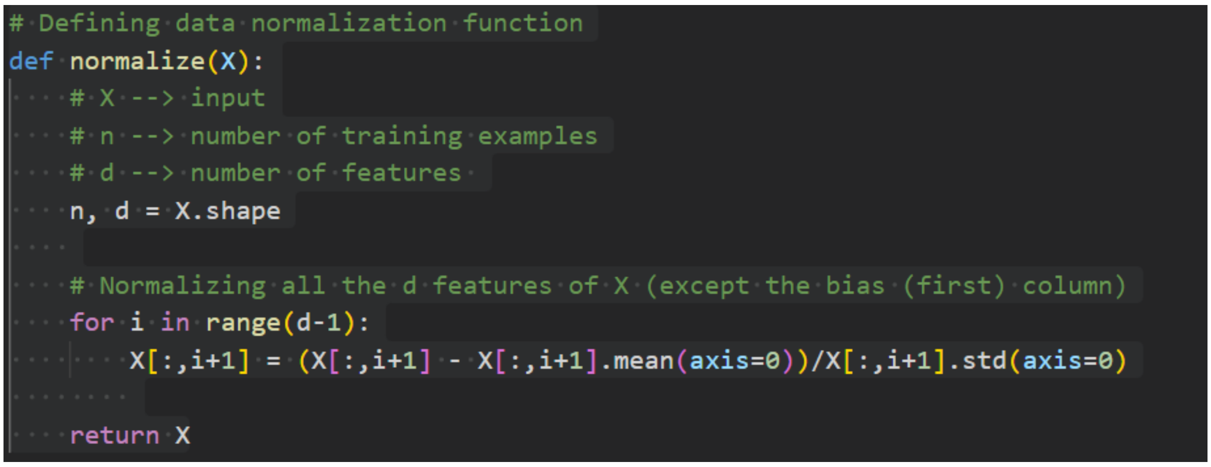


```python
# Defining gradient function
def gradients(X, Y, y_hat):
    # X --> input
    # Y --> true/target value
    # y_hat --> hypothesis/predictions
    # n --> number of training examples
    
    n = X.shape[0]
    
    # Gradient of loss w.r.t weights
    dw = (1/n)*np.dot(X.T,(y_hat-Y))
    
    return dw
```

4. normalize(X): normalizes the input data.

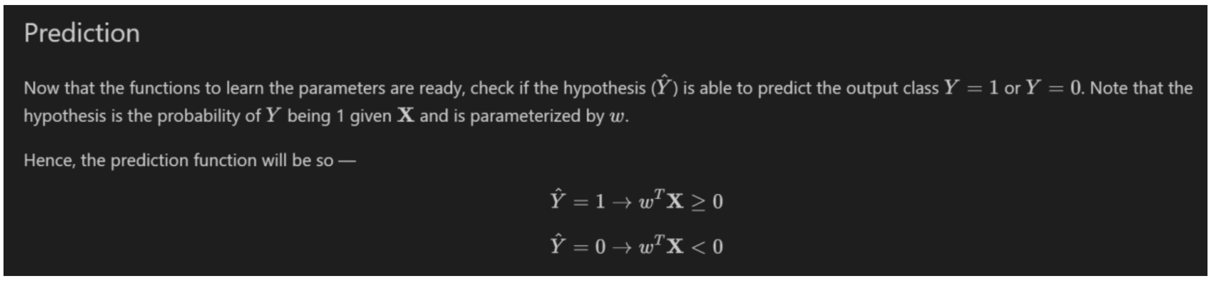


Normalize the data before using/computing gradient. It can accelerate the training process. Make sure you don"t normalize the "bias" term (i.e., first column).



```python
# Defining data normalization function
def normalize(X):
    # X --> input
    # n --> number of training examples
    # d --> number of features 
    n, d = X.shape
    
    # Normalizing all the d features of X (except the bias (first) column)
    for i in range(d-1):
        X[:,i+1] = (X[:,i+1] - X[:,i+1].mean(axis=0))/X[:,i+1].std(axis=0)
        
    return X
```

5. predict(X, w): predicts the class label of input data using the learned weights.



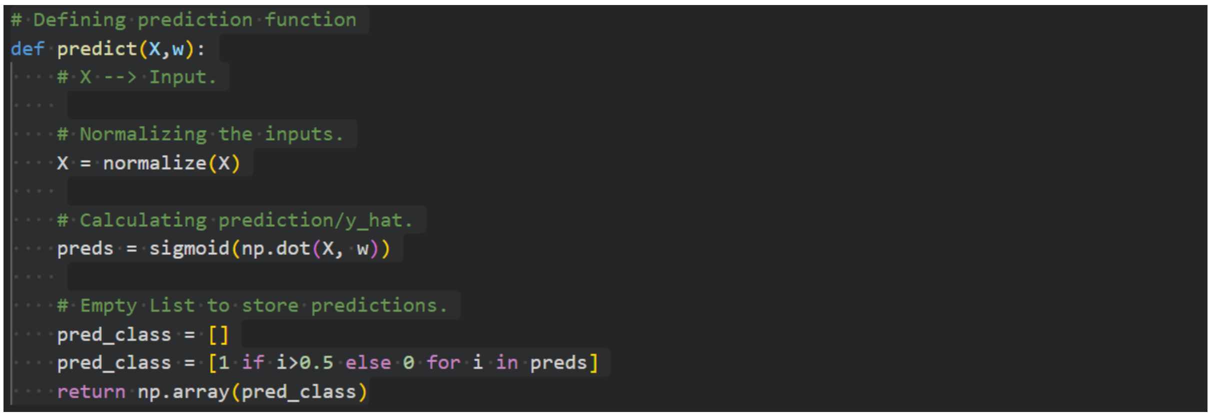
Prediction

Now that the functions to learn the parameters are ready, check if the hypothesis ($\hat{Y}$) is able to predict the output class $Y = 1$ or $Y = 0$. Note that the hypothesis is the probability of Y being 1 given $\mathbf{X}$ and is parameterized by w.

Hence, the prediction function will be so —

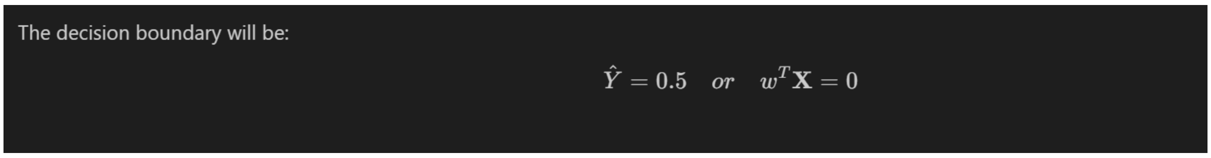
$$\hat{Y} = 1 \rightarrow w^T\mathbf{X} \geq 0$$

$$\hat{Y} = 0 \rightarrow w^T\mathbf{X} < 0$$



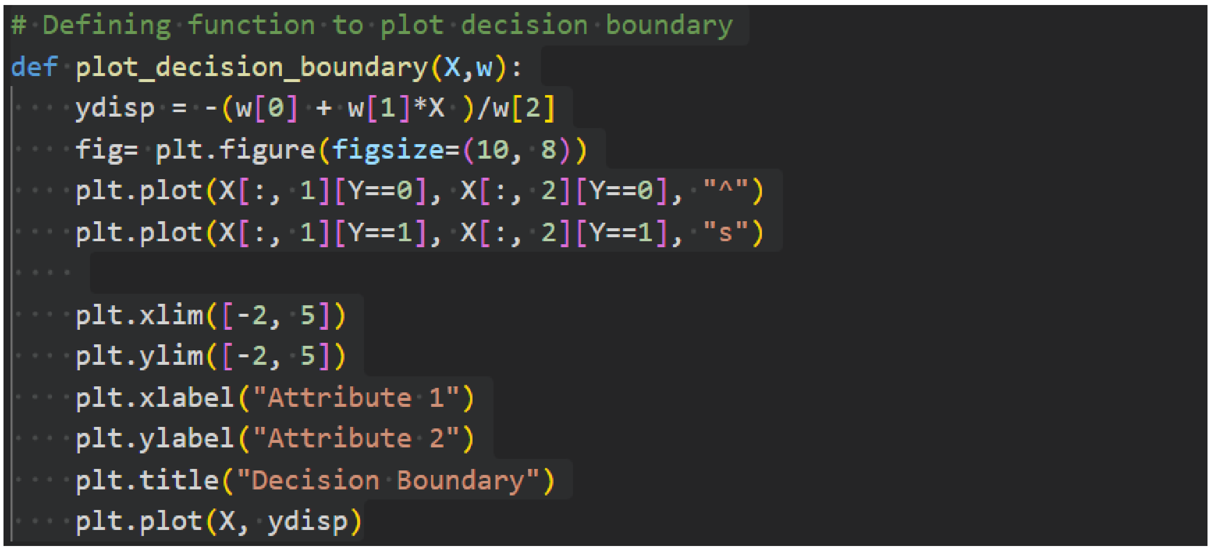
```python
# Defining prediction function
def predict(X,w):
    # X --> Input.
    
    # Normalizing the inputs.
    X = normalize(X)
    
    # Calculating prediction/y_hat.
    preds = sigmoid(np.dot(X, w))
    
    # Empty List to store predictions.
    pred_class = []
    pred_class = [1 if i>0.5 else 0 for i in preds]
    return np.array(pred_class)
```

6. plot_decision_boundary(X, w): plots the decision boundary for the input data.



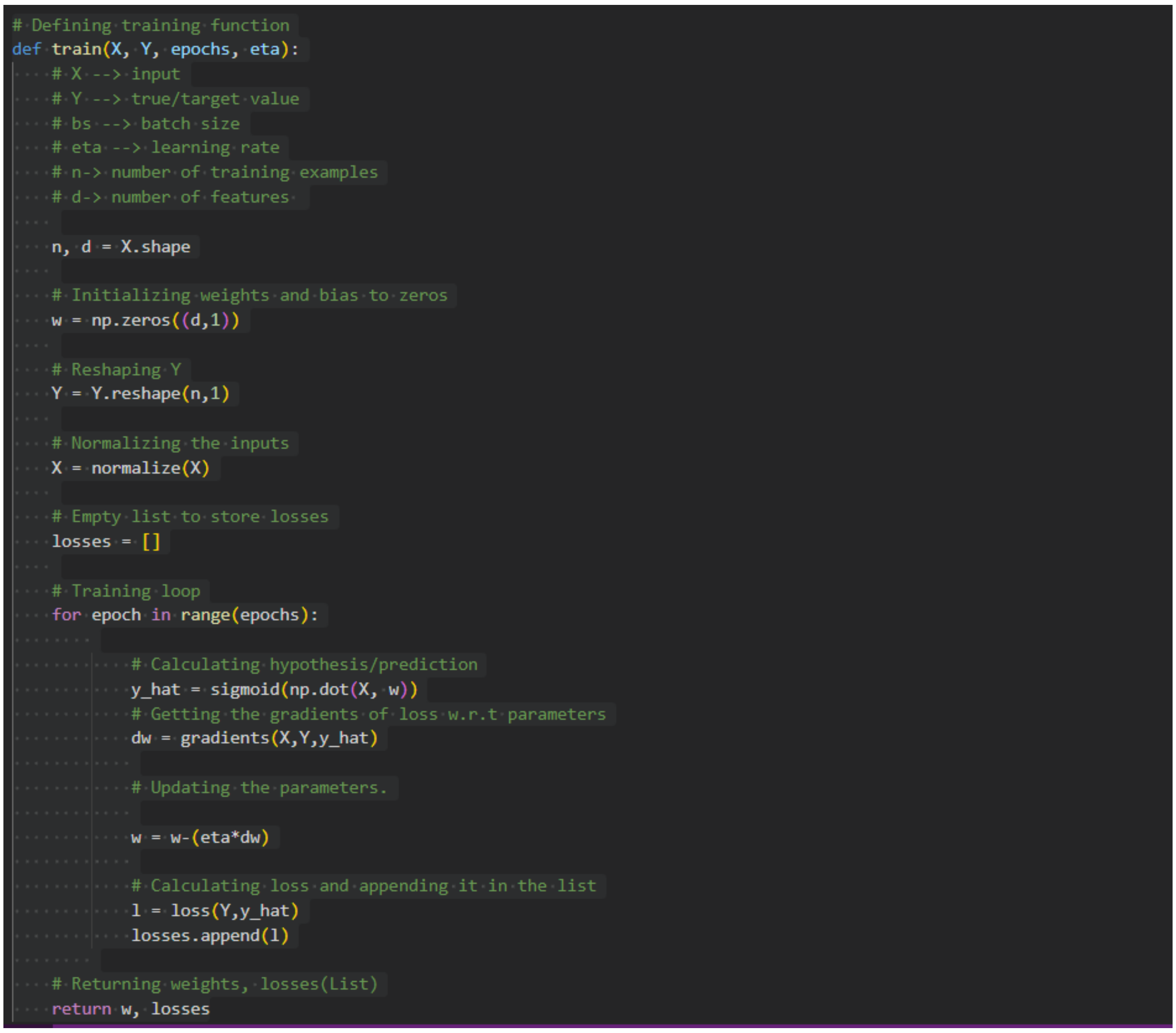
The decision boundary will be:

$$\hat{Y} = 0.5 \quad or \quad w^T\mathbf{X} = 0$$

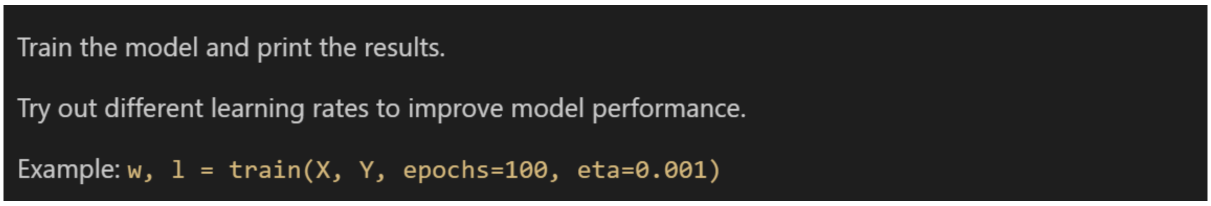


```python
# Defining function to plot decision boundary
def plot_decision_boundary(X,w):
    ydisp = -(w[0] + w[1]*X )/w[2]
    fig= plt.figure(figsize=(10, 8))
    plt.plot(X[:, 1][Y==0], X[:, 2][Y==0], "^")
    plt.plot(X[:, 1][Y==1], X[:, 2][Y==1], "s")
    
    plt.xlim([-2, 5])
    plt.ylim([-2, 5])
    plt.xlabel("Attribute 1")
    plt.ylabel("Attribute 2")
    plt.title("Decision Boundary")
    plt.plot(X, ydisp)
```

7. train(X, Y, epochs, eta): trains the logistic regression model on the input data using gradient descent. It returns the learned weights and the loss vs. epoch list.



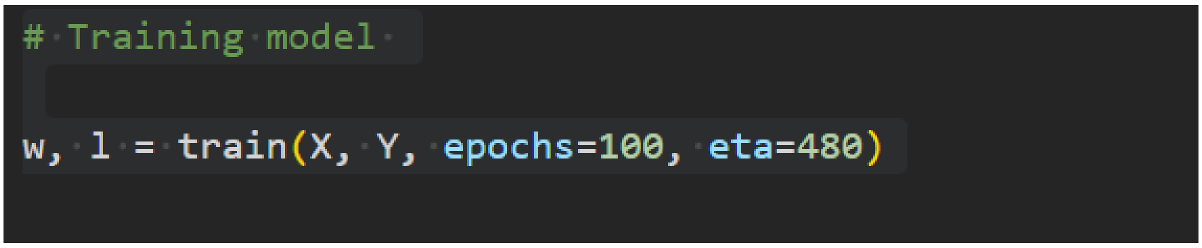
```python
# Defining training function
def train(X, Y, epochs, eta):
    # X --> input
    # Y --> true/target value
    # bs --> batch size
    # eta --> learning rate
    # n-> number of training examples
    # d-> number of features
    
    n, d = X.shape
    
    # Initializing weights and bias to zeros
    w = np.zeros((d,1))
    
    # Reshaping Y
    Y = Y.reshape(n,1)
    
    # Normalizing the inputs
    X = normalize(X)
    
    # Empty list to store losses
    losses = []
    
    # Training loop
    for epoch in range(epochs):
        
            # Calculating hypothesis/prediction
            y_hat = sigmoid(np.dot(X, w))
            # Getting the gradients of loss w.r.t parameters
            dw = gradients(X,Y,y_hat)
            
            # Updating the parameters.
            
            w = w-(eta*dw)
            
            # Calculating loss and appending it in the list
            l = loss(Y,y_hat)
            losses.append(l)
        
    # Returning weights, losses(List)
    return w, losses
```



Train the model and print the results.

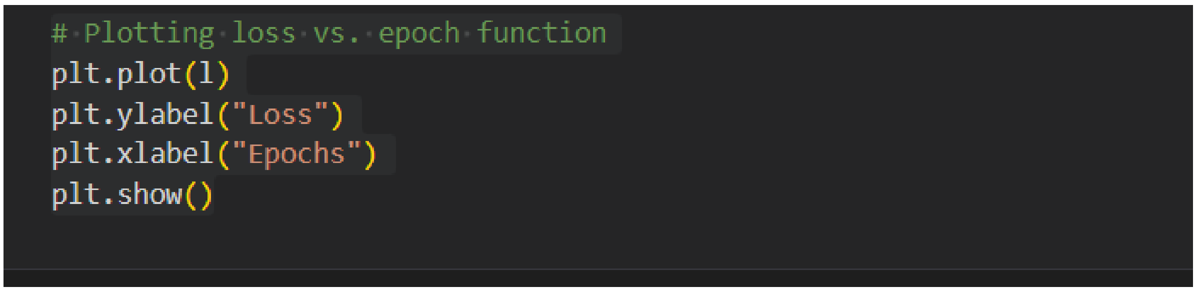
Try out different learning rates to improve model performance.

Example: `w, l = train(X, Y, epochs=100, eta=0.001)`

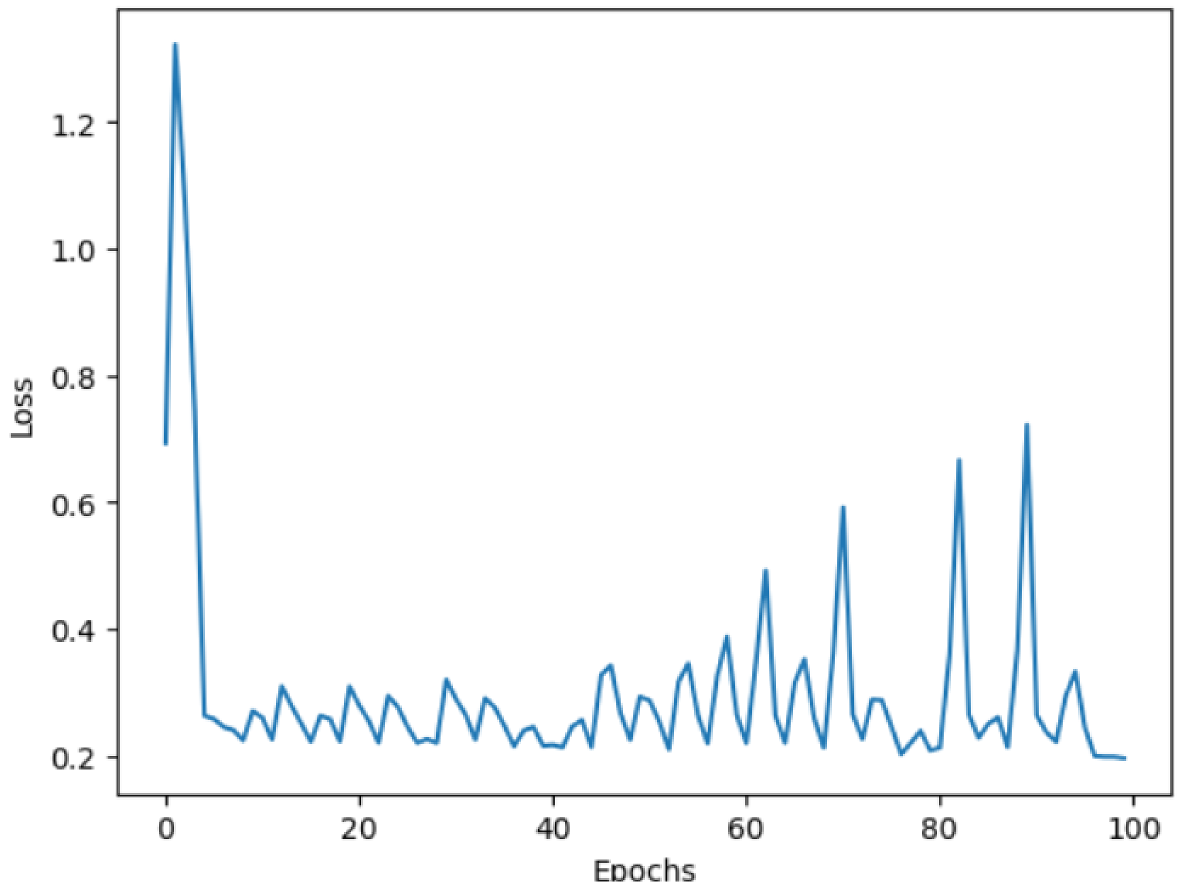


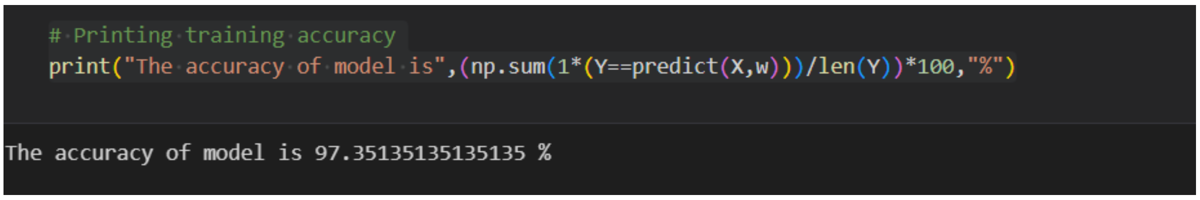
```python
# Training model

w, l = train(X, Y, epochs=100, eta=480)
```



```
# Plotting loss vs. epoch function
plt.plot(l)
plt.ylabel("Loss")
plt.xlabel("Epochs")
plt.show()
```

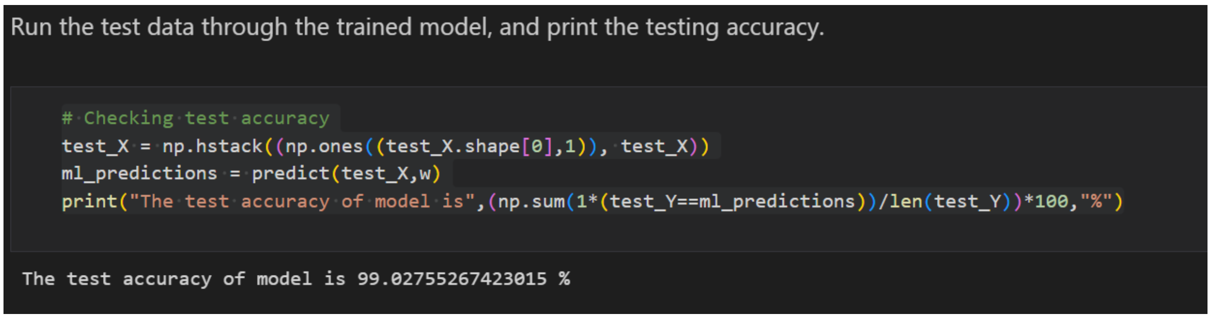




```
# Printing training accuracy
print("The accuracy of model is",(np.sum(1*(Y==predict(X,w)))/len(Y))*100,"%")
```

```
The accuracy of model is 97.35135135135135 %
```

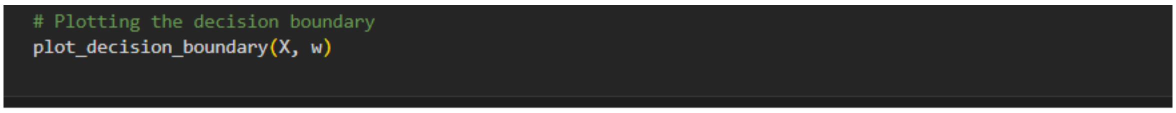
8. Using the trained model, predict the output class labels for the test set.
9. Calculate the accuracy of the model on the test set by comparing the predicted class labels with the true class labels.


Run the test data through the trained model, and print the testing accuracy.

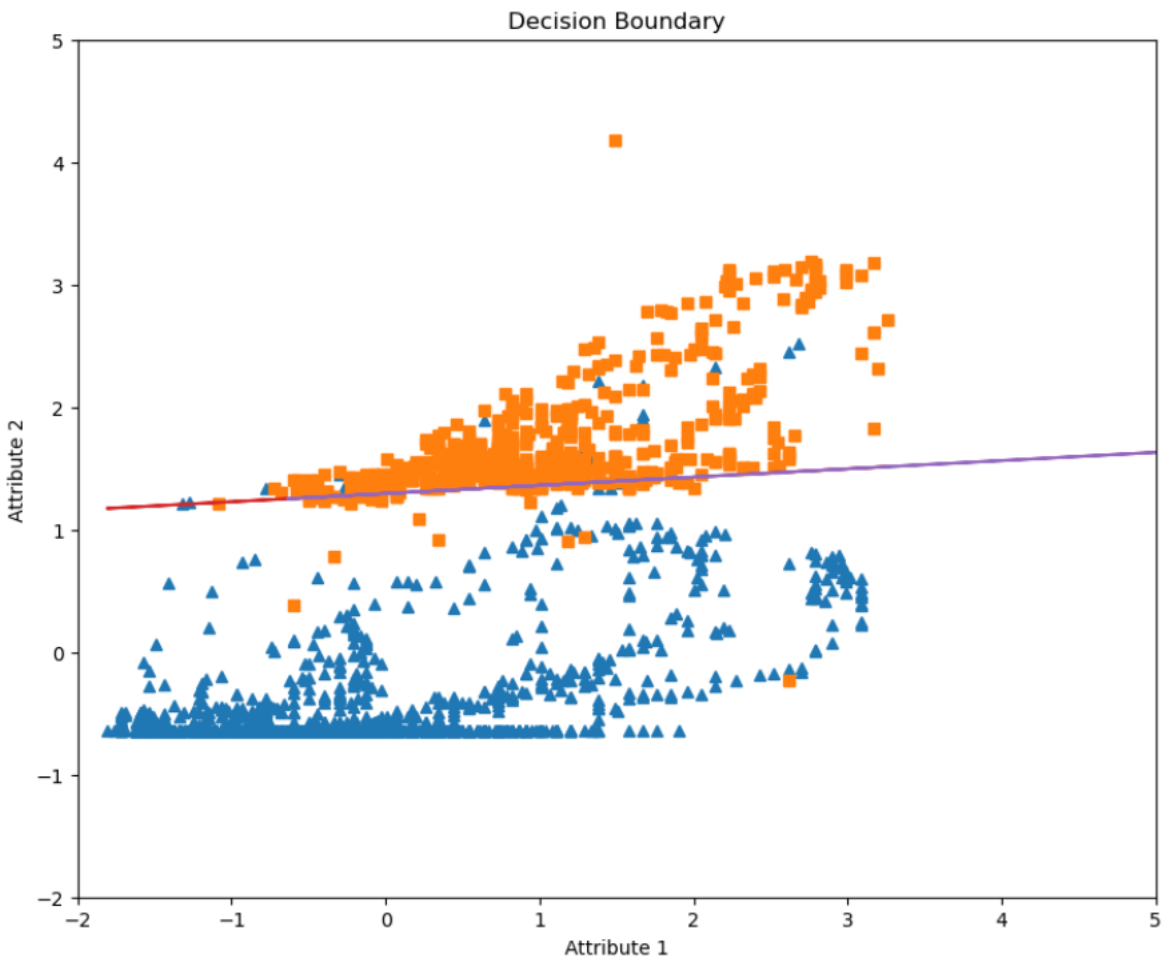
```
# Checking test accuracy
test_X = np.hstack((np.ones((test_X.shape[0],1)), test_X))
ml_predictions = predict(test_X,w)
print("The test accuracy of model is",(np.sum(1*(test_Y==ml_predictions))/len(test_Y))*100,"%")
```

```
The test accuracy of model is 99.02755267423015 %
```

10. Visualize the decision boundary by plotting the test set along with the decision boundary line, which is the line that separates the positive and negative classes.



```
# Plotting the decision boundary
plot_decision_boundary(X, w)
```



11. Interpret the results and draw conclusions.