AVINASH BABU (210797714)
a.babu@se21.qmul.ac.uk

ECS708 Machine Learning

Assignment 1: Part 2 – Logistic Regression and Neural Networks

1. Logistic Regression

Task 1

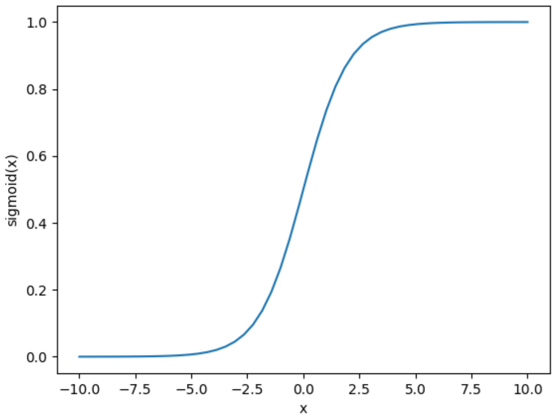
sigmoid.py

```
def sigmoid(z):
    output = 1 / (1 + np.exp(-z))
    return output
```

plot_sigmoid.py

```
def plot_sigmoid():
    x = np.linspace(1,2000)/100.0 - 10
    y = sigmoid(x)
    fig, ax1 = plt.subplots()
    ax1.plot(x, y)
    # set label of horizontal axis
    ax1.set_xlabel('x')
    # set label of vertical axis
    ax1.set_ylabel('sigmoid(x)')
    plt.show()
```

Running the *plot_sigmoid.py* generated the following result

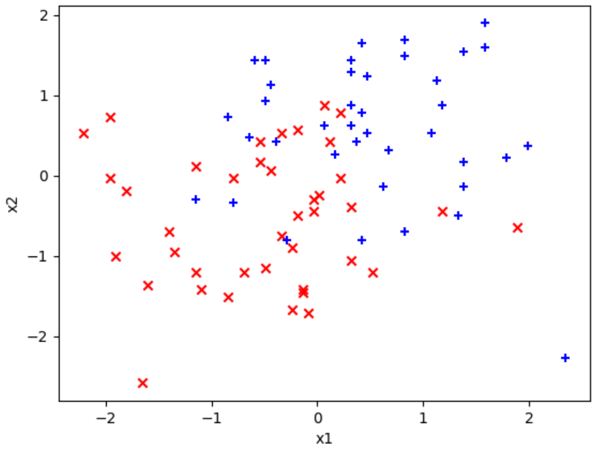


AVINASH BABU (210797714)
a.babu@se21.qmul.ac.uk

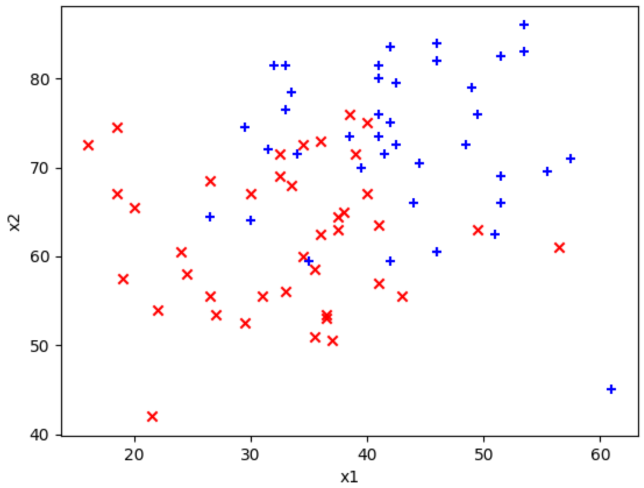
Task 2

Running *plot_data.py* to plot data

With Normalization:



Without Normalization:



1.1. Cost function and gradient for logistic regression

Task 3

calculate_hypothesis.py

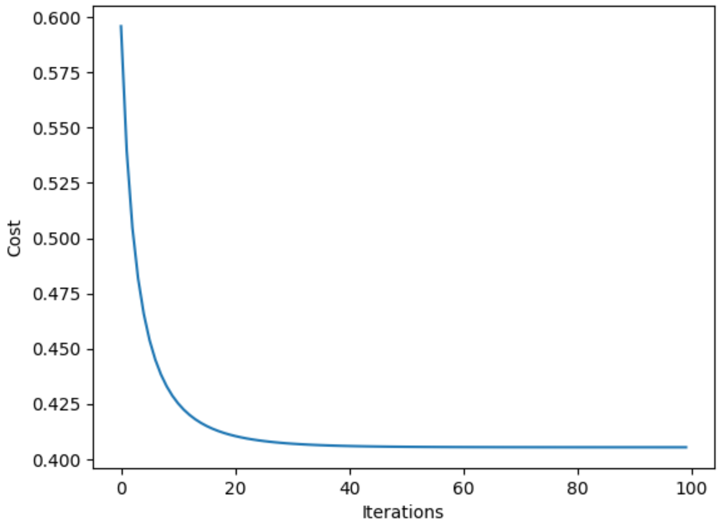
```
def calculate_hypothesis(X, theta, i):
    """
        :param X            : 2D array of our dataset
        :param theta        : 1D array of the trainable parameters
        :param i            : scalar, index of current training sample's
row
    """
    hypothesis = 0.0
    #Calculate the hypothesis for i-th sample of X, given X, theta and i.
    x = X[i]
    hypothesis = np.dot(x, theta)
    result = sigmoid(hypothesis)
    return result
```

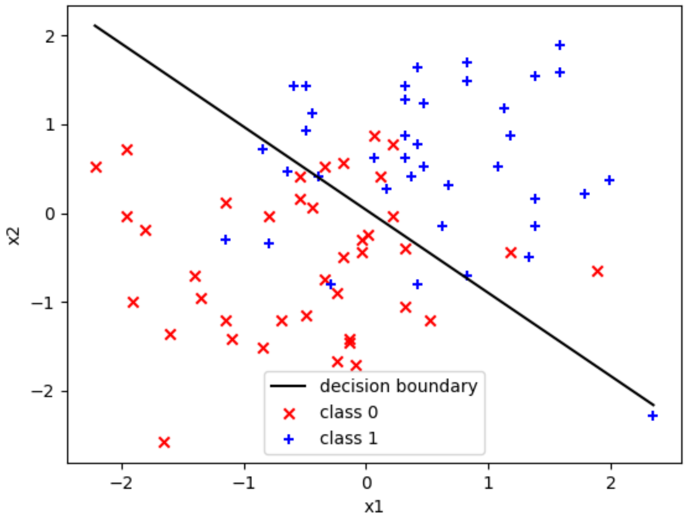
Task 4

compute_cost.py

```
def compute_cost(X, y, theta):
    """
        :param X            : 2D array of our dataset
        :param y            : 1D array of the groundtruth labels of the
dataset
        :param theta        : 1D array of the trainable parameters
    """

    # initialize cost
    J = 0.0
    # get number of training examples
    m = y.shape[0]

    # Compute cost for logistic regression.
    for i in range(m):
        hypothesis = calculate_hypothesis(X, theta, i)
        output = y[i]
        cost = 0.0
        #########################################
        # Write your code here
        # You must calculate the cost
        cost = - output * np.log(hypothesis) - (1 - output) * np.log(1 -
hypothesis)
        ########################################/
        J += cost
    J = J/m

    return J
```

Running the *assgn1_ex1.py with learning rate as 0.01 with 100 iterations produces the following graphs*





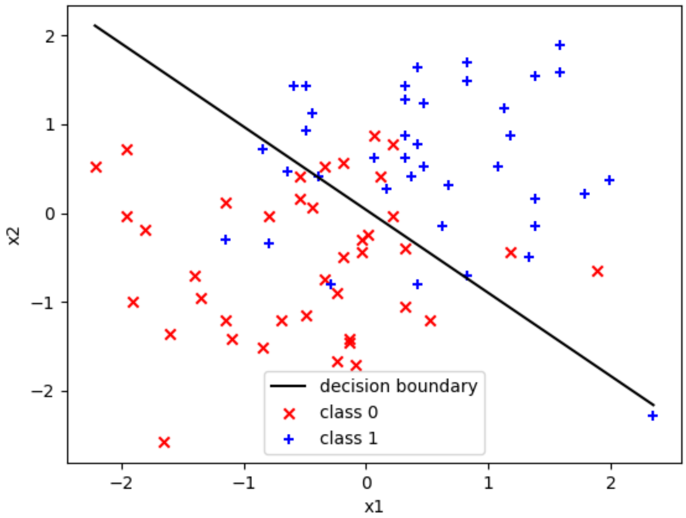
Minimum cost: 0.40545, on iteration #100

Task 5

plot_boundary.py

```
def plot_boundary(X, theta, ax1):

    min_x1 = 0.0
    max_x1 = 0.0
    x2_on_min_x1 = 0.0
    x2_on_max_x1 = 0.0

    # Re-arrange the terms in the equation of the hypothesis function, and
solve with respect to x2, to find its values on given values of x1
    min_x1 = X[:,1].min()
    max_x1 = X[:,1].max()
    x2_on_min_x1 = (- theta[0] - theta[1] * min_x1)/theta[2]
    x2_on_max_x1 = (- theta[0] - theta[1] * max_x1)/theta[2]

    x_array = np.array([min_x1, max_x1])
    y_array = np.array([x2_on_min_x1, x2_on_max_x1])
    ax1.plot(x_array, y_array, c='black', label='decision boundary')

    # add legend to the subplot
    ax1.legend()
```

Decision boundary plot we get by running ml_assgn1_ex1.py is



Task 6

S No	Final training cost	Final test cost	Difference
1	0.11398	0.80046	0.68648
2	0.31403	0.49864	0.18461
3	0.25598	0.60232	0.34634
4	0.18400	0.75385	0.56985
5	0.42778	0.41957	-0.00821
6	0.38726	0.51443	0.12717

AVINASH BABU (210797714)
a.babu@se21.qmul.ac.uk

2. Neural Network

Task 10

NeuralNetwork.py

```
class NeuralNetwork():
    def __init__(self,
                 n_in,
                 n_hidden,
                 n_out):
        """
        :param n_in      : number of elements of each input sample
        :param n_hidden  : number of hidden states
        :param n_out     : number of output states
        """
        self.n_in = n_in
        self.n_hidden = n_hidden
        self.n_out = n_out

        # Initialize weight matrices of the hidden layer and the output 
layer.
        # Both layers will have a bias term, hence we need to add one more 
weight.
        self.w_hidden = np.random.rand(n_in + 1, n_hidden)
        self.w_out = np.random.rand(n_hidden + 1, n_out)

        # initialize the states of the hidden neurons and the output 
neurons
        self.y_hidden = np.zeros((n_hidden))
        self.y_out = np.zeros((n_out))

    def reset_activations(self):
        # reset neuron activations
        self.y_hidden.fill(0)
        self.y_out.fill(0)

    def forward_pass(self, inputs):
        # We will calculate the output(s), by feeding the inputs forward 
through the network

        # If a forward pass has occured before (i.e., bias term has been 
appended to y_hidden), then we have to remove the bias from the hidden 
neurons
        if len(self.y_hidden)==(self.n_hidden+1):
            self.y_hidden = self.y_hidden[1:]

        # set hidden states and output states to zero
        self.reset_activations()

        # append term to be multiplied with the hidden layer's bias
        inputs = np.append(1, inputs)
```

a.babu@se21.qmul.ac.uk


```
        # activate hidden neurons
        for i in range(self.n_hidden):
            hidden_neuron = 0.0
            for j in range(len(inputs)):
                hidden_neuron += + inputs[j] * self.w_hidden[j,i]
            self.y_hidden[i] = sigmoid(hidden_neuron)

        # append term to be multiplied with the output layer's bias
        self.y_hidden = np.append(1.0, self.y_hidden)

        # activate output neurons
        for i in range(self.n_out):
            output_neuron = 0.0
            for j in range(len(self.y_hidden)):
                output_neuron += self.y_hidden[j] * self.w_out[j,i]
            self.y_out[i] = sigmoid(output_neuron)

        predictions = self.y_out.copy()

        return predictions

    def backward_pass(self, inputs, targets, learning_rate):
        # We will backpropagate the error and perform gradient descent on
the network weights

        # We compute the error between predictions and targets
        J = 0.5 * np.sum( np.power(self.y_out - targets, 2) )

        # append term that was multiplied with the hidden layer's bias
        inputs = np.append(1, inputs)

        # Step 1. Output deltas are used to update the weights of the
output layer
        output_deltas = np.zeros((self.n_out))
        outputs = self.y_out.copy()

        for i in range(self.n_out):
            #########################################
            # Write your code here
            # compute output_deltas : delta_k = (y_k - t_k) * g'(x_k)
            if np.isscalar(targets):
                output_deltas[i] = (outputs[i] - targets) *
sigmoid_derivative(outputs[i])
            else:
                output_deltas[i] = (outputs[i] - targets[i]) *
sigmoid_derivative(outputs[i])

            #########################################/

        # Step 2. Hidden deltas are used to update the weights of the
hidden layer
        hidden_deltas = np.zeros((len(self.y_hidden)))

        # Create a for loop, to iterate over the hidden neurons.
```

```
        # Then, for each hidden neuron, create another for loop, to iterate
over the output neurons
        for i in range(len(hidden_deltas)):
            #########################################
            # Write your code here
            # compute hidden_deltas
            hidden_layout_error = 0
            for j in range(len(output_deltas)):
                hidden_layout_error += self.w_out[i, j] * output_deltas[j]
            hidden_deltas[i] = sigmoid_derivative(self.y_hidden[i]) *
hidden_layout_error
            #########################################/

        # Step 3. update the weights of the output layer
        for i in range(len(self.y_hidden)):
            for j in range(len(output_deltas)):
                #########################################
                # Write your code here
                # update the weights of the output layer
                self.w_out[i,j] = self.w_out[i,j] - (learning_rate *
output_deltas[j] * self.y_hidden[i])
                #########################################/

        # we will remove the bias that was appended to the hidden neurons,
as there is no
        # connection to it from the hidden layer
        # hence, we also have to keep only the corresponding deltas
        hidden_deltas = hidden_deltas[1:]

        # Step 4. update the weights of the hidden layer
        # Create a for loop, to iterate over the inputs.
        # Then, for each input, create another for loop, to iterate over
the hidden deltas
        for i in range(len(inputs)):
            for j in range(len(hidden_deltas)):
                #########################################
                # Write your code here
                # update the weights of the hidden layer
                self.w_hidden[i,j] = self.w_hidden[i,j] - (learning_rate *
hidden_deltas[j] * inputs[i])
                #########################################/

        return J
```