Queen Mary, University of London
School of Electronic Engineering and Computer Science

ECS708U: Machine Learning

Assignment 1 (Part 2) - Logistic Regression & Neural Network

Mughees Asif | 180288337

Due: 12 October 2021

1 Logistic Regression

sigmoid.py:

```
# Sigmoid function
output = 1 / (1 + np.exp(-z))

```

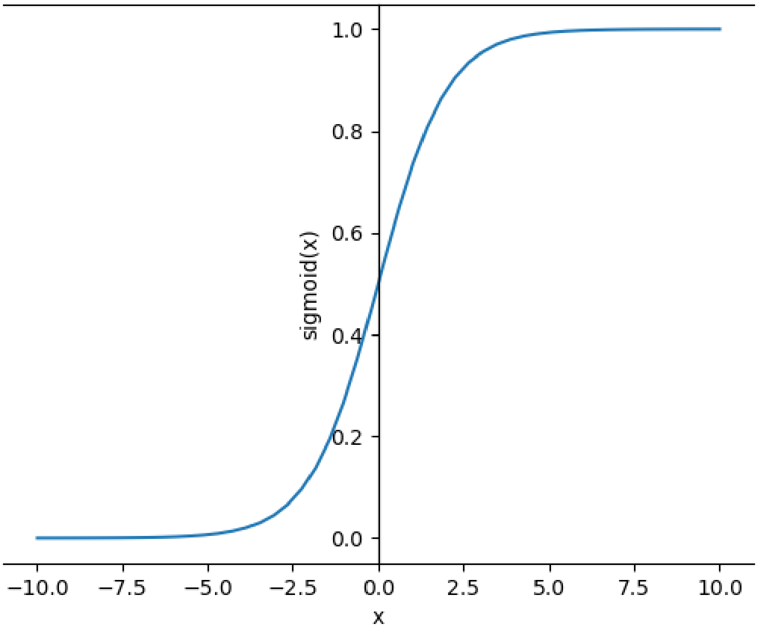
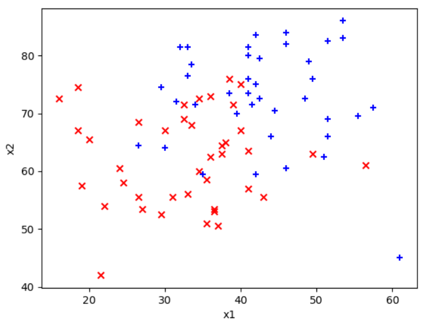
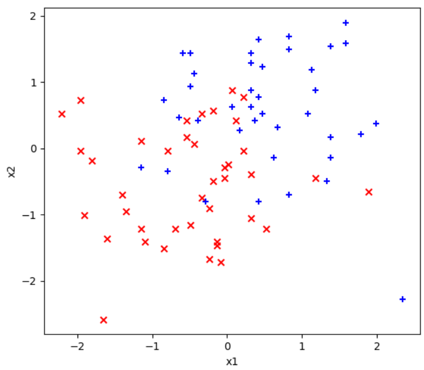


Figure 1: Sigmoid function

Normalized data:



(a) Without normalization



(b) Normalized

Cost function and gradient for logistic regression:

```python
# Hypothesis calculation
hypothesis = np.dot(X[i], theta)
result = sigmoid(hypothesis)

```

```python
# Cost formula
cost = (-output * np.log(hypothesis)) - (1 - output) * np.log(1 -
    hypothesis)

```

```
Minimum cost: 0.40545. on iteration #100
```

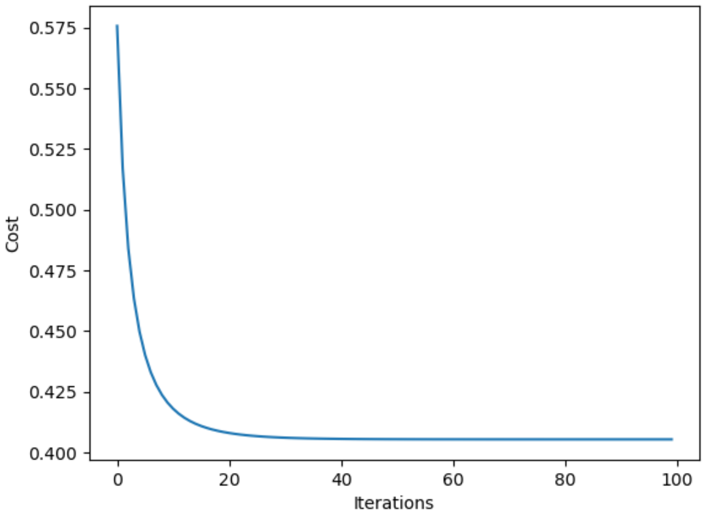
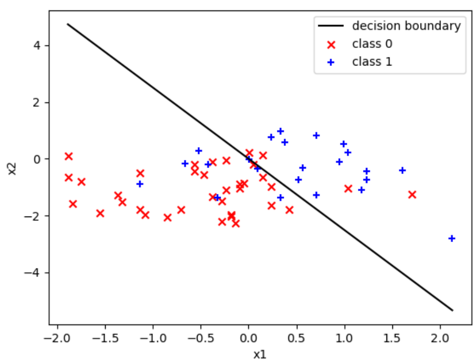
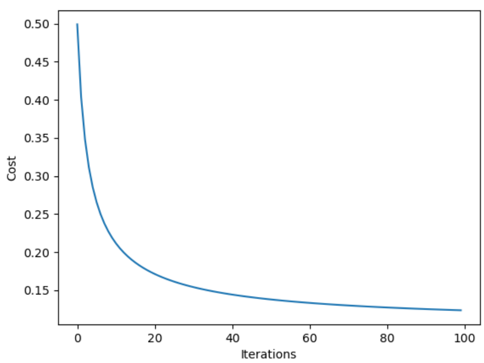


Figure 3: Cost against no. of iterations

```
Minimum cost: 0.40545, on iteration #100
alpha = 1
Final training cost: 0.12380
Minimum training cost: 0.12380, on iteration #100
Final test cost: 0.75521
```



(a) Logistic regression



(b) Cost against no. of iterations

Non-linear features and overfitting:

Sample #1:

```
Final training cost: 0.38781
Minimum training cost: 0.38781, on iteration #100
Final test cost: 0.42052
```

Sample #2:

```
Final training cost: 0.48428
Minimum training cost: 0.48428, on iteration #100
Final test cost: 0.41373
```

Sample #3:

```
Final training cost: 0.22024
Minimum training cost: 0.22024, on iteration #100
Final test cost: 0.58369
```

Sample #4:

```
Final training cost: 0.46901
Minimum training cost: 0.46901, on iteration #100
Final test cost: 0.43196
```

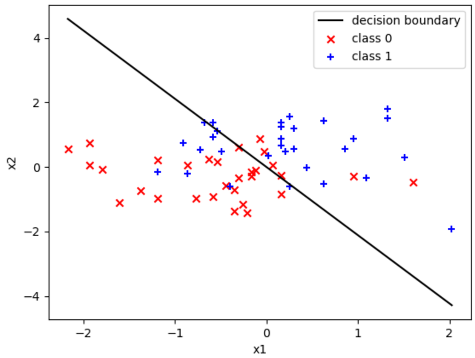
Sample #5:

```
Final training cost: 0.15906
Minimum training cost: 0.15906, on iteration #100
Final test cost: 0.74446
```

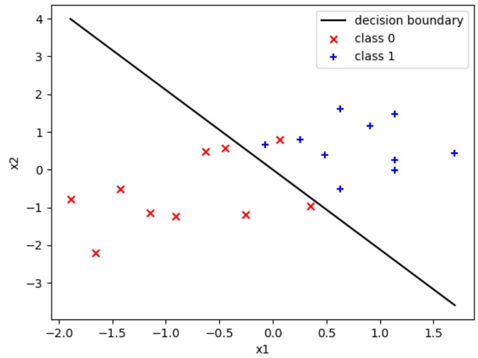
The training cost is generally similar to the test cost. However, for some instances, both values do diverge significantly indicating a poor fit due to the random shuffling.

Bad generalization:

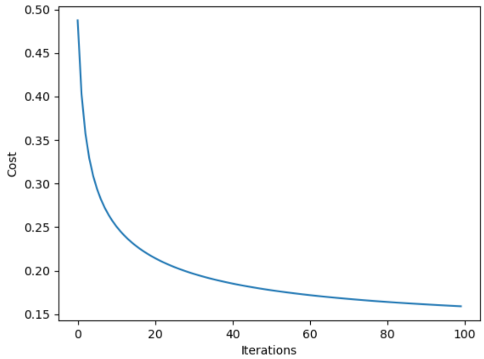
```
Final training cost: 0.15906
Minimum training cost: 0.15906, on iteration #100
Final test cost: 0.74446
```



(a) Original dataset



(b) Split dataset

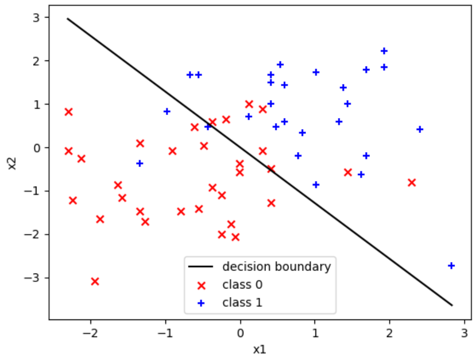


(c) Cost against no. of iterations

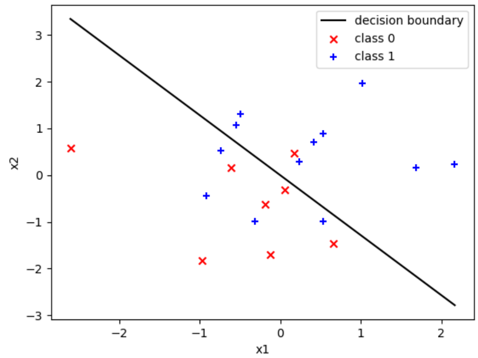
Figure 5: Bad generalisation

Good generalization:

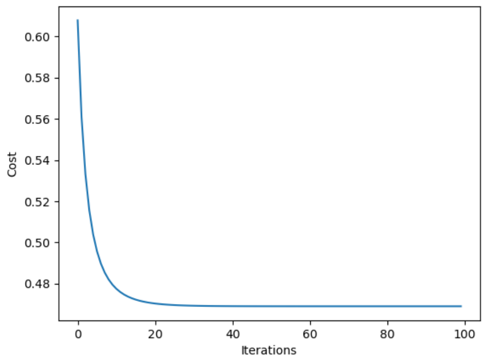
```
Final training cost: 0.46901
Minimum training cost: 0.46901, on iteration #100
Final test cost: 0.43196
```



(a) Original dataset



(b) Split dataset



(c) Cost against no. of iterations

Figure 6: Good generalisation

Random shuffling improves the generalisation capabilities of the model. Fig. 5 displays bad generalisation as the training and test cost values are hugely divergent which indicates a poor fit. Comparatively, Fig. 6 shows good generalisation reinforced by a similar training and test cost.

5D input vector per data point:

```
        # Insert extra singleton dimension, to obtain Nx1 shape
        x1 = X[:, 0, None]
        x2 = X[:, 1, None]
        # Create the features x1*x2, x1^2 and x2^2
        features = [x1*x2, x1**2, x2**2]
        # Append columns of the new features to the dataset, to the
        ↪  dimensionof columns (i.e., 1)
        for i in range(len(features)):
            X = np.append(X, features[i], axis=1)

        ...

        # Initialise trainable parameters theta
        theta = np.zeros(6)

```

```
Final cost: 0.40261
Minimum cost: 0.40261, on iteration #100
```

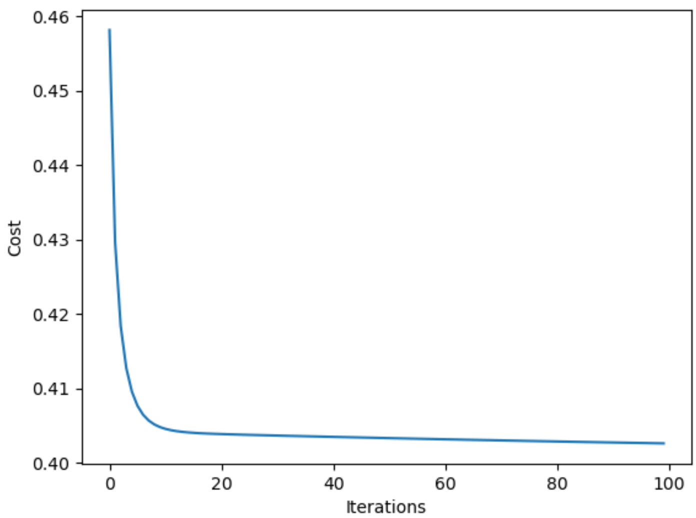
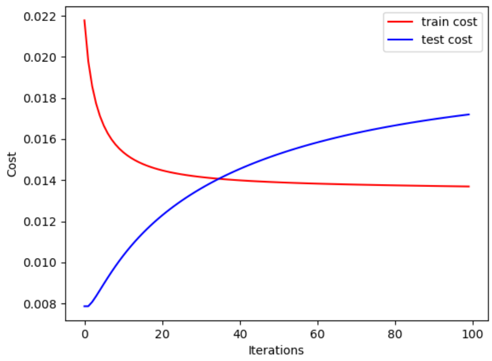


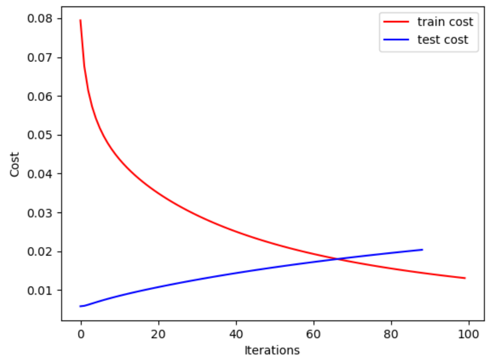
Figure 7: New cost against no. of iterations

The error is slightly (0.7%) less when using more features. There are *more* labels while the number of samples has remained *constant* which results in overfitting and poor generalization.

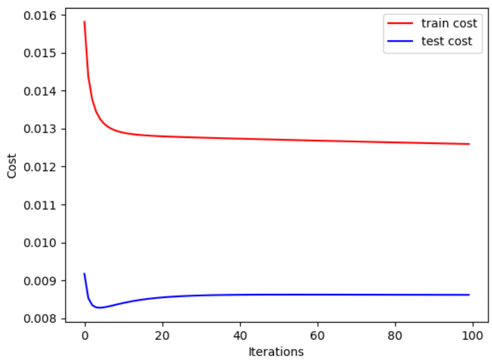
Various `train:test` splits:



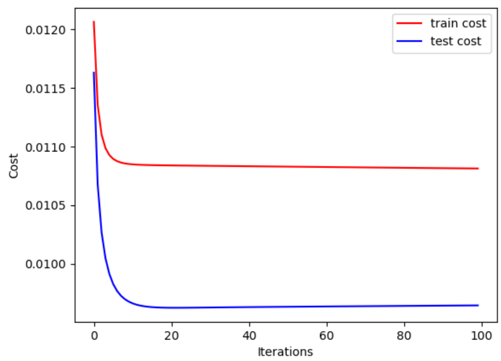
(a) 20:60



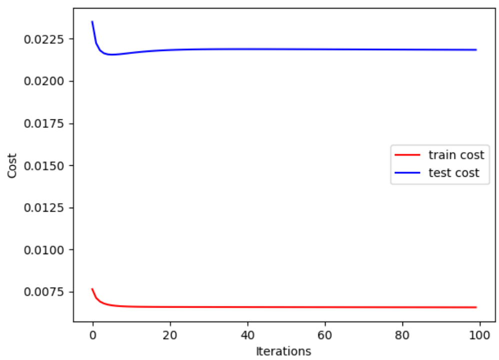
(b) 30:50



(c) 40:40



(d) 50:30



(e) 60:20

Evidently, the best split is around the **50:30** ratio as the `train:test` is closer together and the `test cost` is also low.

Effect of second-order and third-order polynomials:

```python
        # Insert extra singleton dimension, to obtain Nx1 shape
        x1 = X[:, 0, None]
        x2 = X[:, 1, None]
        # Create the features x1*x2, x1^2, x2^2 and x2^3
        features = [x1*x2, x1**2, x2**2, x2**3]
        # Append columns of the new features to the dataset, to the
        ↪  dimensionof columns (i.e., 1)
        for i in range(len(features)):
            X = np.append(X, features[i], axis=1)

        ...

        # Initialise trainable parameters theta
        theta = np.zeros(8)

```

Increasing the features without changing the data size increases the test cost resulting in overfitting.

XOR problem:

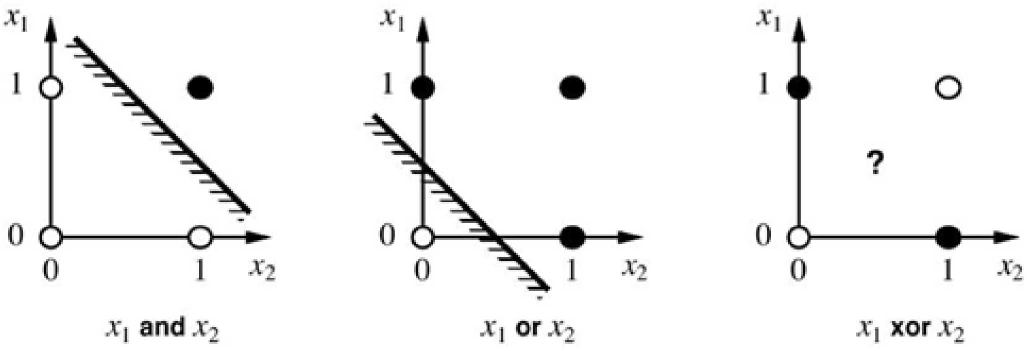


Figure 9: Linear separability[1]

The premise of the problem lies in predicting the output of XOR logic when given non-linearly separable inputs for classification in a single perceptron. Logistic regression can not be used on a XOR problem as it is a binary classification method capable of differentiating between one class e.g., sun/rain, pass/fail, and life/death. This would not be suitable to differentiate between two different classes that are present within a XOR decision space as seen in Fig. 9.

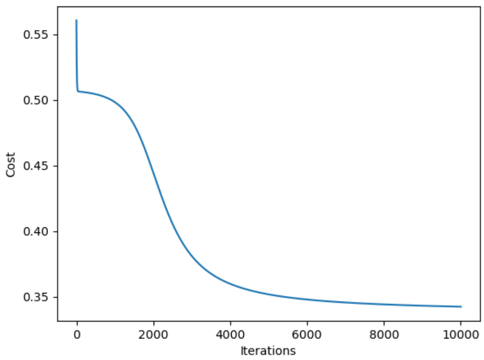
[1] Chandradevan, R., 2017. Radial Basis Functions Neural Networks — All we need to know. [online] Towards Data Science. Available at: <Link> [Accessed 26 October 2021].

2 Neural Network

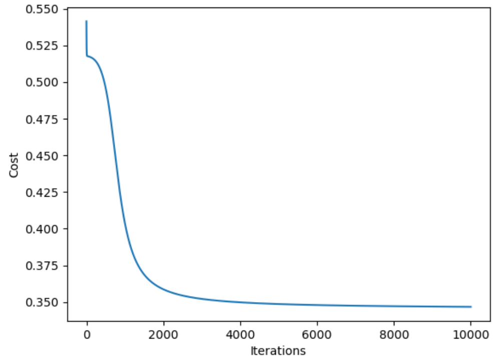
Backpropagation:

```
    # Step 1. Output deltas are used to update the weights of the output layer
    for i in range(self.n_out):
        # compute output_deltas : delta_k = (y_k - t_k) * g'(x_k)
        output_deltas[i] = (outputs - targets) * sigmoid_derivative(outputs)

    # Step 2. Hidden deltas are used to update the weights of the hidden layer
    for i in range(len(hidden_deltas)):
        # compute hidden_deltas
        delta_weight = 0
        for j in range(len(output_deltas)):
            delta_weight = delta_weight + self.w_out[i, j] * output_deltas[j]

        hidden_deltas[i] = sigmoid_derivative(self.y_hidden[i]) * delta_weight

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

    # Step 4. update the weights of the hidden layer
    for i in range(len(inputs)):
        for j in range(len(hidden_deltas)):
            # update the weights of the hidden layer
            self.w_hidden[i, j] = self.w_hidden[i, j] - learning_rate * hidden_deltas[j] * inputs[i]

```

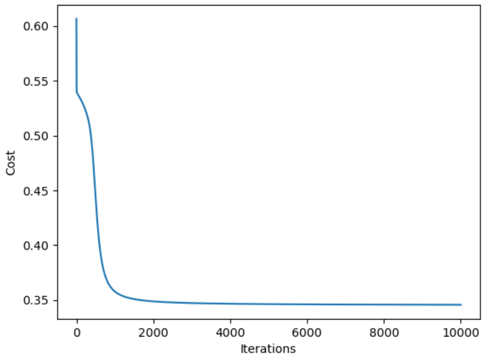
Effect of learning rate α:



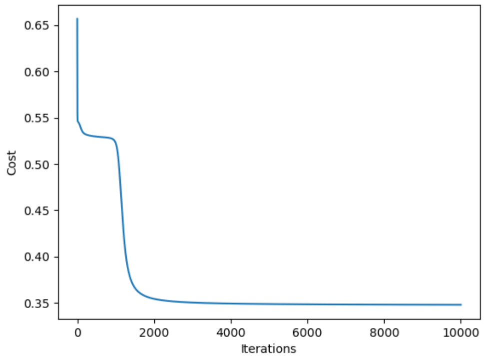
(a) $\alpha = 0.2$



(b) $\alpha = 0.5$



(c) $\alpha = 1$



(d) $\alpha = 1.2$

Figure 10: Cost vs. iterations for various α

An optimum learning rate exists between the interval [0, 1] as the search for the global optima continues within the bounds (highlighted in Fig. 10 by the fluctuating functions above).

Local optima problem:

```
Sample #01 | Target value: 0.00 | Predicted value: 0.48530
Sample #02 | Target value: 1.00 | Predicted value: 0.50067
Sample #03 | Target value: 1.00 | Predicted value: 0.50420
Sample #04 | Target value: 0.00 | Predicted value: 0.51470
```

When the learning rate is low, the predicted value for each target value is $\approx 50\%$ less accurate as the gradient descent does not converge to the global optima fast enough.

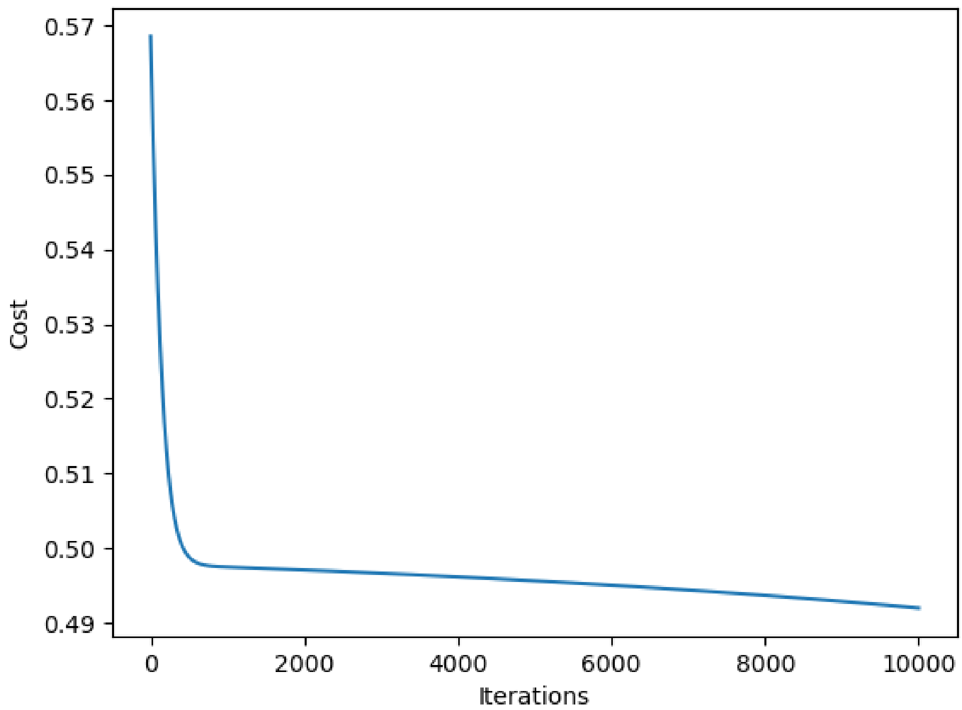


Figure 11: Graphical representation of local optima

AND gate:

```
Sample #01 | Target value: 0.00 | Predicted value: 0.00741
Sample #02 | Target value: 0.00 | Predicted value: 0.03110
Sample #03 | Target value: 0.00 | Predicted value: 0.03125
Sample #04 | Target value: 1.00 | Predicted value: 0.95437
Minimum cost: 0.00204, on iteration #10000
```

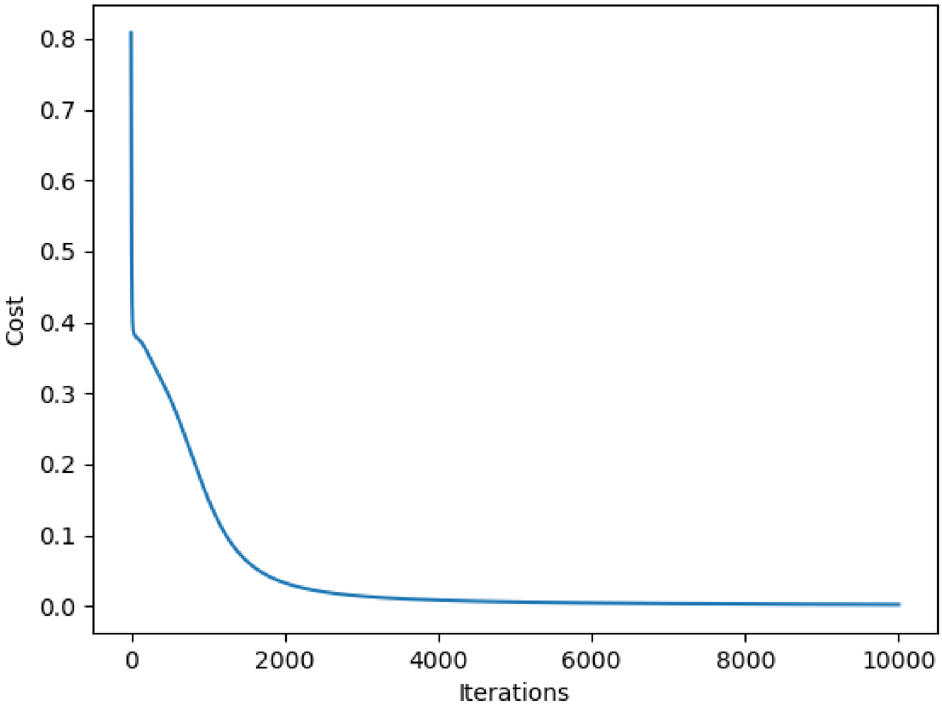
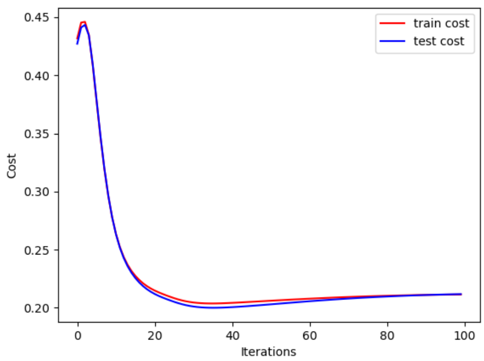


Figure 12: Cost graph for the AND gate

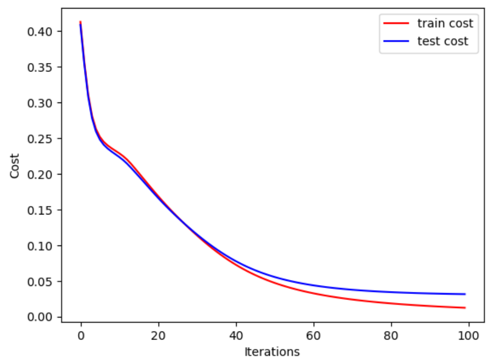
Logistic regression unit for three classes:

Logistic regression is a generalized linear model which can not capture complex features of the dataset. For multi-class problems, multinomial logistic regression can be used to discern which category the dependent variable belongs to. Since, this problem requires classifying between three species of flowers, a probability distribution generated by executing logistic regression on each class can be obtained. This distribution can then be used to choose the desired class.

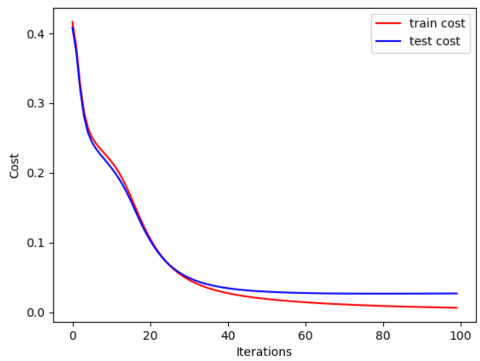
<u>Effect of changing the number of hidden neurons</u>:



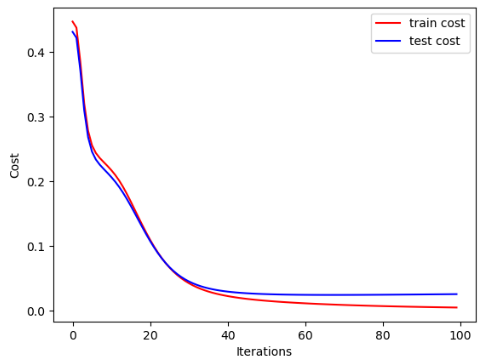
(a) 1



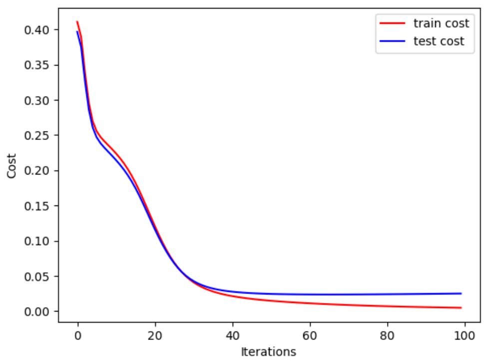
(b) 2



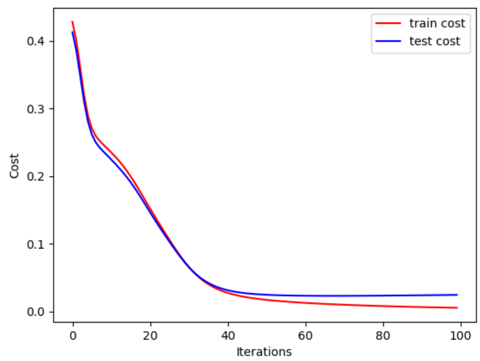
(c) 3



(d) 5



(e) 7



(f) 10

Figure 13: Changing the number of indicated hidden layers

From Fig. 13, the training and test cost diverge around the 2 neuron mark. The minimum cost reduces as the number of neurons is increased.