CS671 – Deep Learning and Applications

Assignment 1

Implementing Perceptron for Classification and Regression

to be submitted by

Group – 13

Members:

Prachi Sharma (V21078)

Ayush Tiwari (T22053)

Single neuron model – Perceptron

The perceptron model takes an input vector **x** = [x_1 x_2 … x_d] along with weight **w** = [w_1 w_2 … w_n] as input and gives an activation value

$$a = w^T . x + w_0$$

A *sigmoidal activation function* is then applied on **a** as,

$$s = f(a) = \frac{1}{1+e^{-a}}$$

Here, we are using a logistic activation function which will give values in the range [0,1].

The neuron can finally classify two classes according to the rule,

- If $s < 0.5$, **x** belongs to class 1
- If $s \geq 0.5$, **x** belongs to class 2

The objective of the neuron is to learn a linear hyperplane that can separate the two classes with minimum error. To do so, we use the *gradient descent method*.

Gradient Descent Algorithm

Gradient descent method is used to minimize the error between the actual and predicted output of the model. To guide our algorithm, we define an *instantaneous error function* as

$$E_n = \sum_{k=1}^{K} \left(y_n - \hat{y}_n\right)^2$$

The gradient points in the direction of the steepest ascent so we tend to move in the direction opposite to the gradient to reach local minima of the error surface.

$$\Delta w \propto -\frac{\delta E_n}{\delta w}$$

We can compute the gradient as,

$$\frac{\delta E_n}{\delta w} = \eta\left(y_n - s_n\right)\frac{\delta f\left(a_n\right)}{\delta a_n} x_n$$

In case of logistic activation function,

$$\frac{\delta f\left(a_n\right)}{\delta a_n} = s_n\left(1 - s_n\right)$$

Here, η is the learning rate that regulates the convergence of our learning algorithm. In general, $0 < \eta \leq 1$.

Task 1: Classification

Model: Perceptron with sigmoidal activation function for each of the datasets. Use one-against-one approach for 3-class classification. Implement gradient descent method for perceptron learning algorithm.

Training Algorithm:

(1) Initialise weight vector with random values.
(2) Choose a training pattern and concatenate 1 (for bias term).

$$\hat{x}_n = \begin{bmatrix} 1 & x_{n0} & x_{n1} \end{bmatrix}$$

(3) Compute dot product $a = w^T . x + w_0$ and apply logistic activation function to compute s_n.
(4) Compute instantaneous error, E_n
(5) Update weights as

$$w = w + \Delta w$$

(6) Repeat steps 2 to 5 till all training patterns are presented.
(7) Calculate average error, $E_{av} = \frac{1}{N} \sum_{n=1}^{N} E_n$
(8) Repeat steps 2 to 7 till convergence criterion is satisfied.

Testing Algorithm:

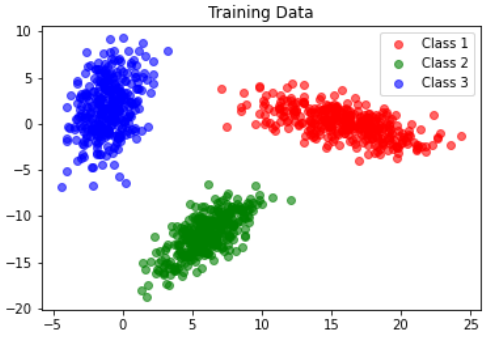
(1) For a given test example, compute $a = w^T . x + w_0$ using final weights and bias.
(2) Apply logistic activation function to compute s_n.
(3) Classify x to class 1 if output is less than 0.5, otherwise to class 2.

Results

Linearly separable classes:

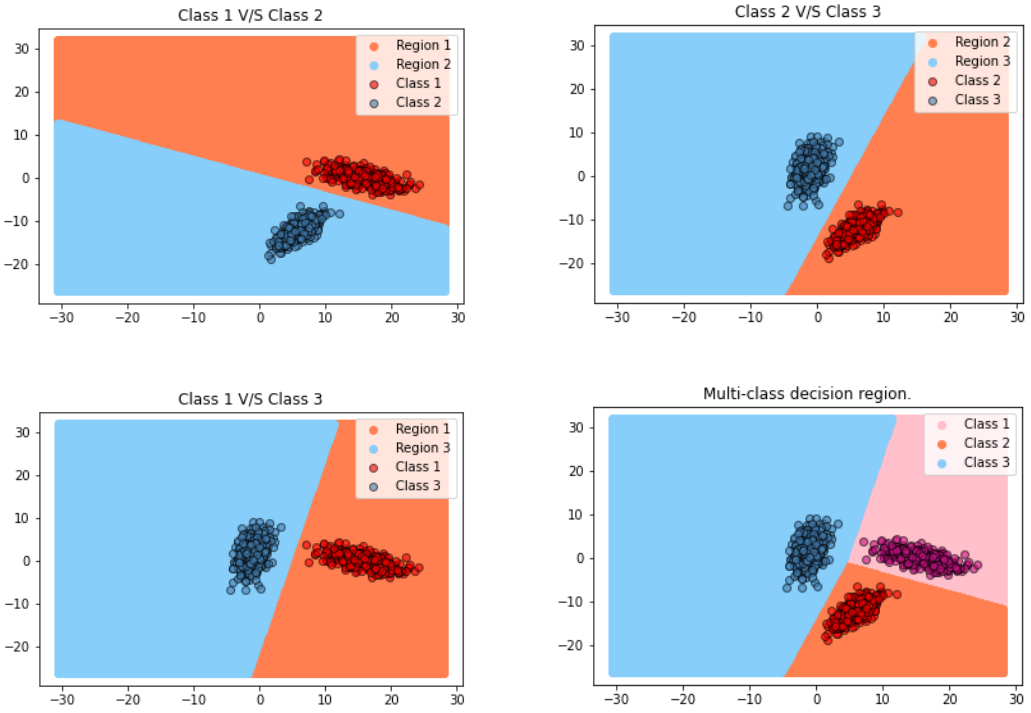
3 classes, 2-dimensional linearly separable data is given. Each class has 500 data points. Divide the data from each class into training, and test data. From each class, train, and test split should be 70% and 20% respectively.

Scatter plot of data points



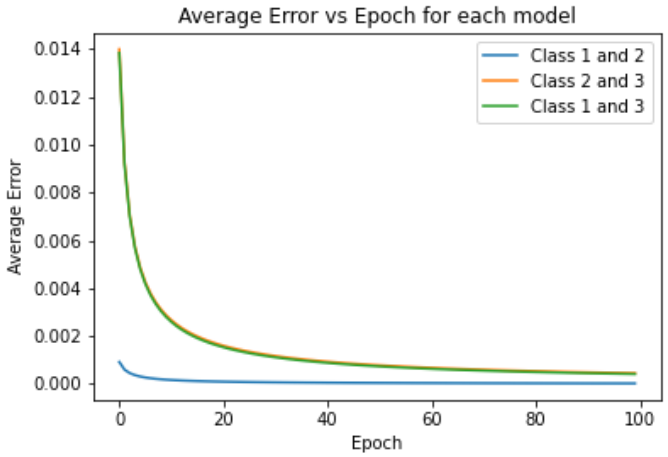
The data has been classified using a one-to-one approach and the outputs are then combined for a complete decision boundary. The training data has been superimposed on the decision region.

Decision region plot



It can be observed through the plots and other performance measures that the model classifies the data perfectly. The boundary between classes is the hyperplane that the model has learnt during training.

Average error v/s epoch plot



It can be observed that the average error decreases exponentially as the number of epochs increases.

Performance measures:

- Confusion matrix:

```
Confusion Matrix for 3-class classification
[[300   0   0]
 [  0 300   0]
 [  0   0 300]]
```

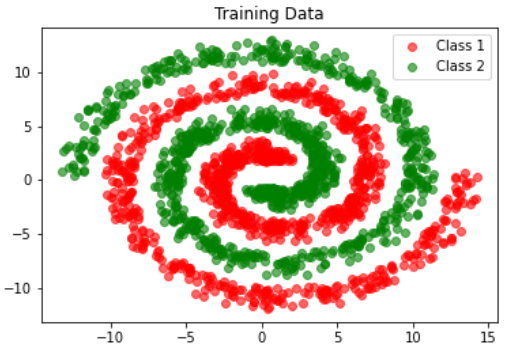
- `Accuracy of model:  100.0`
- `Average precision:  100.0`
- `Average recall:  100.0`
- `Average F-measure:  1.0`

The performance measures have been calculated using the test data. The results suggests that the classification is perfect.

Nonlinearly separable classes:

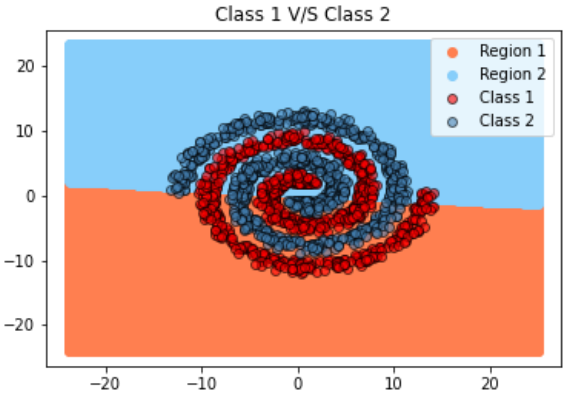
2-dimensional data of 2 or 3 classes that are non linearly separable. The number of examples in each class and their order is given at the beginning of each file. Divide the data from each class into training, and test data. From each class, train, and test split should be 70% and 20% respectively.

Scatter plot of data points



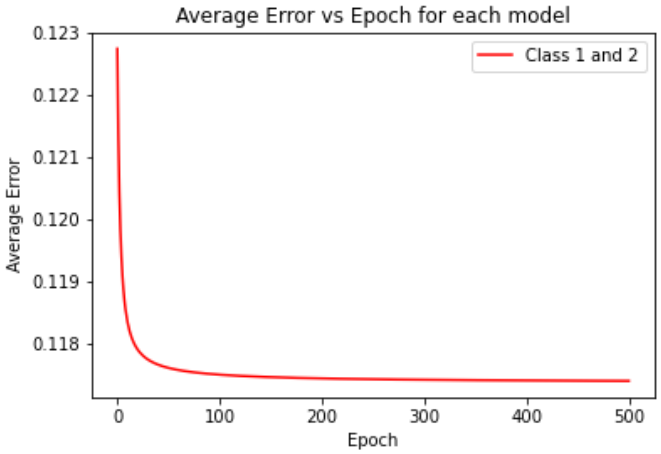
The data has been classified using a one-to-one approach and the outputs are then combined for a complete decision boundary. The training data has been superimposed on the decision region.

Decision region plot



It can be observed that the single neuron model is not a good choice for classifying non linearly separable data.

Average error v/s epoch plot



It can be observed that the average error decreases exponentially as the number of epochs increases.

Performance measures:

- Confusion matrix:

```
Confusion Matrix for 2-class classification for non linearly
separable classes
[[198 193]
 [189 202]]
```

- `Accuracy of model:  51.150895140664964`
- `Precision of class 1:  50.63938618925832`
  `Precision of class 2:  51.66240409207161`
- `Recall for class 1: 51.162790697674424`
  `Recall for class 2: 51.13924050632911`
- `F-measure for class1:  0.5089974293059126`
  `F-measure for class2:  0.5139949109414759`

The performance measures have been calculated using the test data. The results suggest that the model performance is poor and (almost) half of data is classified incorrectly.

Conclusion:

After using the single neuron model on both linearly separable and non linearly separable data, we can infer that the model works pretty well for linearly separable data, on the other hand, works poorly for non linearly separable data. This is because the single neuron model learns a linear hyperplane and a linear hyperplane cannot separate non linear data.

Task 2: Regression

Problem: Given two type of dataset, one with univariate dimensions and other with bivariate dimensions, we have to perform linear regression on it. The expected output is parameters of line (in case of univariate data) and the parameters of a hyperplane (in case of bivariate data).

Data preprocessing:

1. Divided the dataset into two parts, 70% for training and 30% for testing, for each of the univariate and bivariate case.
2. Append 1 to the x-values for including bias in the Weight matrix.

Training:

The train_model function takes 5 parameters namely epochs, initial_weight, x, y, threshold. The threshold is defaulted to 0. If no value of threshold is provided, then the model will train for all the epochs mentioned. If a threshold is being provided, then the training stops as soon as the absolute difference of current loss and previous loss falls below the threshold.
It returns the MSE loss for all the epochs and the weight matrix obtained.

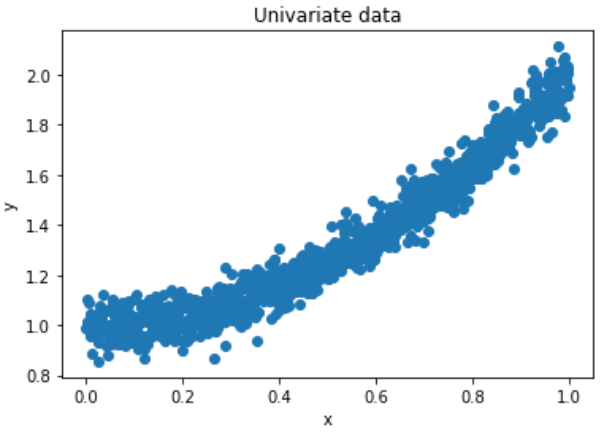
Algorithm:

1. Define loss list and model_weight (with default value provided)
2. Run a loop from 1 to number of epochs
    a. Define learning rate to be 1/i
    b. Iterate through all the training samples one by one
        i. Calculate the predicted output using weight and X
        ii. Append predicted output to a list for cost calculation
        iii. Apply gradient_descent on the weight and update it with new weight
    c. Calculate MSE for the epoch, using the actual y and the predicted output vector and append it to the loss vector initialized above.
    d. Check for threshold criteria, if it is met then stop the iterations and return weight along with loss

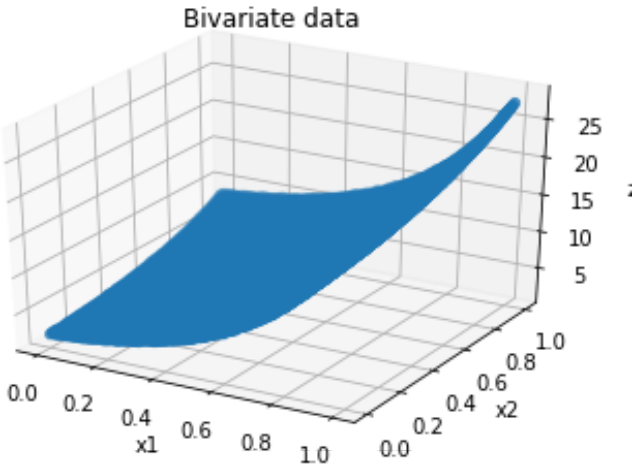
Testing algorithm:

1. Get the weights from each of the models by using the training algorithm defined above.
2. Iterate over all the data points in the test data set
    a. With the weights, call the predictOutput function, which calculates the dot product with Weight and the test data point.
    b. Calculate error with the predicted output and actual output for plotting and visualization

Initial Data plots:

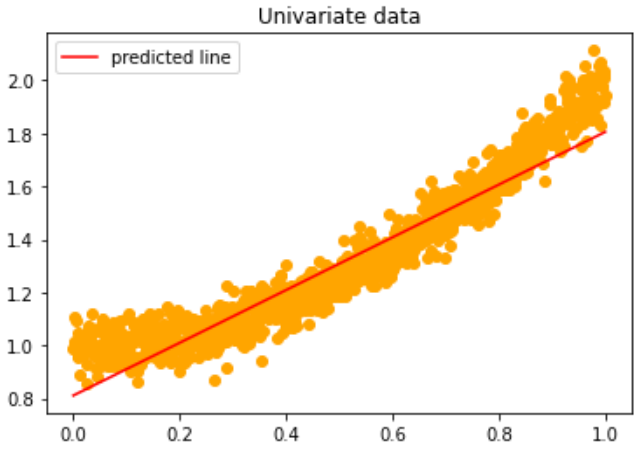


Univariate dataset with x-axis as feature and y-axis as target output.



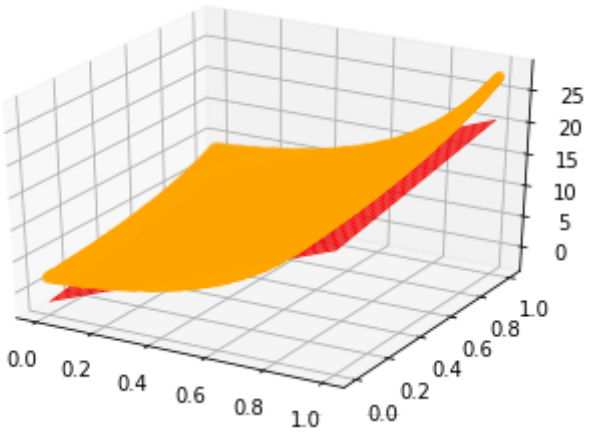
Bivariate data set, with x and y as features and z-axis as target output.

Predicted line for univariate data:



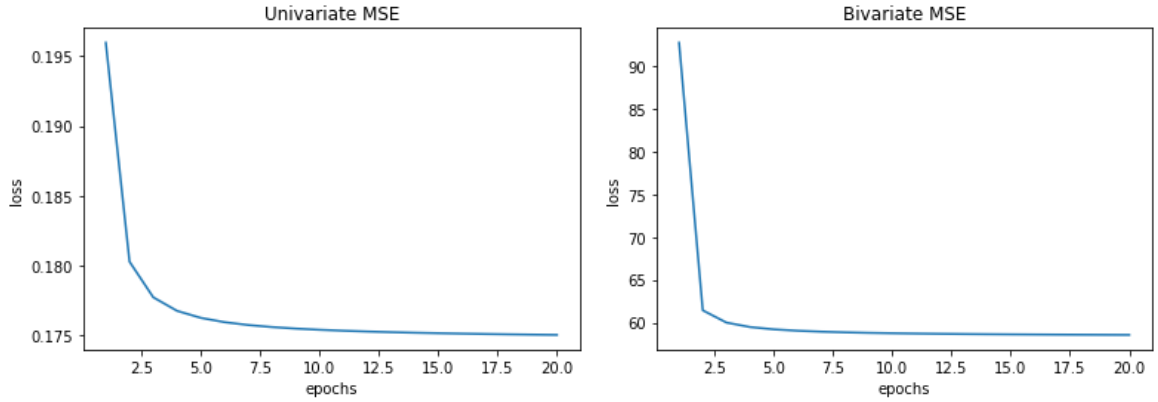
The predicted weights try to minimize MSE from as many data points as possible and thus try to fit a line midway between all the data points.

Predicted surface (in red) for bivariate data:



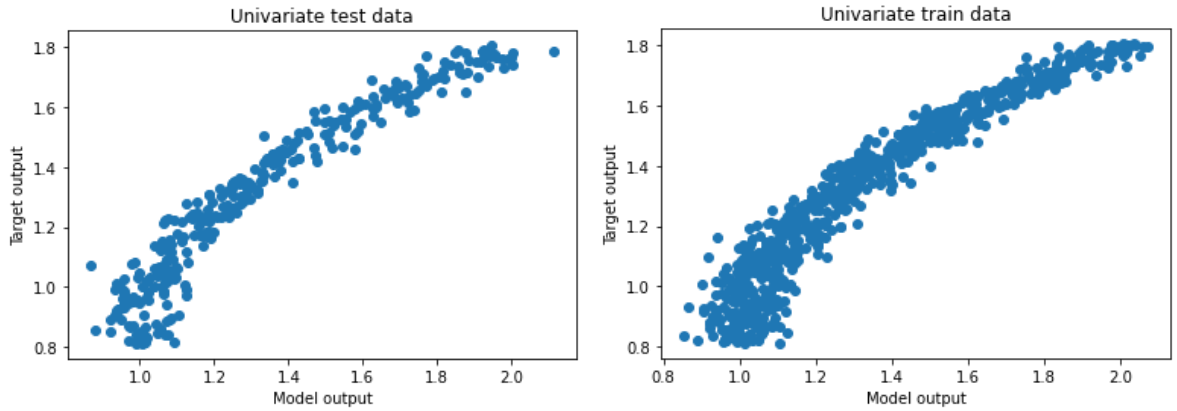
The hyperplane minimizes the distance of all other points from the plane.

Loss (MSE) plot:

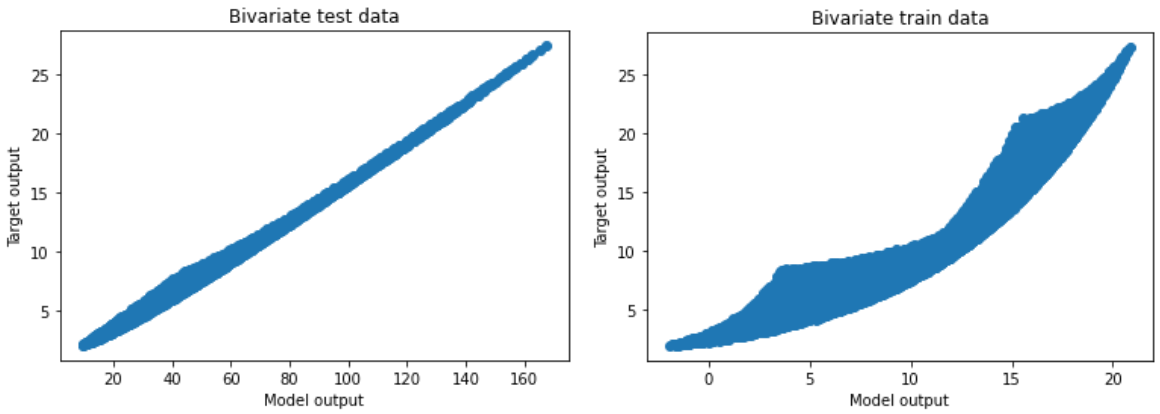


The error falls exponentially in the beginning and saturates at different loss values (different for univariate and bivariate) as the number of epochs increases. It can be observed that the saturated loss is more in case of bivariate data than in the case of univariate data.

Plot of model vs target output for univariate and bivariate data:



The predicted value is almost equal to the target value as the target output vs model output is “almost” following a linear fit for both the test and the train data, in case of univariate sample.



In bivariate case, for test data, the target and model output follows almost a straight line ie. the model performs well in case of test data but for train data the model performs more error as compared to test data. This suggests that the data may not be following a linear curve that our model is trying to fit (the hyperplane).

Conclusion:

After using a single neuron model on both univariate data and bivariate data, we can see that in both cases, the model tries to fit a straight line or a linear hyperplane to the data. But it cannot predict a curved surface and hence cannot perform non-linear regression.