CT5165 Principles of Machine Learning: Assignment 2

Jiarong Li 20230033, 1MAI1
Zhe Jing Chin 22221970, 1CSD1

The School of Computer Science
University of Galway
j.li11@nuigalway.ie, z.chin1@nuigalway.ie

1 Introduction

The objective of this assignment is to read, understand, implement and evaluate the machine learning algorithm called Perceptron and Multi-Layer Perceptron.

2 Data Preprocessing

The dataset we are using in this project is "wildfires.txt". The dataset consists of 204 rows with 10 columns, 9 independent variables with 1 target, the "fire".

Firstly, we remove the extra spaces in the "fire" column to standardise the texts. Next, we encode the "fire" column with no: 0, yes: 1 as the model only takes in numerical values. We apply min-max normalisation to the independent variables to standardised the range of the features.

The data is then split to a proportion of 1/3 for testing set and 2/3 for training set. The random splitting is stratified with balance proportion of the 2 classes in both training and testing set. The testing set is not used until the final model is trained.

We pre-set the cross validation to be 5 folds, hence the training dataset from the previous section is randomly split to 5 stratified versions. Thus in every fold of cross validation, 1/5 of the training set is treated as validation set while the rest is the train set.

To construct a dataset that is suitable for machine learning model, we remove the target column to form a dataset X to feed into our model while the target column form an array of Y which is the true labels.

3 Design and Implementation

3.1 The overview of our architecture

As shown in Figure1, our architecture consists of the data loader, n-fold cross validation, 2 hidden layers perceptron with four nodes per layer to form a binary classification model.

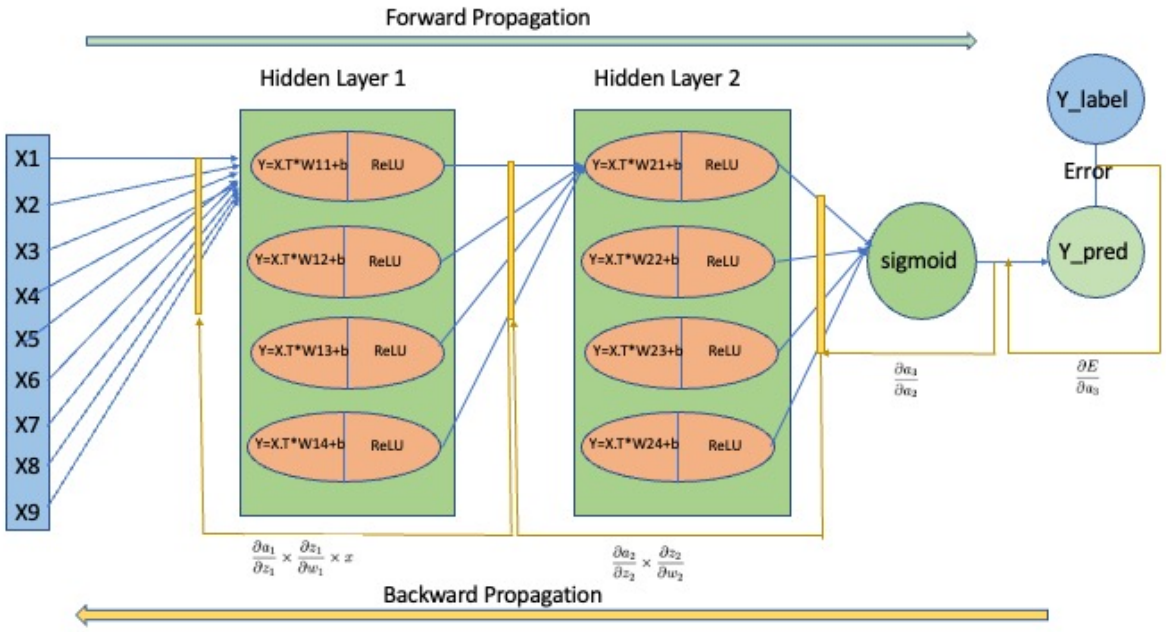

Figure 1: The overview of our MLP architecture. The pipeline consists of data loader, n-fold cross validation, 2 hidden layers perceptron with four nodes per layer to form a binary classification model. The model uses ReLU activation functions to avoid vanishing gradient.

3.2 Activation function

The model uses ReLU activation functions to avoid vanishing gradient. Besides, it applies sigmoid function in the output layer as the model is dealing with a binary classification hypothesis.

3.3 Feedforward function

The output of feedforward process can be computed by:

$$z = W \times X.T + b$$

where z denotes the output, W represents the weights matrix, X is the input vector and b is the bias.

3.4 MSE loss

We apply mean squared error (MSE) to measure the quality of our model as it is one of the most widely used loss function and it has good performance in analysing classificatio[1].

The MSE is given by:

$$MSE = \frac{1}{2 \times n} \times \sum_{i=1}^{n} (Y_{label} - Y_{predict})^2$$

where n represents the number of classes, Y_{label} denotes the target label of the input features, and $Y_{predict}$ represents the output predictions of the input features.

3.5 Backward Propagation

A typical computation of gradient descent of backward propagation is given by:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial \sigma_j}{\partial w_{ij}} \times \frac{\partial E}{\partial \sigma_j} = \begin{cases} \frac{\partial \sigma_j}{\partial z_j} \times \frac{\partial E}{\partial \sigma_j} & \text{if node } j \text{ is in Output Layer} \\ \frac{\partial z_j}{\partial w_{ij}} \times \frac{\partial \sigma_j}{\partial z_j} \times \frac{\partial E}{\partial \sigma_j} & \text{if node } j \text{ is in Hidden Layers} \end{cases} \quad (1)$$

where E denotes the error distance between target labels and output predictions from the MLP classifier, w_{ij} denotes the weight between ith node in layer $l-1$ and the jth node of layer l, σ_j represents the output of the activation function of jth node in layer l, and z_j represents the output of jth linear node in layer l.

4 Results and Evaluation

4.1 Comparison of loss and accuracy results of training.

The Figure2 shows the loss and accuracy results of our custom MLP model versus the SKlearn MLP classifier of each epoch during the training process. Compared with SKlearn MLP, our MLP model has a better performance in classification. The average loss during training process per epoch of our MLP is less than the average loss during training per epoch of SKlearn MLP. In summary, our average accuracy during training and testing is better than the average accuracy during training and testing of SKlearn MLP.

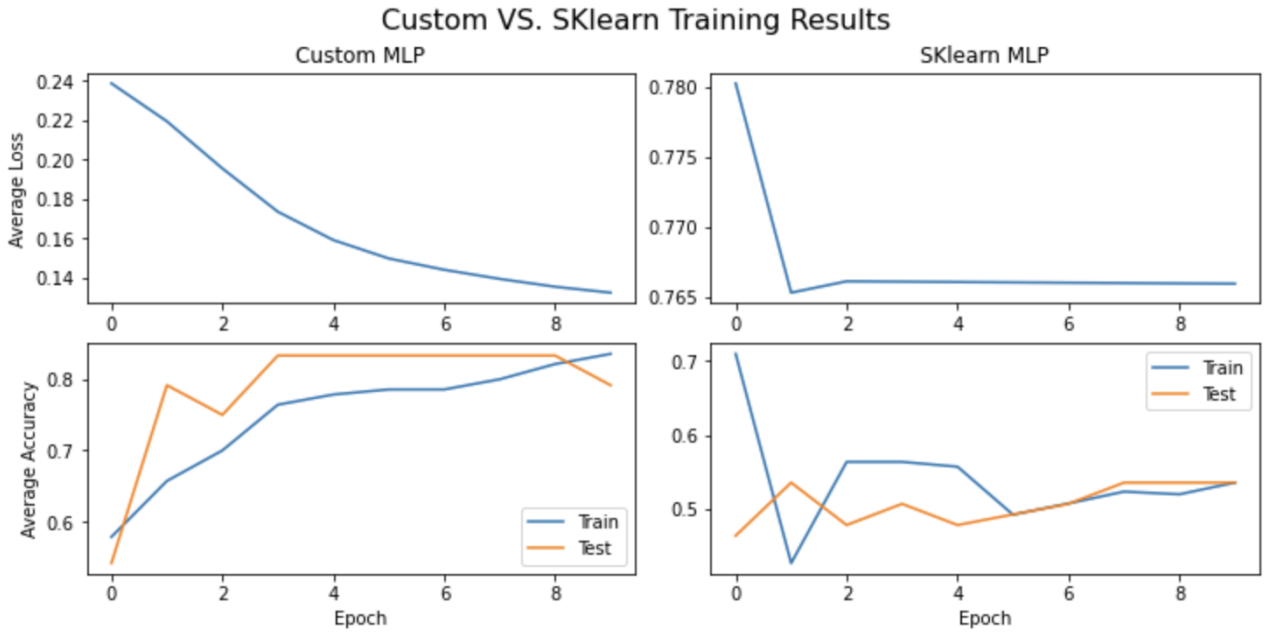

Figure 2: The loss and accuracy results of our custom MLP model versus the SKlearn MLP classifier of each epoch during the training process. The right-hand-side column shows the loss and accuracy results of our custom MLP of each epoch during the training process, and the left-hand-side column shows the loss and accuracy results of the SKlearn MLP classifier of each epoch during the training process.

4.2 Comparison of loss and accuracy results of testing.

The Figure3 shows confusion matrix of our custom MLP and the confusion matrix of SKlearn MLP. Compared with SKlearn MLP, our MLP model has a better performance in classification measuring with confusion matrix.

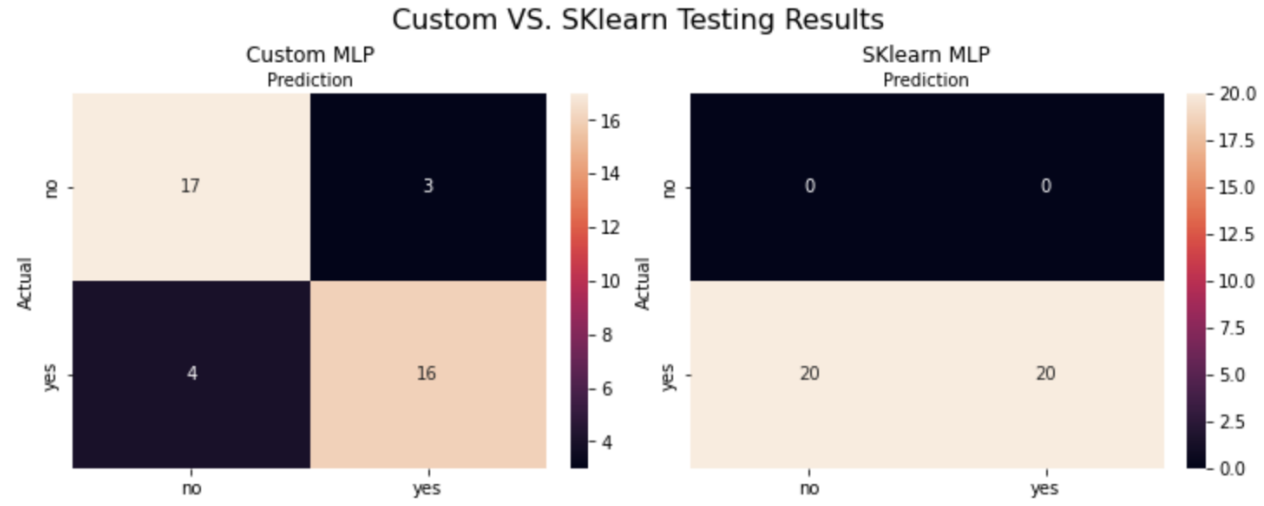

Figure 3: The confusion matrix of our custom MLP and the confusion matrix of SKlearn MLP. It shows that the values of precision and recall in custom MLP are higher than the values of precision and recall in SKlearn MLP.

4.3 Comparison with AUC-ROC Curve.

The Figure4 shows the AUC-ROC curve of our custom MLP and the AUC-ROC curve of SKlearn MLP. Compared with SKlearn MLP, our MLP model has a better performance in classification.

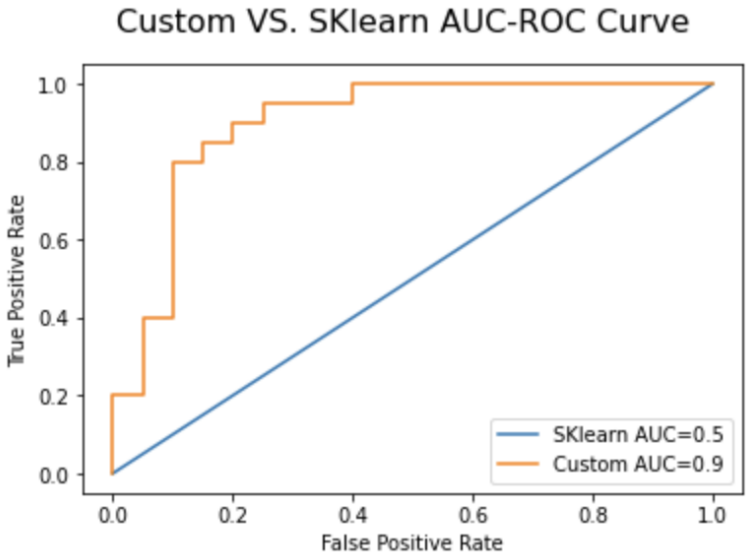

Figure 4: The AUC-ROC curve of our custom MLP and the AUC-ROC curve of SKlearn MLP. The orange line represents the AUC of our custom MLP. It means there is a 90.1% chance that the model will be able to distinguish between 2 classes. The blue line represents the AUC of our SKlearn MLP. It means there is a 50% chance that the model will be able to distinguish between 2 classes.

5 Conclusions

In conclusion, our custom Multi-layers Perceptron model is performing better than the built-in SKlearn MLP classifier on the "wildfires" dataset when both are implemented with the same hyperparameters including learning rate, epoch, threshold, and hidden layers. However, this does not conclude that the built-in model is not a good model. Based on the "No Free Lunch" theorem[2], there is no a single good hypothesis or model setting that fits for all problems. Each model should have their own configurations to suit even the same issue. Hence, it is possible to obtain different results with the same hyperparameters on the same issue. Other than that, the built-in model has included more functions such as momentum for gradient descent update, and they might be using different weight initialization method.

6 Contributions

Table 1: Tasks distribution.

PIC	Tasks
Jiarong Li	Single Perceptron; Feedforward function; Backpropagation; Report compiling;
Zhe Jing Chin	Data processing; Model construction; Model initialising; Gradient descent; Evaluation; Code compiling;

The above table only shows the simplified task distribution of the project. However, both members contributed in the whole process equally and have thorough understanding in each step.

References

[1] Erich L Lehmann and George Casella. *Theory of point estimation*. Springer Science & Business Media, 2006.

[2] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

A Appendix

See **'Assignment2 Model/Assignment2_MLP.ipynb'** for the detailed coding. Select *"Run All Cells"* in jupyter notebook to run the code.

```
class CustomMLP:

    def feedforward(self, x):
        self.FF_Output = []

        self.FF_Output.append(self.relu(np.dot(self.Weights[0], x) +
        ↪  self.Biases[0]))

        for i in range(1, len(self.hiddenLayer)):
            self.FF_Output.append(self.relu(np.dot(self.Weights[i],
            ↪  self.FF_Output[i-1]) + self.Biases[i]))

        self.FF_Output.append(self.sigmoid(np.dot(self.Weights[len(self.
        ↪  FF_Output)], self.FF_Output[-1]) +
        ↪  self.Biases[len(self.FF_Output)]))

    def backpropagation_error(self, y):
        #start from last to first layer
        self.BP_Error = []

        z = self.FF_Output
        self.BP_Error.append((z[-1] - y) * self.deriv_sigmoid(z[-1]))

        self.BP_Error.append(self.BP_Error[0][0]*self.Weights[-1] *
        ↪  self.deriv_relu(z[-2]))

        for i in range(1, len(z)-1):
```

```
            self.BP_Error.append(np.dot(self.BP_Error[i],
            ↪  self.Weights[len(z)-1-i]) *
            ↪  self.deriv_relu(z[len(z)-2-i]))

        #flip to [first layer, last layer]
        self.BP_Error = np.flip(self.BP_Error, 0)

    def gradient_descent(self, x):
        self.Weights[0] -= self.lr * np.kron(self.BP_Error[0],
        ↪  x).reshape(self.hiddenLayer[0], self.X.shape[1])
        self.Biases[0] -= self.lr * self.BP_Error[0]

        for i in range(1, len(self.Weights)-1):
            self.Weights[i] -= self.lr * np.kron(self.BP_Error[i],
            ↪  self.FF_Output[i-1]).reshape(self.hiddenLayer[i],
            ↪  self.hiddenLayer[i-1])
            self.Biases[i] -= self.lr * self.BP_Error[i]

        self.Weights[-1] -= self.lr * np.kron(self.BP_Error[-1],
        ↪  self.FF_Output[-2]).reshape(self.hiddenLayer[-1])
        self.Biases[-1] -= self.lr * self.BP_Error[-1]

```