Machine Learning Assessment Report

August 21, 2024

JC3509: Machine Learning

University of Aberdeen

1 Task1 - Data Preparation

1.1 Import the dataset

This dataset includes the chemical characteristics of food products manufactured by three different manufacturers comprising 178 entries.
The target(response) is the initial column "Producer" specifying the manufacturer of each analyzed sample. There are 13 features (predictors) including Amino_acid, Malic_acid, Ash, Alc, Mg, Phenols, Flavanoids, Nonflavanoid phenols, Proanth, Colo_int, Hue, OD, Proline.

```
# load data set
data_path = os.path.join(os.getcwd(),'Assessment1_Dataset.csv')
dataset_raw = pd.read_csv(data_path,header=None,sep=',')

# Use the first row as the column name and reload the dataset
dataset_raw.columns = dataset_raw.iloc[0]
dataset_raw = dataset_raw[1:]

dataset_raw
```

Code Snippet 1: Import the dataset from the csv file.

1.2 Preprocess the data

The data preprocessing in this assessment has been divided into following steps:

1. **Missing Data Handling**: Check for any NaN values, and if present drop the row. This can avoid the situation where there are some rows that contain missing values.

2. **Normalization**: The dataset should be normalized. This is because in the dataset, the data in the column "Proline" are around 1,000, the data in the column "Mg" are around 90-150, the data in column "Amino acid" and "Acl" are around 10-20, others are around 1-10. Thus, it is necessary to normalize the data due to its non-scale consistency, which can ensure the contribution of each feature to the model learning process. To normalize the data x, it is divided by the norm value of each dimension of the data as:

$$x = x/\text{norm}(x) \tag{1}$$

3. **Dataset Splitting**: To ensure the balance of the training and testing dataset, I first calculate the number of data from different classes and then randomly select 75% of the dataset from each class as the training dataset and the remaining (25%) is used as the testing dataset.

4. **K-fold Cross-Validation**: K-fold cross-validation would be used in this assessment due to the dataset's small scale. It can have better performance estimate and effective use of data, and also provides a more robust estimate of a model's performance compared to a single train-test split. Here, I set $k = 5$ of the k-fold cross-validation.

At the end of data processing stage, the clean and normalized training and testing datasets are achieved. In addition, k-fold cross-validation is also implemented to evaluate the model design.

2 Task2 - Model Construction

2.1 Loss function

This is a multi-class classification problem, so the softmax cross entropy loss would be used to compute the loss of the problem.

$$\mathcal{L}(w) = -\frac{1}{N}\sum_{n=1}^{N}\log p(y = y_n|x_n). \tag{2}$$

2.2 Network Design

There are two hidden layers and one output layer in this fully connected artificial neural network to obtain strong approximation ability. In the first layer, there are a number of (would be selected in a list) neurons that would be activated by the ReLu function to introduce nonlinearity and avoid gradient vanishment. The second layer is same as the first layer. For the output layer, there are three neurons that would be activated by the softmax function to restrict the output range. The reason I select this architecture is that this task is relatively simple and the neural network with two hidden layers is sufficient to solve this problem.

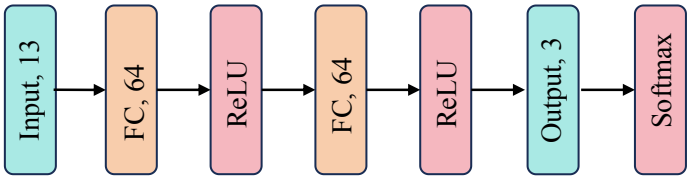


Figure 1: The architecture of the designed neural network.

Forward Pass: x is the input data, w is the weight of each layer, b is the bias of each layer, a is the output of activation function of each layer.
In the first layer, the forward propagation is expressed as follows:

$$z_1 = w_1 x + b_1 \tag{3}$$
$$a_1 = \text{ReLU}(z_1) \tag{4}$$

In the second layer, the forward propagation is expressed as follows:

$$z_2 = w_2 a_1 + b_2 \tag{5}$$
$$a_2 = \text{ReLU}(z_2) \tag{6}$$

In the output layer, the forward propagation is expressed as follows:

$$z_3 = w_3 a_2 + b_3 \tag{7}$$
$$a_3 = \text{Softmax}(z_3) \tag{8}$$

Backward Pass: y_i is the ground truth label. Backward propagation for each parameters is derived as follows: For the output layer:

$$\frac{\partial \mathcal{L}}{\partial a_3} = -\frac{1}{a_3^k} \tag{9}$$

$$\frac{\partial \mathcal{L}}{\partial z_3} = \frac{\partial \mathcal{L}}{\partial a_3}\frac{\partial a_3}{\partial z_3} = a_3^k - y_i \tag{10}$$

$$\frac{\partial \mathcal{L}}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial z_3}\frac{\partial z_3}{\partial w_3} = (a_3^k - y_i)a_2^k \tag{11}$$

$$\frac{\partial \mathcal{L}}{\partial b_3} = \frac{\partial \mathcal{L}}{\partial z_3}\frac{\partial z_3}{\partial b_3} = (a_3^k - y_i) * 1 \tag{12}$$

$$\frac{\partial \mathcal{L}}{\partial a_2} = \frac{\partial \mathcal{L}}{\partial z_3}\frac{\partial z_3}{\partial a_2} = (a_3^k - y_i)w_3 \tag{13}$$

For the second hidden layer:

$$\frac{\partial \mathcal{L}}{\partial z_2} = \frac{\partial \mathcal{L}}{\partial a_2} * 1 = (a_3^k - y_i)w_3 \tag{14}$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial z_2}\frac{\partial z_2}{\partial w_2} = (a_3^k - y_i)w_3 a_1^k \tag{15}$$

$$\frac{\partial \mathcal{L}}{\partial b_2} = \frac{\partial \mathcal{L}}{\partial z_2}\frac{\partial z_2}{\partial b_2} = (a_3^k - y_i)w_3 \tag{16}$$

$$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial z_2}\frac{\partial z_2}{\partial a_1} = (a_3^k - y_i)w_3 w_2 * 1 \tag{17}$$

For the first hidden layer:

$$\frac{\partial \mathcal{L}}{\partial z_1} = \frac{\partial \mathcal{L}}{\partial a_1} * 1 = (a_3^k - y_i)w_3 w_2 \tag{18}$$

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial z_1}\frac{\partial z_1}{\partial w_1} = (a_3^k - y_i)w_3 w_2 x \tag{19}$$

$$\frac{\partial \mathcal{L}}{\partial b_1} = \frac{\partial \mathcal{L}}{\partial z_1}\frac{\partial z_1}{\partial b_1} = (a_3^k - y_i)w_3 w_2 * 1 \tag{20}$$

2.3 Gradient Descent

The mini-batch stochastic gradient descent is used here, which can improve the computational effiency.
First, I create mini-batches from the training data.

```
# create mini-batches
def create_mini_batches(trainset, batch_size):
    y_data = trainset[:, 0].astype(np.int32)
    X_data = trainset[:, 1:]

    mini_batches = []

    # index
    index = np.arange(X_data.shape[0])
```

```
    np.random.shuffle(index)
    mini_batch_index = [index[i:i+batch_size] for i in range(0, index.shape[0],
    batch_size)]

    mini_batch_x = []
    mini_batch_y = []
    for batch_index_ in mini_batch_index:

        mini_batch_x.append(X_data[batch_index_])

        #create one hot label to record the target
        one_hot_label = np.zeros((batch_index_.shape[0], 3))

        one_hot_label[np.arange(one_hot_label.shape[0]), y_data[batch_index_]-1] = 1

        mini_batch_y.append(one_hot_label)

    return mini_batch_x, mini_batch_y
```

Code Snippet 2: The function to create mini-batch.

Then stochastic gradient descent is implemented to update the parameters of the neural network. Here, α and lr refer to the learning rate.

1. First, the gradient is computed according to the derivation in Section 2.2: $\frac{\partial f(w)}{\partial w}$
2. Then, repeat the following step until the pre-defined number of training iterations is finished:

$$w^{(k+1)} = w^{(k)} - \alpha \frac{\partial f(w^{(k)})}{\partial w} \tag{21}$$

```
def SGD(W_1, b_1, W_2, b_2, W_3, b_3, dW_1, db_1, dW_2, db_2, dW_3, db_3, lr):

    W_1 = W_1 - lr * dW_1
    b_1 = b_1 - lr * db_1
    W_2 = W_2 - lr * dW_2
    b_2 = b_2 - lr * db_2
    W_3 = W_3 - lr * dW_3
    b_3 = b_3 - lr * db_3

    return W_1, b_1, W_2, b_2, W_3, b_3
```

Code Snippet 3: The implementation of stochastic gradient descent.

Or using advanced optimisation Adam. Here, I use the Adam because it can influence the gradient update to move in the direction of the previous update to stabilise training and reduce oscillations, and add a decay factor to stop growing and never shrinking.
In Adam optimizer, m_t is the momentum, v_t is the moving average of the squared gradient, β_1 and β_2 are exponential decay rates for these moment estimates, w is the parameter of the neural

network, η is the learning rate, ϵ is a small value to avoid division by zero.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta \mathcal{L}(w) \tag{22}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_\theta \mathcal{L}(w))^2 \tag{23}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{24}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{25}$$

$$w = w - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}} \tag{26}$$

3 Task3 - Model Training

3.1 Model Training

In the model training stage, I have undertaken the following tasks:

1. In training process, the parameters I set are a list of learning rate and a list of number of neurons. **lr_list = [0.001, 0.01, 0.1], neuron_list = [32, 64, 128]**.

2. In order to simplify the code, I predefine some functions to initialize the parameters of the neural network. Accordingly, forward propagation and Adam optimisation are modified. And in training part, k-fold cross validation is used to select the optimum combination of parameters.

More details about the model training is described in Algorithm 1, the training result is shown in Tab 1.

learning rate	number of neurons	accuracy
0.001	32	0.904
0.001	64	0.948
0.001	128	0.956
0.01	32	0.993
0.01	64	0.978
0.01	128	0.941
0.1	32	0.970
0.1	64	0.985
0.1	128	0.985

Table 1: The accuracy of model training.

3.2 Module Regularisation

This artificial neural network only has two hidden layers and has compared the number of neurons in the previous sections, so it is unnecessary to reduce the layers or parameters. Instead, shrinkage methods can be used to do regularization, including Ridge Regression and Lasso Regression.

Algorithm 1 Train Neural Network with Varying Learning Rates and Neuron Counts

```
Initialize lr_list with different learning rates
Initialize neuron_list with different neuron counts
Initialize losses as an empty dictionary
for each learning rate lr in lr_list do
    for each neuron count numNeuron in neuron_list do
        Initialize an entry in losses for the current lr and numNeuron
        Initialize loss_train and loss_validate as empty lists
        Initialize acc = 0
        for each fold i in 5-fold cross-validation do
            Generate training and validation sets for the current fold
            Initialize neural network parameters for the current numNeuron
            Initialize Adam optimizer with learning rate lr
            for epoch = 1 to 100 do
                Create mini-batches from the training set
                Initialize train_loss_tmp as an empty list
                for each mini-batch (batch_x, batch_y) do
                    Perform forward propagation on batch_x
                    Perform back propagation to compute gradients
                    Update parameters using Adam optimizer
                    Compute training loss and append to train_loss_tmp
                end for
                Compute average training loss for the epoch and append to loss_train_
                Perform forward propagation on validation data
                Compute validation loss and append to loss_validate_
            end for
            Append loss_train_ and loss_validate_ to loss_train and loss_validate, respectively
            Evaluate model accuracy on the validation set and add to acc
        end for
        Compute average training and validation losses, store in losses dictionary
        Compute average accuracy across all folds
        Print learning rate, neuron count, and average accuracy
    end for
end for
```

Through regularization, we can penalize complex models and favour simpler ones:

$$\min_{w} L(w) + \Omega(w) \tag{27}$$

1. The loss function of Ridge Regression:

$$\mathcal{L}(w) = -\frac{1}{N}\sum_{n=1}^{N} \log p(y = y_n|x_n) + \lambda \sum_{j=1}^{P} w_j^2. \tag{28}$$

Tab 2 shows the training result by using Ridge Regression.

learning rate	number of neurons	accuracy
0.001	32	0.993
0.001	64	0.978
0.001	128	0.963
0.01	32	0.993
0.01	64	0.993
0.01	128	0.978
0.1	32	0.949
0.1	64	0.911
0.1	128	0.933

Table 2: The accuracy of Ridge Regression.

2. The loss function of Lasso Regression:

$$\mathcal{L}(w) = -\frac{1}{N}\sum_{n=1}^{N} \log p(y = y_n|x_n) + \lambda \sum_{j=1}^{P} |w_j|. \tag{29}$$

Tab 3 shows the training result by using Lasso Regression.

learning rate	number of neurons	accuracy
0.001	32	0.970
0.001	64	0.70
0.001	128	0.993
0.01	32	0.985
0.01	64	0.978
0.01	128	0.978
0.1	32	0.963
0.1	64	0.970
0.1	128	0.985

Table 3: The accuracy of Lasso Regression.

3.3 Model inference

In this part, the trainset and testset splitted in section 1.2 (Preprocess the data) would be used to train and test. According to the training results, there are several sets of parameters with the highest accuracy which is 0.993. I randomly pick one of them, that is when the learning rate is 0.001, the number of neurons is 32 and the loss is calculated by cross_entropy_loss with Ridge regression. So the parameters are the learning rate 0.001 and the number of neurons in two hidden layers, 32 and 32 respectively, and three neurons in the output layer. Tab 4 shows the result of model inference.

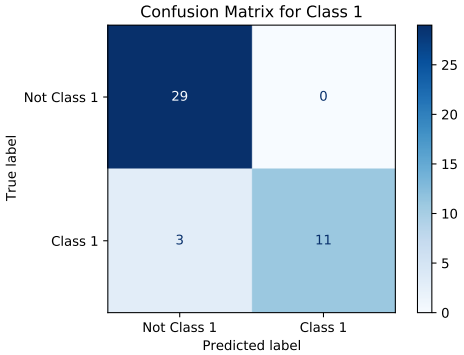
learning rate	number of neurons	accuracy
0.001	32	0.907

Table 4: The accuracy of model inference.

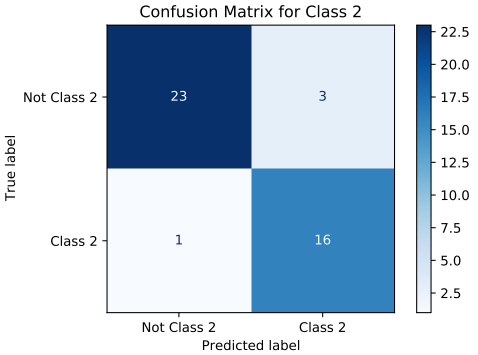
4 Task 4 – Evaluation

4.1 Present Results

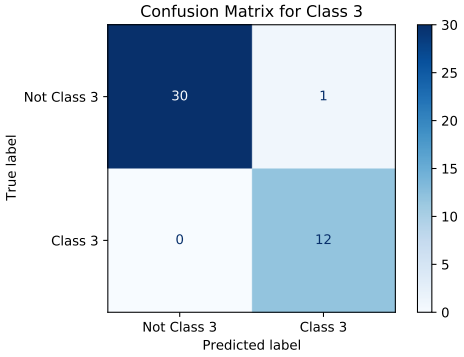
Here, I use the confusion matrix to show the summary of the prediction results for this classification problem, as it can evaluate the performance of the classification models. The problem of this assessment has more than two classes, so the confusion matrix expands to accommodate all possible classes, allowing for the evaluation of class-specific performance. In the section 1.2 (Preprocess the data), I have made the class balance, so the accuracy can be used to measure. The ROC curve for each class has been drawn, which can show the performance of a classification model at all classification thresholds. Class 1 refers to Producer 1, Class 2 refers to Producer 2, Class 3 refers to Producer 3.
Fig. 2 shows the confusion matrixes for three classes.



(a) Class 1



(b) Class 2



(c) Class 3

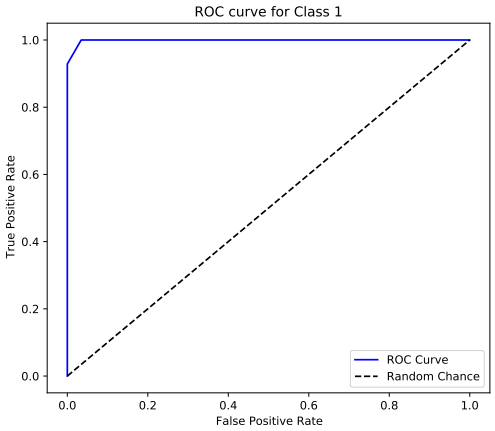
Figure 2: Confusion Matrix.

Tab. 5 illustrates the precision, sensitivity, specificity, NPV and accuracy for three classes.

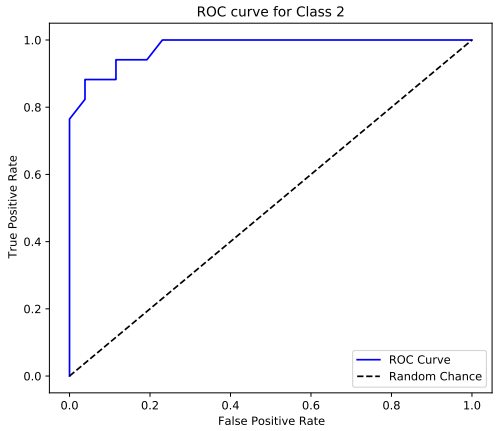
	Precision	Sensitivity	Specificity	NPV	Accuracy
Class 1	1.00	0.79	1.00	0.91	0.93
Class 2	0.84	0.94	0.88	0.96	0.91
Class 3	0.92	1.00	0.97	1.00	0.98

Table 5: Evaluation metrics for three classes.

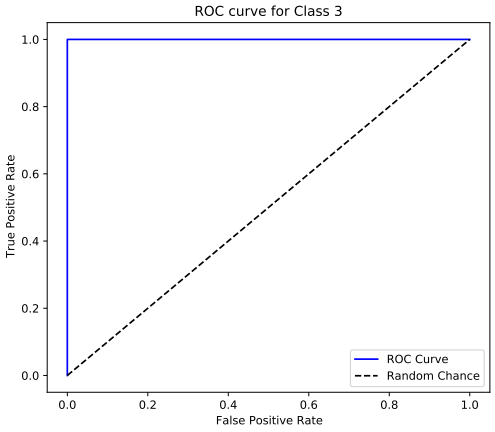
Fig. 3 shows the ROC curve for each class.



(a) Class 1



(b) Class 2



(c) Class 3

Figure 3: ROC curve for three classes.

4.2 Plot

(a) **What does your loss curve tell you?**

Fig. 4 shows the loss curve with different learning rates and the number of neurons. According to the result of model training, the model with 0.01 learning rate and 32 neurons has the highest accuracy, which is 0.993. Both training and validation losses converge to nearly zero by around 40 epochs, which indicates that the model is learning effectively from the data and generalizing well to the validation set within that period.

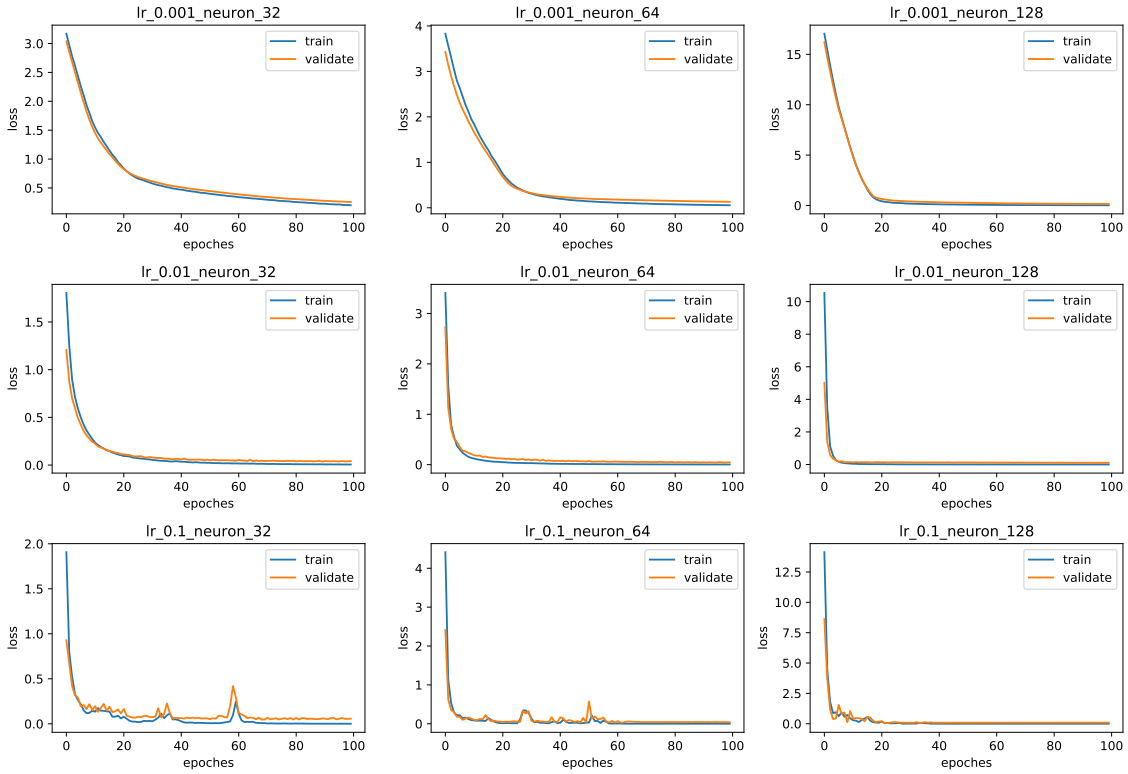


Figure 4: Loss curve.

(b) **Are you observing any overfitting or underfitting?**

For the graph of model with 0.01 learning rate and 32 neurons, since both training and validation loss are converging to a similar low value, there is no immediate indication of overfitting and underfitting.

(c) **Does the addition of regularisation help?**

Fig. 5 and Fig. 6 show the loss curve of these two regressions. Both the Ridge regression and the Lasso regression do not help with model training. Although some of them converge, they require more epochs compared with the model without regularisation and some of them even haven't converge.

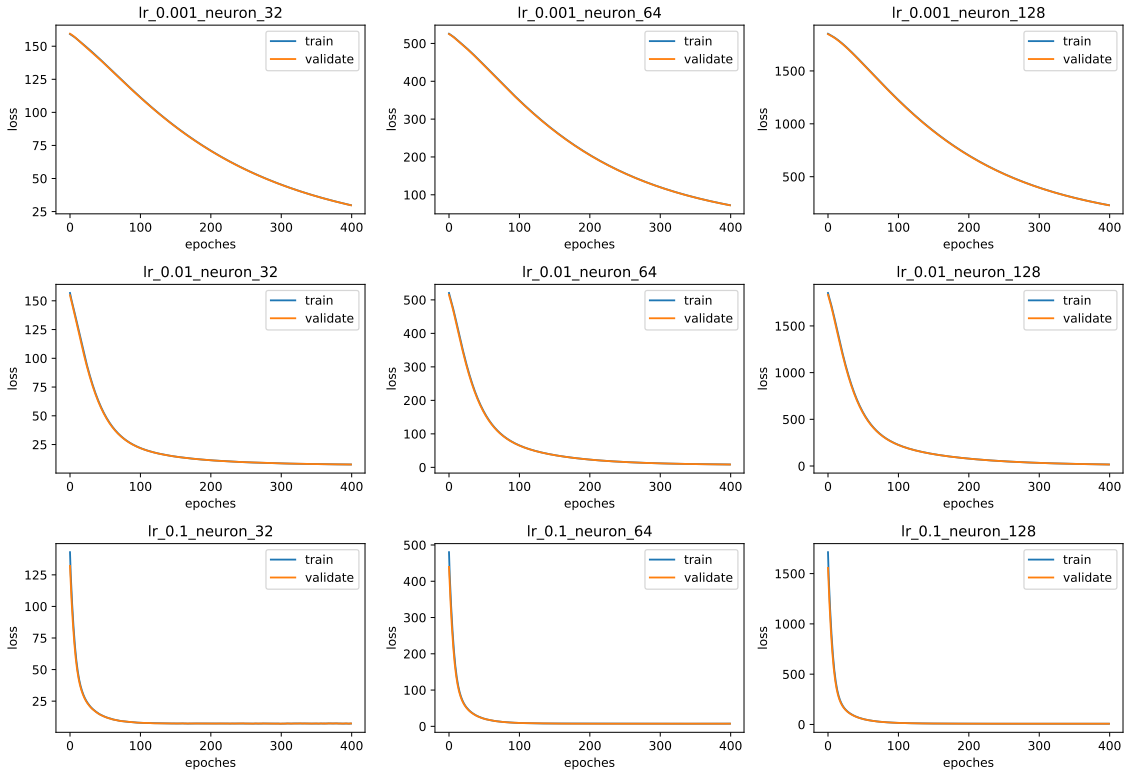


Figure 5: Loss curve for Ridge Regression.

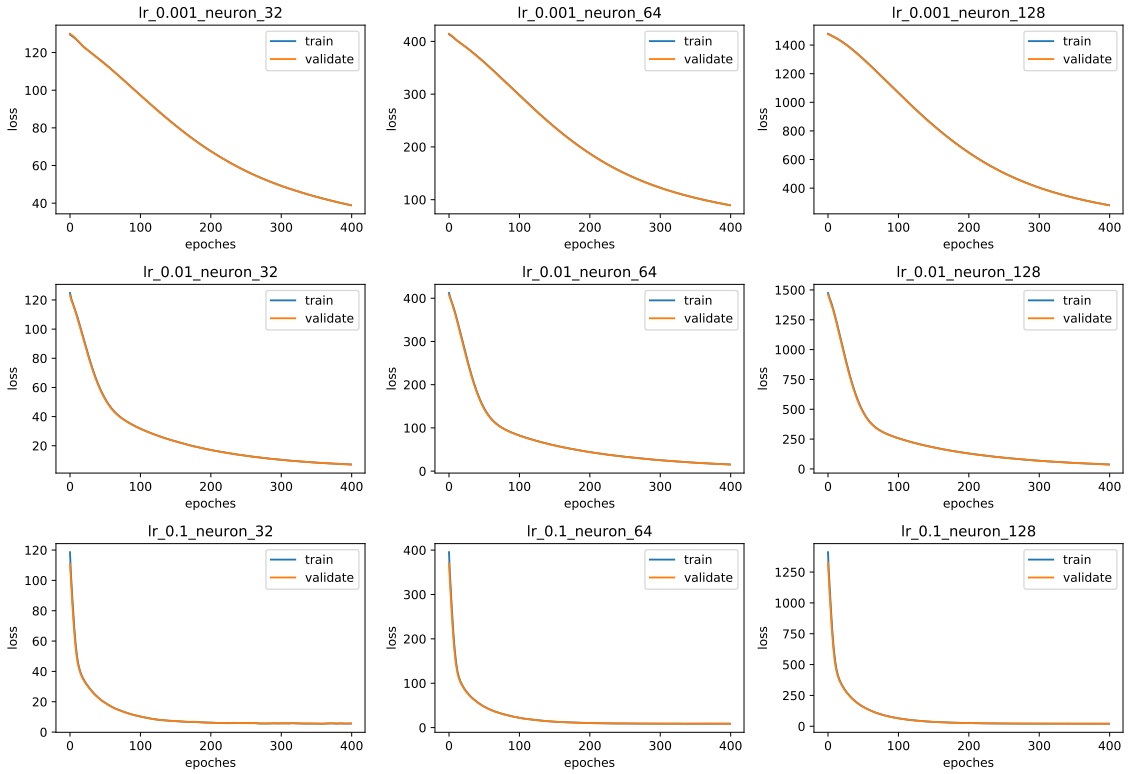


Figure 6: Loss curve for Lasso Regression.

4.3 Explain Results

Confusion matrix for Class 1

- True Negative(TN): 29 instances were correctly predicted as not belonging to Class 1.
- False Positive(FP): 0 instances were incorrectly predicted to belong to Class 1 (i.e., no instances from other classes were mistakenly identified as Class 1).
- False Negative(FN): 3 instances that are actually of Class 1 were incorrectly predicted to not belong to Class 1.
- True Positive(TP): 11 instances were correctly identified as belonging to Class 1.
- Precision: 1.00 (No instances were incorrectly labeled as Class 1)
- Sensitivity: 0.79 (Some instances of Class 1 were missed)
- Specificity: 1.00 (All instances that were not Class 1 were correctly identified)
- NPV: 0.91 (High probability that a non-Class 1 prediction is correct)

- Accuracy: 0.93 (Overall high correct classification rate)

This model classifies the products of Producer 1 well, though it has missed some products truly of Producer 1, shown by perfect precision and sensitivity. All products not from Producer 1 are correctly classified by the model, shown by perfect specificity.

ROC curve for Class 1

According to Fig 3a, the model performs well in classifying the products of Producer 1 as the top left corner is close to 1.

Confusion matrix for Class 2

- True Negative(TN): 23 instances were correctly predicted as not belonging to Class 2.
- False Positive(FP): 3 instances were incorrectly predicted to belong to Class 2 (i.e. instances of other classes were mistakenly identified as Class 2).
- False Negative(FN): 1 instance that is actually of Class 2 was incorrectly predicted as not belonging to Class 2.
- True Positive(TP): 16 instances were correctly identified as belonging to Class 2.
- Precision: 0.84 (Some instances from other classes were incorrectly labeled as Class 2)
- Sensitivity: 0.94 (A high proportion of Class 2 instances were correctly identified)
- Specificity: 0.88 (A few instances that were not Class 2 were incorrectly labeled as Class 2)
- NPV: 0.96 (Very high probability that a non-Class 2 prediction is correct)
- Accuracy: 0.91 (Overall high correct classification rate)

For classifying Producer 2, the model performs generally well, but is sometimes incorrectly shown by precision.

ROC curve for Class 2

According to Fig. 3b, when classifying the products of Producer 2, the model shows a relatively poor performance, because the top left corner has a large distance to 1.

Confusion matrix for class 3

- True Negative(TN): 30 instances were correctly predicted as not belonging to Class 3.
- False Positive(FP): 1 instance was incorrectly predicted as belonging to Class 3 (i.e., an instance of another class was mistakenly identified as Class 3).
- False Negative(FN): 0 instances of Class 3 were incorrectly predicted as not belonging to Class 3 (i.e., all instances of Class 3 were correctly identified).
- True Positive(TP): 12 instances were correctly identified as belonging to Class 3.
- Precision: 0.92 (Few instances from other classes were incorrectly labeled as Class 3)
- Sensitivity: 1.00 (All instances of Class 3 were correctly identified)
- Specificity: 0.97 (Almost all instances that were not Class 3 were correctly identified)

- NPV: 1.00 (All instances predicted as not Class 3 were correct)
- Accuracy: 0.98 (Very high overall correct classification rate)

When classifying Producer 3, the model performs outstandingly shown by perfect sensitivity and very high precision and specificity. It has the highest accuracy of all, suggesting it is the most reliably classified by the model.

ROC curve for Class 3

According to Fig 3c, the model performs perfectly in classifying the products of Producer 3, as the top left corner is at 1, that is, the true positive rate is 1.0 and the false positive rate is 0.0.

Overall, this model appears well-calibrated, particularly for Producer 3, with its high accuracy indicating strong performance in classifying instances across all three classes. When classifying Producer 1 and Producer 2, the model still has some room for improvement, particularly in sensitivity for Producer 1 and precision for Producer 2.