UiO : Department of Physics
University of Oslo

FYS-STK4155 Project 2

Classification and Regression

Astrid Tesaker, Vemund Thorkildsen & Sofie Tunes

Abstract

In this study we explored two classification algorithms, namely logistic regression and Artificial Neural Networks (ANNs). We saw how these algorithms are closely related to ANNs used for regression. In the first part, the classification algorithms were employed on the the original skewed credit card dataset and a downsampled dataset (non biased output). From the results obtained in this study, it seems that the classification algorithms perform similarly on both datasets, but there is a larger discrepancy between the datsets. This manifested itself as a higher accuracy score on the skewed dataset, but lower F1 score. A comparison between gradient decent and stochastic gradient descent was also made, obtaining similar results. The neural network was then modified to perform regression on the widely used Franke function. The results were compared to Ordinary Least Squares (OLS) regression and we found that OLS performed slightly better than our Neural Network.

Contents

Introduction

The ultimate goal of Machine Learning (ML) is to find and use relationships in data to make educated predictions. Such predictions can largely be divided into two groups: classification and regression. Regression can be seen as the preferred method of predicting continuous functions. As an example, Tesaker et al. (2019) successfully employed regression methods for modelling elevation as a function of spacial coordinates. Classification algorithms are used for predicting discrete outcomes, and it can therefore seem that these methods share few characteristics. However, both rely on relationships extracted from input data, and on minimizing a given cost function. There are multiple ways of extracting these relationships. This article will have two main focus areas: logistic regression employed for solving classification problems, and how neural networks can be used for both classification and regression problems.

The report will be structured as follows. First, an introduction to relevant theory will be presented. This is followed by a brief introduction to the credit card dataset, and the Franke function, which has been employed for classification and regression analysis respectively. The results for classification and regression will then be presented. A brief discussion will then conclude this report. This report will explore why the flexibility of Neural Networks has made it so popular in data analysis.

Theory

As mentioned in the Introduction, this report will focus on methods for classification and regression. First, linear and logistic regression will be introduced, before diving into how the optimal parameters are found (i.e gradient descent methods). In the final part of this section, neural networks will be introduced.

3.1 Linear regression

Linear regression is a useful tool for predicting continuous functions. In this report, the cost functions of Ordinary Least Squares (OLS), Ridge and LASSO regression will simply be stated. For further information regarding these cost functions, the reader is referred to Tesaker et al. (2019). For OLS, the cost function is given as:

$$C(\mathbf{X}, \hat{\beta}) = \frac{1}{n}\{(\mathbf{y} - \mathbf{X}^T\hat{\beta})^T(\mathbf{y} - \mathbf{X}^T\hat{\beta})\}, \tag{3.1}$$

where $\mathbf{X}$ is the design matrix, $\mathbf{y}$ is the observed data, and $\hat{\beta}$ is the unknown parameters for fitting a model to the observed data. For ridge regression, the cost function is defined as:

$$C(\mathbf{X}, \hat{\beta}; \lambda) = \frac{1}{n}\{(\mathbf{y} - \mathbf{X}\hat{\beta})^T(\mathbf{y} - \mathbf{X}\hat{\beta})\} + \lambda\hat{\beta}^T\hat{\beta}, \tag{3.2}$$

where the last term is known as L2 regularization and λ is the penalty parameter. In equation 3.3, the cost function for Lasso regression is shown,

$$C(\mathbf{X}, \hat{\beta}; \lambda) = \frac{1}{n}\{(\mathbf{y} - \mathbf{X}\hat{\beta})^T(\mathbf{y} - \mathbf{X}\hat{\beta})\} + \lambda\sqrt{\hat{\beta}^T\hat{\beta}}. \tag{3.3}$$

LASSO regression employs L1 regularization. This implies that the $\hat{\beta}$ parameters can be shrinked to zero. The addition of this term, results in a cost function with no analytical solution for its derivative (Hjorth-Jensen, 2019a). Thus, an iterative approximation must be used for solving the system. This is in contrast to OLS and Ridge, which can be solved directly through matrix inversion. However, it more closely resembles logistic regression, which must also be solved iteratively (section 3.2.1).

3.2 Logistic regression

Whereas linear regression can be used for fitting continous functions, logistic regression can be employed for predicting discrete outcomes or categories. In this report, the focus will be on binary logistic regression (i.e two outcomes), as the response in the selected dataset is binary (section 4). However, it should be noted that it is possible to perform similar analysis with multiple outcomes, often denoted multiclass logistic regression (Hjorth-Jensen, 2019a).

It is possible to use linear regression for classification purposes. However, a linear regression cost function could end up being a function with many local minimums (Hjorth-Jensen, 2019a). Thus making it hard to find the global minimum. Therefore a new cost function for logistic regression is defined:

$$C(\hat{\beta}) = -\sum_{i=1}^{n} y_i(\beta_0+,...,+\beta_N x_i) - log(1 + e^{(\beta_0+,...,+\beta_N x_i)}). \tag{3.4}$$

y_i denotes discrete outcomes. This function is known as the cross entropy, which can be used to give an estimate of how well the classifier performs (Hjorth-Jensen, 2019a). This function is minimized through gradient descent methods, which employ the derivative of the cost function:

$$\frac{\partial C(\hat{\beta})}{\partial \hat{\beta}} = -\mathbf{X}^T(\hat{y} - \hat{p}), \tag{3.5}$$

where p is the predicted likelihood, as given by the sigmoid function (section 3.3.3). By evaluating equation 3.4, it is evident that if $y_i = 0$, the cost is given as $-log(1 + e^{(\beta_0+,...,+\beta_N x_i)})$. Moreover, this means that if the cost is low, the term $e^{\beta_0+\beta_1 x_i}$ has to be small. This can be achieved when $\beta_0+,...,+\beta_N x_i \to -\infty$. If $y_i = 1$, the contribution to the total cost from one datapoint is given as $(\beta_0+,...,+\beta_N x_i) - log(1 + e^{(\beta_0+,...,+\beta_N x_i)}$. Given that $\beta_0+,...,+\beta_N x_i \to \infty$, the cost will tend towards zero.

3.2.1 Gradient descent

As mentioned in section 3.1, not all minimization methods have an analytical solution for the derivative. When working with such methods, the gradient descent algorithm can be used as an iterative process for finding the minimum of the cost

function. Gradient descent adjust the weights in the direction where the gradient of the cost function is large and negative (Mehta et al., 2019). This ensures that one are on the right path to finding the local minimum of the cost function. Finding the local minimum of a cost function is relatively easy, but finding the global minimum is more demanding. Therefor the choice of cost function must be adapted to the relevant task at hand. Each weight, w is repetitively adjusted according to the functions:

$$\mathbf{v}_i = \eta \nabla_w C(w_i), \tag{3.6}$$

$$w_{i+1} = w_i - \mathbf{v}_i, \tag{3.7}$$

here $\nabla_w C(w)$ is the gradient of $C(w)$, and η denotes the learning rate, which controls the step length taken each iteration towards the minimum (Mehta et al., 2019). This can be seen as moving downhill in an n-dimensional space. The learning rate is a ML hyperparameter that should be tuned. Figure 3.1 shows the same minimization problems, but solved with different learning rates. When employing a large learning rate, the gradient descent algorithm might struggle with converging, because it jumps over the minimum. Using a smaller learning rate might lead to a more stable descent towards the minimum, but is more computationally demanding. Small learning rates can also mean that the algorithm gets stuck in a local minimum (Mehta et al., 2019). A more refined approach could be to use an adaptive η which decays as it approaches the minimum.

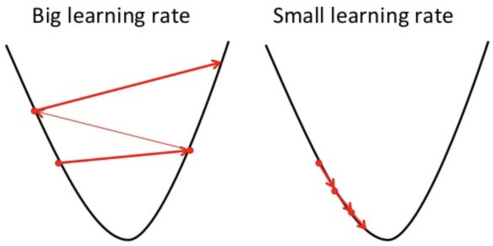


Figure 3.1: Learning rates in gradient decent (Unknown, 2019)

A variant of gradient decent is called Batched Gradient Descent. An approximation of the gradient is now calculated on a subset of the data, called a minibatch, instead of calculating it on the whole dataset. This is commonly used in Neural Networks (Hjorth-Jensen, 2019b) . This variant of gradient decent has several advantages: the

calculations of the gradient is significantly faster, and if the minibatches are created randomly, the stochasticity decreases the chance that the algorithm becomes stuck in a local minimum (Hjorth-Jensen, 2019b).

3.2.2 Accuracy metrics

It is important to use accuracy metrics when evaluating classification algorithms. In this report, Accuracy score, the confusion matrix and F1 score have been employed. The authors also acknowledge AUC as a relevant accuracy metric, however this will not be used in this study.

Accuracy score

Accuracy score gives an indicator of correctly classified data compared to the total data. With the accuracy score, the performance of classification algorithms can be evaluated, and is calculated as shown in equation 3.8.

$$Accuracy = \frac{1}{N}\sum_{i=1}^{N} I(t_i = y_i), \tag{3.8}$$

here t denotes the target values, and y denotes the predicted values. Put in different words, the accuracy score counts the total correct predictions and divides by the total number of samples (Mehta et al., 2019). However, if the dataset is biased (i.e containing a large majority of one class), the accuracy score can give a false impression of how much the algorithm has learned. In such cases, the confusion matrix or the F1 score can be used.

Confusion Matrix

The confusion matrix contributes additional information as compared to the accuracy score, and has gotten its name because it represents what labels the classifier is prone to confuse (Pedregosa et al., 2011). In the case of a skewed dataset, where 90 % of the data belong to one label, a classifier can get 90 % accuracy by simply guessing one label, and never learn the characteristics of the other label. By evaluating the confusion matrix, these kinds of pitfalls will become clear. Figure 3.2 shows a sketch explaining the confusion matrix. Another method for measuring classification performance is the F1 score, which employs some of the values computed in the confusion matrix to calculate a performance scalar (Pedregosa et al., 2011).

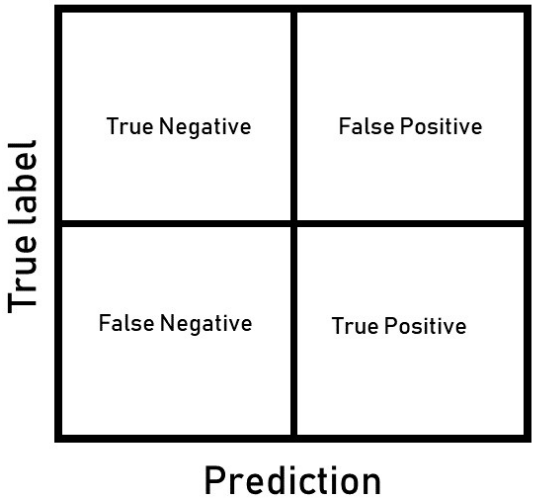


Figure 3.2: The confusion matrix, as defined by `scikit-learn`.

F1-score

Another method to look at the performance of the classification is F1-score. This method considers the precision p, and the recall r. Mathematically, the F1 score can be expressed as:

$$F_1 = 2 \times \frac{p \times r}{p + r}, \tag{3.9}$$

where p and r is defined in equations 3.10 and 3.11.

$$p = \frac{|true\ \ positives|}{|true\ \ positives| + |false\ \ positives|}, \tag{3.10}$$

$$r = \frac{|true\ \ positives|}{|true\ \ positives| + |false\ \ negative|}. \tag{3.11}$$

Precision gives a ratio of how well the classifier predicts a true positive versus the total number of times it predicts a positive. Recall is a measure of how accurately the classifier predicts a true positive versus the total number of positives (Derczynski, 2016). F1-score penalizes therefor false positives and false negatives.

3.3 Artificial Neural Networks

Artificial Neural Networks (ANN) can be seen as a tool that when presented with data, can learn hidden structures in this dataset, which thereafter can be used for prediction. The name ANN originates from how these computational tools mimic biological neurons (Hjorth-Jensen, 2019b). There exists a myriad of different Neural Network structures, which can easily make the terminology seem rather blurred. To keep the terminology clear, only Multi Layer Perceptron (MLP) networks will be focused on in this report. MLPs are often referred to as fully connected networks with more than three layers (Mehta et al., 2019). "Fully connected" refers to how each node is connected to all the nodes in the adjacent layers. In the simplest case, with only three layers, these layers are the input layer, one hidden layer, and the output layer (Mehta et al., 2019). Figure 3.3 shows a MLP network with two hidden layers.

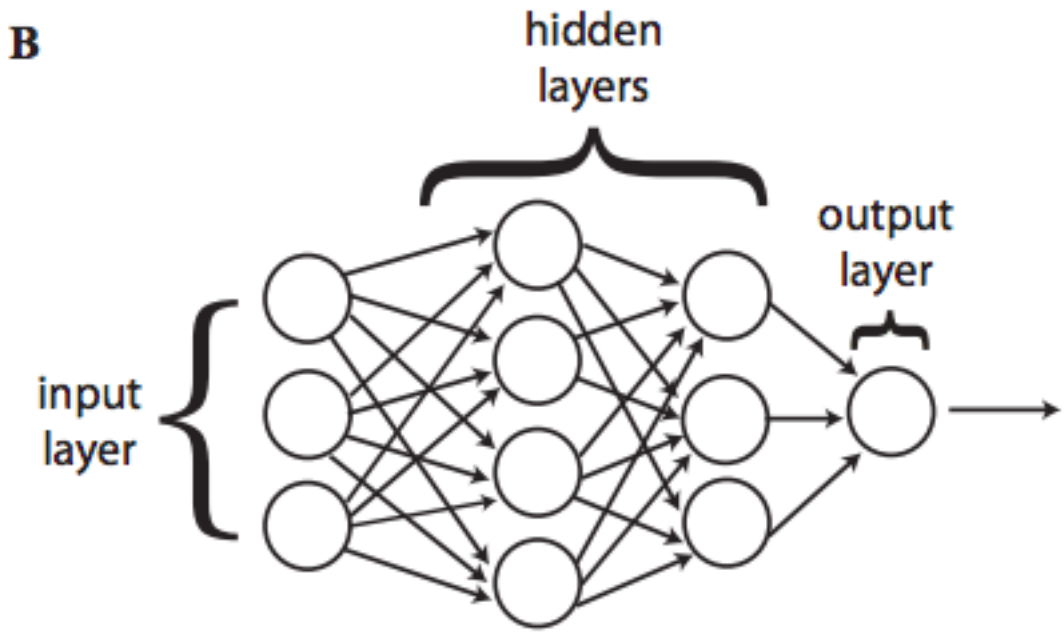


Figure 3.3: Structure of a MLP network with two hidden layers (Mehta et al., 2019).

The learning process of such networks can be split into two parts: feed forward, and back propagation. However, before delving into these processes, it is necessary to examine the workhorses of the neural network, namely the nodes.

Node

Figure 3.4 shows a single node. The node first calculates a weighted sum of the input data (can either be from input layer or output from preceding nodes). The output is then computed by sending the weighted sum through an activation function (step function in figure 3.4). The output of the node is then sent to one or more nodes in the next layer depending on the network structure. Note that it is common to add a bias to this weighted sum (Géron, 2017).

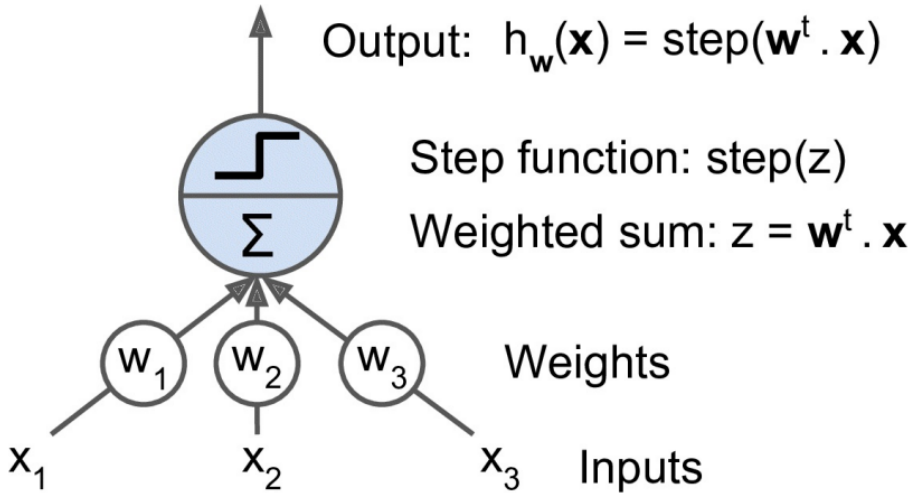


Figure 3.4: A single node without added bias. Note that this particular kind of node is known as a perceptron, as the activation function is a threshold function (i.e either active or inactive) (Géron, 2017).

3.3.1 Feed forward

The feed forward process of the neural network can be seen as an extension of the processes inside a single node. For a fully connected layer (which is the case for MLPs), this process can be expressed mathematically as:

$$y_i^l = f^l(y_i^{l-1}) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_i^{l-1} + b_i\right), \tag{3.12}$$

where y is the output, l is the current layer, f is the activation function, N_{l-1} denotes the size of layer $l-1$, and b denotes the bias. This process is then repeated for each layer until reaching the output layer (Hjorth-Jensen, 2019b).

3.3.2 Back-propagation

When the feed forward cycle is completed, the error is computed. The error is then used for updating the weights through back-propogation (Hjorth-Jensen, 2019b). This error is then propagated back through the network to update the weights and biases. Let f denote an arbitrary activation function, while C and a denotes the cost function and activation function respectively. By employing the hadamard product ($\circ$), the error of the output layer can be written as:

$$\delta_j^L = f'(z_j^L) \circ \frac{\partial C}{\partial a_j^L}, \tag{3.13}$$

This error can be used to calculate the gradients w.r.t bias and weights. The gradient of the cost function w.r.t bias (b) is simply the error of the output layer

$$\frac{\partial C}{\partial b_j^L} = \delta_j^L, \tag{3.14}$$

while the gradients of the weights is given as the product of the error and the activation of second to last layer

$$\frac{\partial \hat{W}^L}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}. \tag{3.15}$$

These are the equations needed to kickstart the backpropogation algorithm. By using the chain rule however, it can be shown that replacing the output layer L with an arbitrary layer l yields

$$\delta_j^l = \sum_k \delta_k^{l+1} w_k^{l+1} f'(z_j^l). \tag{3.16}$$

by analogy to equation 3.15, the weight and bias can now be updated respectively as

$$w_j k^l = w_j k^l - \eta \delta_j^l a_k^{l-1}, \tag{3.17}$$

and

$$b_j^l = w_j^l - \eta \delta_j^l. \tag{3.18}$$

For a more through explanation of both the feed forward and backpropagation processes, the reader is referred to Hjorth-Jensen (2019b). An integral part of the neural networks is the activation functions. These will therefore be discussed now.

3.3.3 Activation functions

To fullfill the universal approximation theorem, some constrains are put on what constitutes an activation function. It must be non constant, bounded, monotonically increasing and continuous. One function that fullfills these criteria is the sigmoid function.

Sigmoid, step function and tanh

Sigmoid is one of the most widely used activation functions. It takes any real number, and outputs a number between zero and one.

$$\sigma(y) = \frac{1}{1+e^{-y}} = \frac{e^{y}}{1+e^{y}} \tag{3.19}$$

From equation 3.19, it follows that $p(-\infty) = 0$, $p(0) = 0.5$ and $p(\infty) = 1$. As mentioned, back-propagation employs the derivative of the cost function to update the weights. For exceedingly large, or small values input into the sigmoid function, its derivative is very small. This implies that some nodes become virtually inactive due to small gradients. The hyperbolic tangent $tanh(y)$ is a similar activation function. It can be shown that $tanh(y)$ is related to sigmoid through

$$tanh(y) = 2\sigma(2y) - 1. \tag{3.20}$$

It can therefore seem arbitrary which of these functions are used. However, as explained by LeCun et al. (2012), the choice matters for saturation of the gradients. Which in turn affects how fast the network learns. By plotting the derivative of the cost functions, it becomes evident that the hyperbolic tangent has a higher gradient around the origin, and can therefore in some instances converge faster. In figure 3.5, the first activation function is denoted Perceptron (also denoted step function). This function has only two outputs; zero below threshold and one above threshold. More importantly, its derivative is always zero (except at zero, where the derivative is not defined), making it unpractical for gradient descent methods. It can however be recognized as the derivative of the ReLU function.

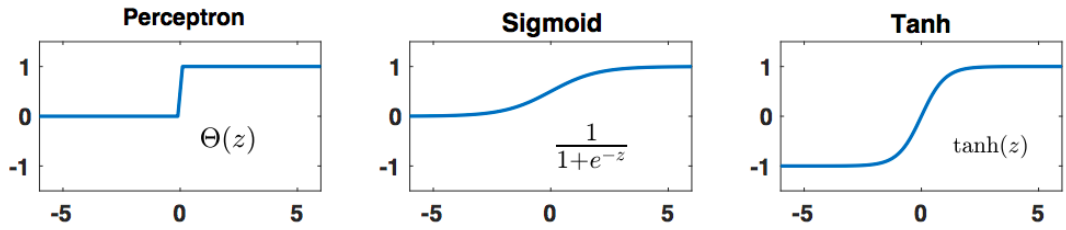


Figure 3.5: The possible non-linear activation functions for neurons (Mehta et al., 2019).

ReLU- family

ReLU is an abbreviation for Rectified Linear Unit. Mathematically ReLU is defined as $y = max(0, x)$. ReLU is therefore defined in the range of $0 \rightarrow \infty$. Unlike sigmoid, ReLU does not suffer from the problem of wanishing gradients, and it has been shown to train faster (Mehta et al., 2019). On the other hand, as ReLU has no upper boundary, the gradients can explode and lead to an unstable algorithm. This problem can be circumvented by clipping large values. ReLU is part of a larger family of similar activation functions. Two of these functions are shown in figure 3.6. Employing ReLU can lead to many dead nodes, if the input is smaller than zero. Leaky ReLU and the ELU function has been shown to perform better, as their derivative is nonzero for input values smaller than zero.

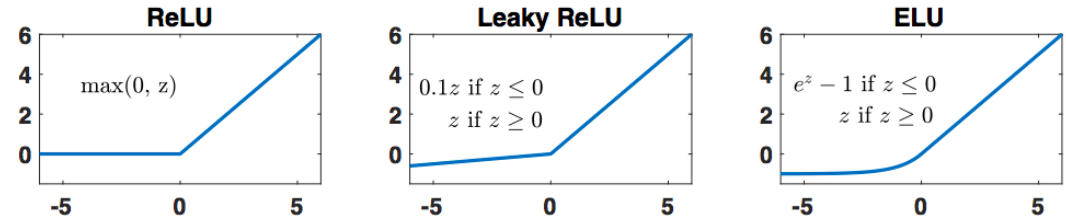


Figure 3.6: The possible non-linear activation functions for neurons (Mehta et al., 2019).

In multiclass classification methods an activation function called the softmax function is commonly used.

Softmax

The softmax function is commonly used in mulitclass classification methods, as logistic regression, and artificial neural networks (Mehta et al., 2019).

$$p(C = k|\mathbf{x}) = \frac{e^{(\beta_k 0 + \beta_k 1 x_i)}}{1 + \sum_{l=1}^{K-1} e^{(\beta_l 0 + \beta_l 1 x_i)}} (Hjorth - Jensen,\ 2019a). \tag{3.21}$$

In this study this activation function has not been used.

Dataset

The credit card dataset holds information about customers default payments (Dua and Graff, 2017). The payment data is taken from an important bank in Taiwan. The targets were credit card holders of the bank. The real probability of default is the response variable (Y), and the predictive probability of default is the independent variable (X).

This dataset use a binary response variable. If the payment is default the number is set to one, and if not the number is set to zero. The dataset includes 23 explanatory variables which are listed in table 4.1 (I-Cheng Yeh, 2007).

Table 4.1: Credit card data explanatory variables

Explanatory variables	
X1	Amount of the given credit (NT dollar)
X2	Gender (1 = male; 2 = female)
X3	Education (1 = graduate school; 2 = university; 3 = high school; 4 = others)
X4	Marital status (1 = married; 2 = single; 3 = others)
X5	Age (year)
X6-X11	Past monthly payment records (from April to September, 2005)
X12-X17	Amount of bill statement (NT dollar)(from April to September, 2005)
X18-X23	Amount of previous payment (NT dollar)(from April to September, 2005)

Among the total 25,000 observations, 5529 observations (22.12 %) are the cardholders with default payment. Some of these variables are categorical. Therefore these variables have been `onehot-encoded`. If this was not done, for instance being female would have been weighted more than being male (cf. table 4.1).

To compare the credit card dataset, the neural network code was tested on the Franke Function, which is a dataset used in Project 1. This dataset can be found in Tesaker et al. (2019).

Implementation

All the code can be accessed from the github repository (https://github.com/VemundST/Project2). The code can roughly be divided into two subgroups, Logistic Regression and Neural Network. Both of these are implemented as object oriented classes. The `README`-file in the repository explains where the files can be found.

Based on a 1-layer NN code (avaliable from Hjorth-Jensen (2019b)), the Neural Network has been further developed and implemented as a fully object oriented N-layer network. A neural network object can be created by calling the class ANN and defining some of the hyperparamters.

```
neural_net = ANN(lambda,
                 bias,
                 eta,
                 tolerance,
                 early_stop,
                 mode,
                 regularization)
```

After creating the network, the layers are added. Note that the weights are initialzed in accordance to kilde HE. ET AL 2015.

```
def add_layers(self, n_neurons, n_features, n_layers):
    for i in range(n_layers):
        if i == 0:
            layer_weights = np.random.randn(n_features[i], n_neurons[i])
        else:
            layer_weights = np.random.randn(n_features[i],\
             n_neurons[i])*np.sqrt(2/n_neurons[i-1])
        self.layers['w'+str(i)] = layer_weights
        layer_bias = np.zeros(n_neurons[i]) + self.bias
        self.layers['b'+str(i)] = layer_bias
```

Now that the weigths and biases are initialized, the next stage is training the network. This is done through the feed forward and backpropogation process described in section 3.3. To limit the extent of this section the reader is refereed to the github repository for further information regarding the implementation of these processes. When the neural network has converged, further training might lead to overfitting. It is therefore essential to devise an effective method for ending the training when the validation loss has converged. Such a method can be implemented as:

```
if tolerance > cost[i-1] -cost[i]:
    break
```

However, when combining this stopping method with stochastic gradient decent, the inherent randomness of the mini-batches can force the network to stop training long before reaching convergence. A more robust approach can be to implement the algorithm as:

```
if tolerance > mean(cost[i-N,...,i]) -mean(cost[i-n,...,i]):
    break
```

The two terms shown here can be recognised as running averages where $N > n$. By employing these running averages, the stochasticity induced by the mini-batches will be limited, thus leading to a more robust stopping algorithm. This method has been implemented as a stopping algorithm for the Neural Network. For further information about the implementation, confer the github repository.

Results

The first part of this section will go through the results obtained from the classification algorithms. As mentioned in section 4, the credit card data show a skewed distribution, where 78 % of the outcomes are zero. The classification algorithms will therefore be tested on a downsampled dataset (50/50 split of zeros and ones), and on the full skewed dataset. The second part of this section will present the results from employing neural networks on the Franke function. To increase the confidence of the results, k-fold cross-validation has been used as a resampling technique for the accuracy metrics. However, the loss plots show only the results from the last fold. A `random seed` was set for all neural network calculations to achieve consistent weight initialisation. The authors recognize that the best results can not be guaranteed, however this was done to reduce the number of tune-able parameters.

6.1 Classification

6.1.1 Logistic Regression

Gradient descent

Figure 6.1 shows the confusion matrix and log-loss for gradient descent, on the downsampled credit card data. The confusion matrix shows that 83% of the users that are expected to pay their dept, actually will pay. The algorithm correctly predict that 57% of the people that are not expected to pay their dept actually do not pay their debt. However, the algorithm wrongly classifies 43% of people to defaulting on their payment, and 17% to paying their debt. Both the training and test cost function show a large decrease in log loss over the first iterations. When increasing the iterations, the log loss stabilizes around 0.6 for both the datasets. The log loss for the training data is a bit larger than the test loss. However, this discrepancy might be explained by the splitting of the data, and might not be representative for all folds.

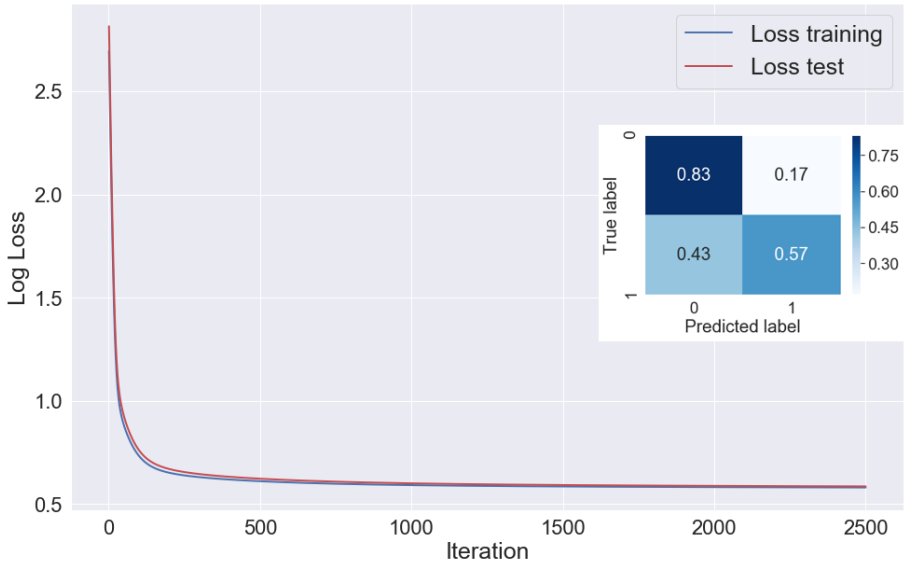


Figure 6.1: The log-loss for the kth-fold, with the confusion matrix for gradient descent for the downsampled dataset.

Figure 6.2 shows the confusion matrix and log-loss for gradient descent, where the original skewed dataset is used. This figure is quite similar to figure 6.1. However, the confusion matrix show a even higher percentage of true negatives, but lower percentage of true positives. The log loss behaves similarly in both figures.

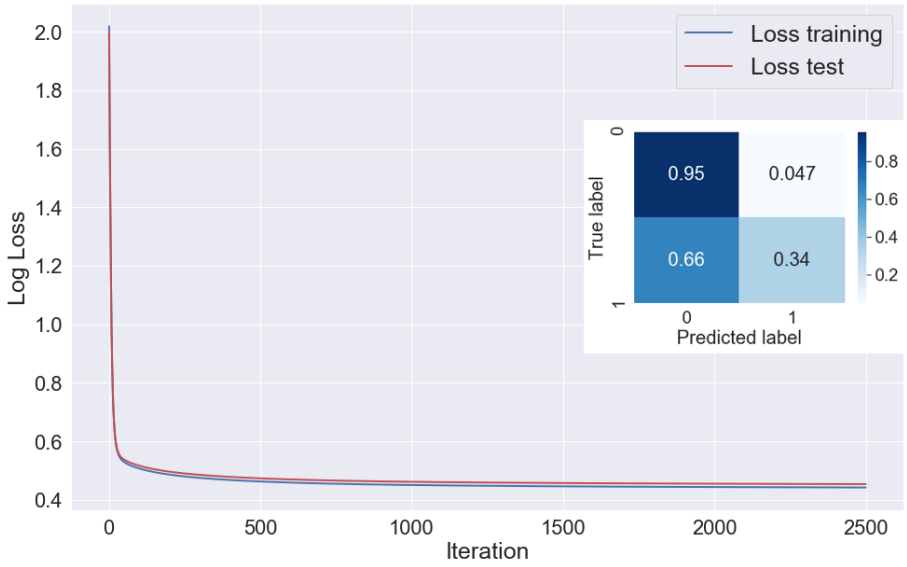


Figure 6.2: The log-loss for the kth-fold, with the confusion matrix for gradient descent for the original dataset.

Stochastic gradient descent

Figure 6.3 shows the confusion matrix and log-loss for stochastic gradient descent on the downsampled data. From the figure, one can see that the confusion matrix has the exact same result as for gradient decent in figure 6.1. However, the log loss for the training data is much more variable than for gradient decent.

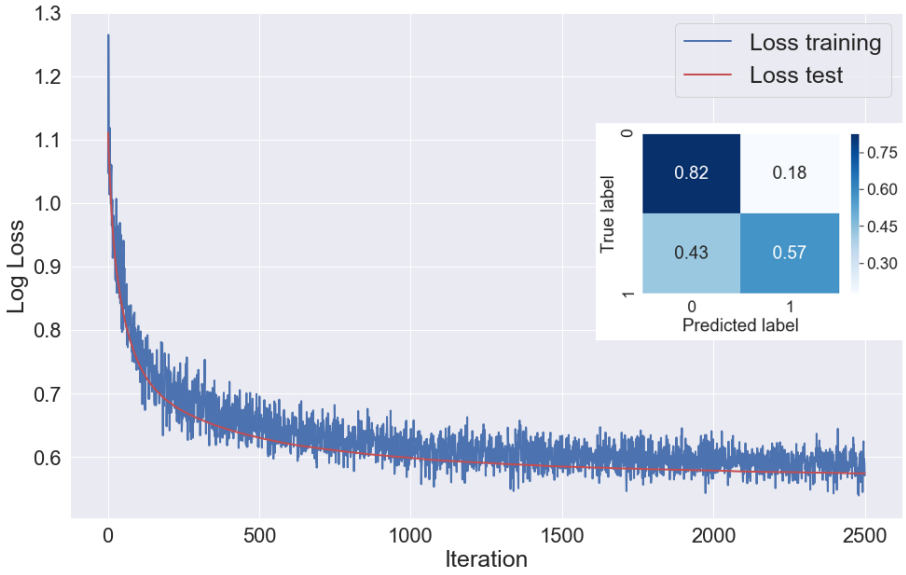


Figure 6.3: The log-loss for the kth-fold with the confusion matrix for gradient descent with downsampled dataset. Batch size= number of samples/20= 663.

Figure 6.4 is showing the confusion matrix and log-loss for stochastic gradient descent, where the original skewed data is used. By direct comparison with figure 6.2, it becomes evident that the confusion matrices are almost identical. Similarly to the SGD on the downsampled data, the training loss is more variable than the test loss.

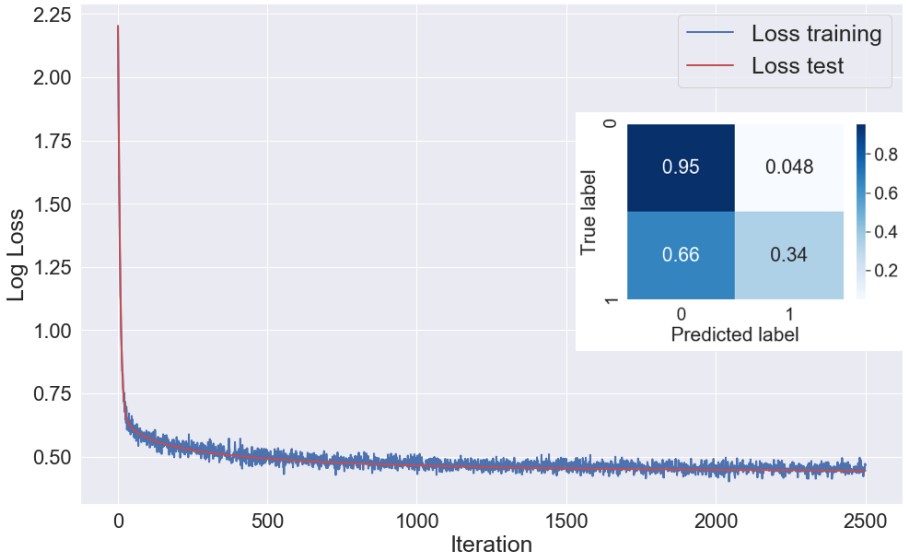


Figure 6.4: The log-loss with the the confusion matrix for stochastic gradient descent with the original dataset. Batch size= number of samples/20= 1500.

Table 6.1 shows the accuracy score and F1-score for the different logistic regressions, and is divided into downsampled data and original data. Within the different datasets, there are only small variations in accuracy and F1 score. However, the downsampled has lower accuracy score, but higher F1 score than the original data.

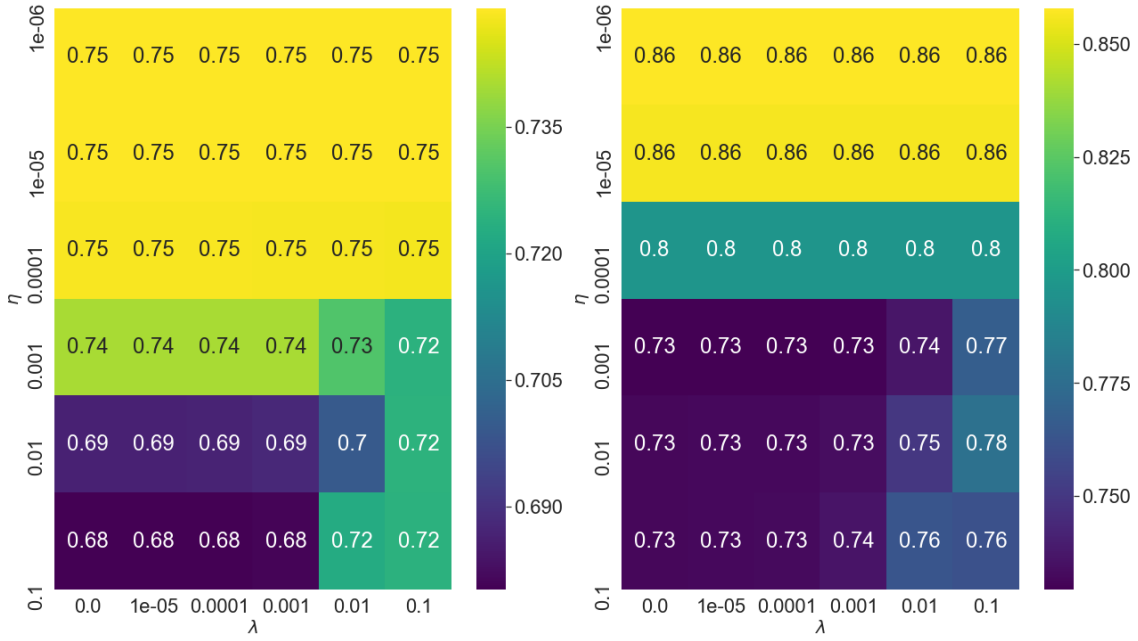
	Downsampled data		Original data	
	Accuracy score	F1 score	Accuracy score	F1 score
GD	70.1 %	0.65	81.6 %	0.44
SGD	69.9 %	0.65	81.8 %	0.46
NN	69.2 %	0.66	81.3 %	0.44

Table 6.1: Test accuracy score and F1 score for the downsampeld dataset, and the original dataset.

6.1.2 Neural network

The classification network has two hidden layers where the first layer has 50 nodes and ReLU activation, while the second layer has 20 nodes and sigmoid activation. To obtain the optimal results, the hyperparameters η and λ have been tuned. The resulting heatmaps for the downsampled and original dataset is shown in figures 6.5a and b respectively. Both heatmaps show a similar pattern, where loss values

decrease with smaller penalty parameter and higher learning rate. It should be noted that for the original data, the lowest values are actually found with $\eta = 0.001$. This value has not been used further in the calculations, which will be further discussed later.



(a) (b)

Figure 6.5: Heatmaps of the log loss values from the grid search, for (a) the downsampled data, and (b) the original skewed data. The random seed was set constant for these heatmap calculations.

Figure 6.6 shows the confusion matrix and log-loss for the neural network, using the downsampled dataset. The results are relatively similar to the other results obtained with the downsampled dataset. Similarly to the other stochastic gradient descent examples, the training loss is variable.

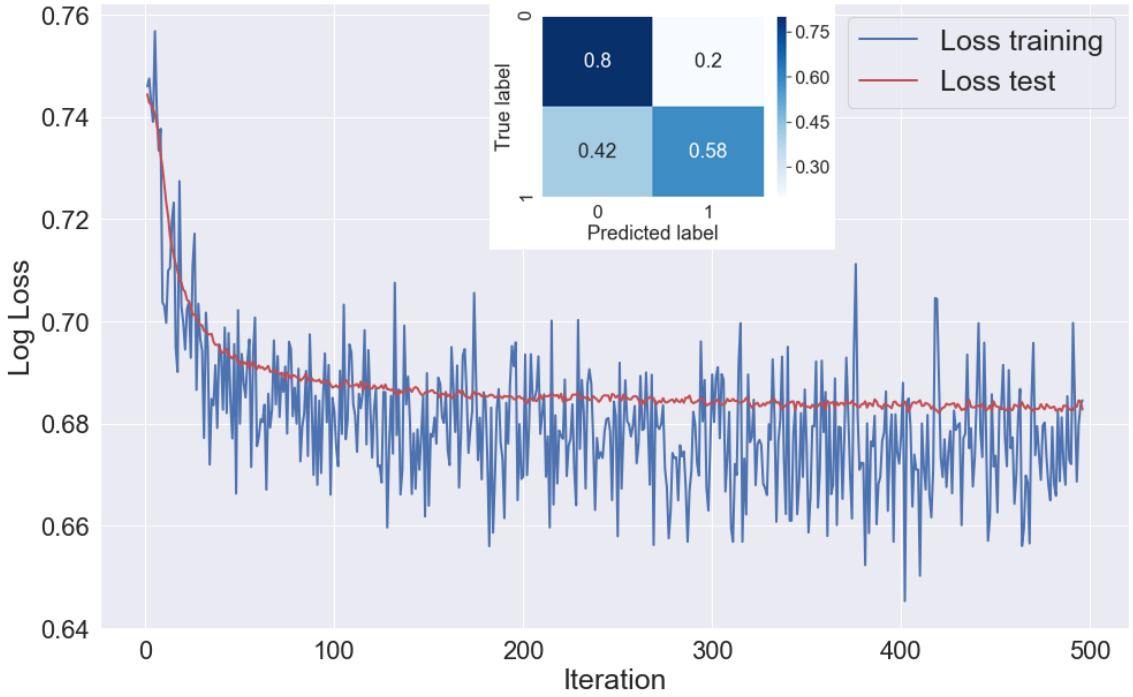


Figure 6.6: The confusion matrix and log-loss for Neural Network with the test data, on the downsampled dataset. Batch size= number of samples/20= 663.

Figure 6.7 shows the confusion matrix and log-loss for the neural network, using the test data on the original data set. In this figure there are two different confusion matrices. The left, is the confusion matrix after 15 epochs, which shows that the network almost exclusively predicts that people will pay their debts. After 15 epochs, the log loss grows before decreasing and stabilizing. The confusion matrix related to the final epoch exhibit similar results to figures 6.4 and 6.2.

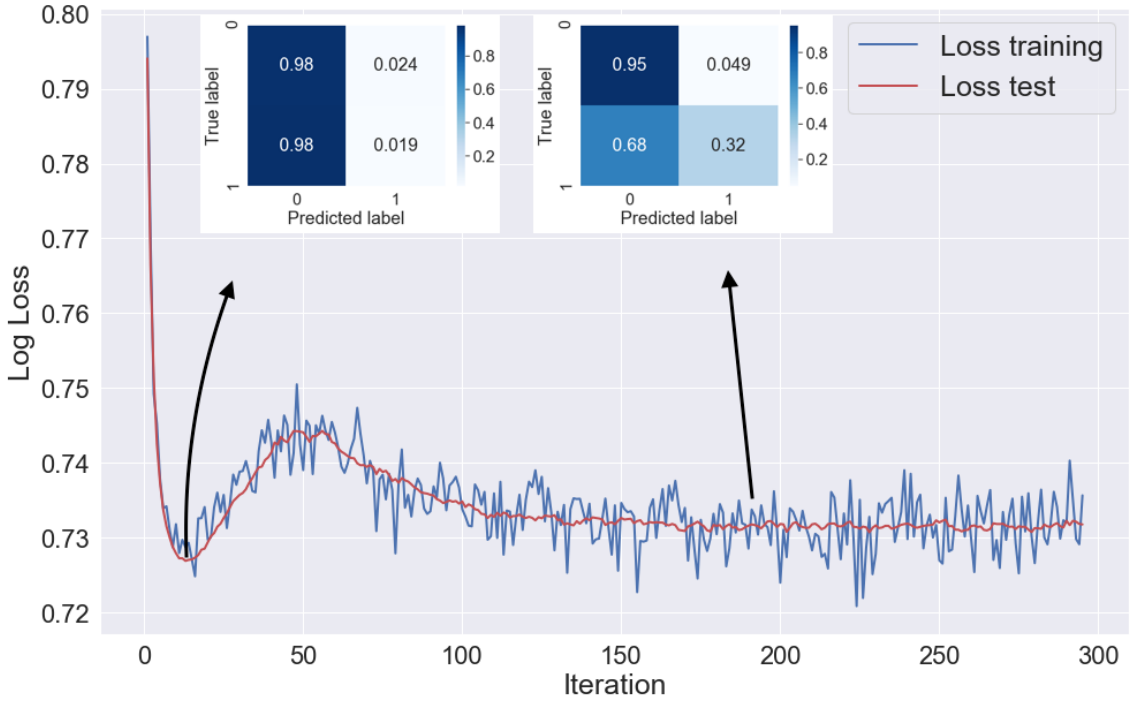
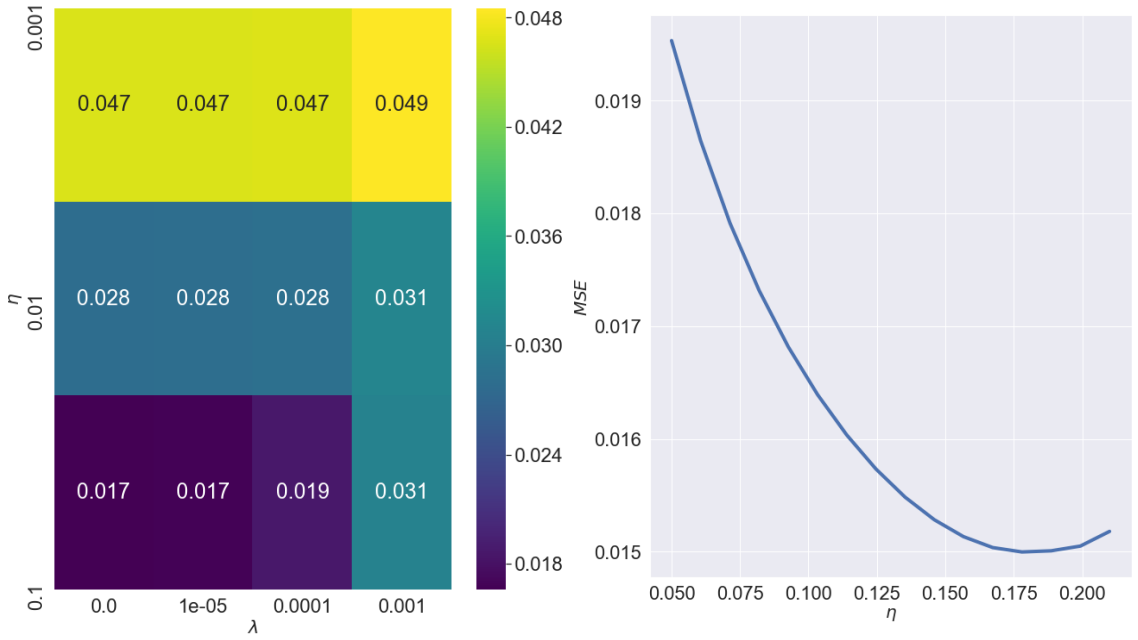


Figure 6.7: The log loss for Neural Network with the test data, on the original data. Batch size= number of samples/20= 1500.

6.2 Regression

To control if our neural network is representative for several datasets, the Franke function is introduced. The noise level is set to 0.1. The regression network has two hidden layers, with 20 and 10 nodes respectively. Both hidden layers use sigmoid as activation function. In the previous report polynomials in x and y up to fifth order were used. From the results it was found that Ordinary Least Squares gave the best MSE and R2-score, with respectively 0.06360 and 0.55190. In this section a regression analysis using the Franke function and neural networks will be presented. In the end the results will be compared to the results from project 1.

Figure 6.8a show the heatmap for the Franke function. Loss values decrease in the direction of lower λ and higher η. In order to get the optimal result, the learning rate η was further tuned, as seen in figure 6.8b.



(a) (b)

Figure 6.8: Heatmap of the MSE loss values from the (a) grid search and (b) refining η search using the Franke function. The random seed was set constant for these heatmap calculations

Figure 6.9 shows the log loss for the neural network when using the Franke function. Compared to the log loss figures that are already described, the log loss for the Franke function is much smaller. In this case the log loss stabilizes around 0.07. A variation in the loss training can still be seen.

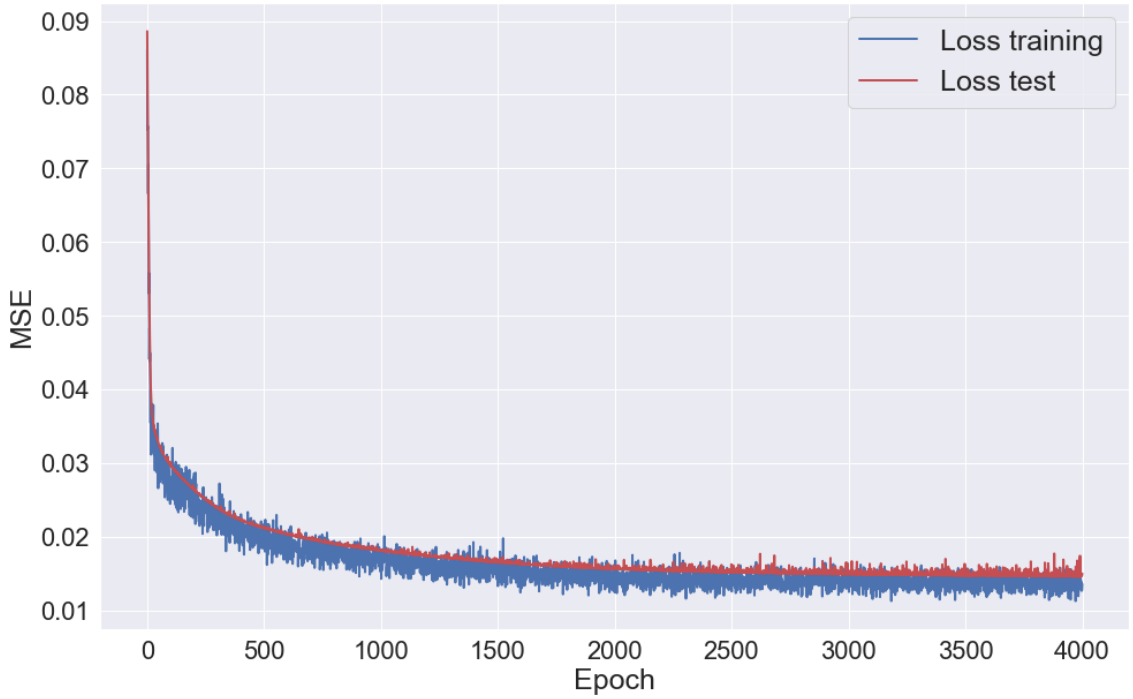
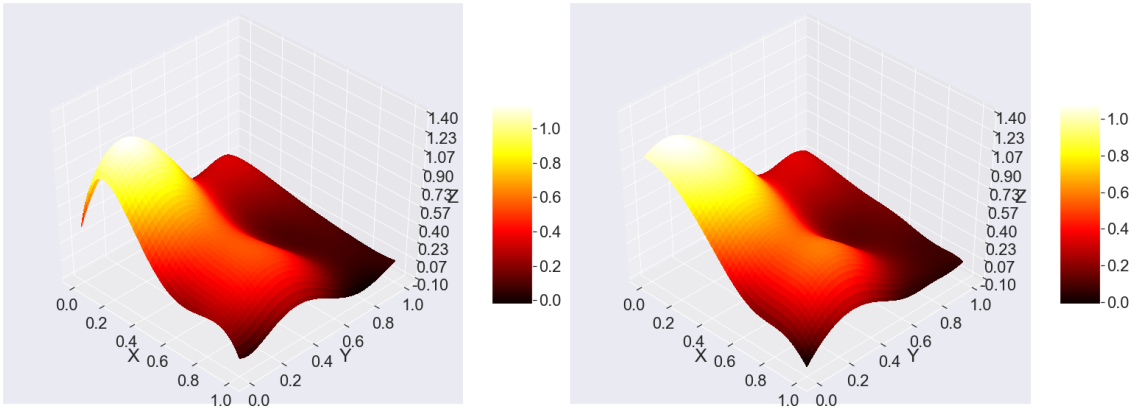


Figure 6.9: The MSE loss for neural network using the Franke function. In this example η is right on the edge of being too large.

Figure 6.10 shows the OLS regression (a) and the neural network regression (b) when using the Franke function. In figure 6.10a, OLS recreates the Franke function better than what the neural network does in figure 6.10b. For comparison the visualization of the true Franke function can be seen in section 4.1 in project 1 Tesaker et al. (2019).



(a) (b)

Figure 6.10: Regression analysis of the Franke function, for (a) OLS, and (b) neural network.

Discussion

When employing neural networks, there are several hyperparamters that can be tuned. In this report, only two of these parameters have been properly tuned, namely the penalty parameter λ and the learning rate η. These parameters can be seen in figure 6.5 and figure 6.8. From these figures one can conclude that the loss values decrease with smaller penalty parameter and higher learning rate. There are also some parameters that have not been tuned, such as: the batch size, the layer size, the number of layers and l2 vs l1 regularization. The results could be better if these parameters had been tuned.

Classification

As described previously, this report includes results for both the downsampled credit card data, and the original dataset. By direct comparison between figures 6.1 and 6.3, it can be seen that the results are almost identical. However, the training loss is much more variable when using stochastic gradient descent. This is mirrored when doing the same calculations for the full dataset, but there is a discrepancy in the variability of the stochastic gradient descent cases. The batch size is equal relative to the datasets (number of samples/20), but since the number of samples is lower for the downsampled data, more variability is expected due to smaller batch size.

There is a strong contrast between the neural networks when employed on downsampled and original data. While the log loss curve for the downsampled data is more or less monotonically decreasing, the log loss curve of the full dataset quickly drops to a minimum before rising and then decreasing again. By analysing the confusion matrix after only 15 epochs, it is evident that the network has only learned one class. Moreover, it is clear that is has not learned to distinguish the two classes. By analyzing the confusion matrix after the 300 epochs, it becomes evident that the network has better capability to distinguish the two classes, although it still overwhelmingly predict class zero. In this example, figure 6.7, the loss stabilizes at a higher level than the initial local minimum. It was mentioned earlier that the

lowest log loss values were found with $\eta = 0.001$, and that this value was not used further in the calculations. The local minimum is reached after 10-20 epochs. With a learning rate 100 times smaller than the optimal ($\eta \approx 0.1$), the local minimum seen in figure 6.7 is now reached after around 1000 epochs. The confusion matrices produced for $\eta = 0.001$ back up this claim. The log loss stabilizes at a higher level than the initial local minimum, which might be attributed to a too high learning rate, where the gradient points in the direction of a local minimum but constantly overshoots (cf. figure 3.2.1). Many of the open packages that are available in python (e.g `tensorflow` and `scikit-learn`) include an adaptive learning rate. The learning rate could then be lowered as the network approaches the minimum, thus not overshooting.

Table 6.1 presents the accuracy and F1 scores for the different runs. It is clear that the accuracy score is $\approx 12\%$ higher on the original data, but the F1 score is lower for all methods. By analogy to the confusion matrices (figures 6.1 - 6.4 and 6.6 - 6.7), the higher F1 score on the downsampled data indicates that the algorithms have learned better to distinguish between the two classes. The discrepancy can be attributed to the original skewed data (I-Cheng Yeh, 2007).

The choice of activation functions can greatly affect the results for the Neural Network. In this project work, only the sigmoid function and ReLU function have been used. As mentioned earlier, both the sigmoid and ReLu function can struggle with dead neurons. However, this has not proved to be a significant problem in this project work. An idea to remedy these shortcomings could be to introduce the ELU and Leaky ReLU instead of ReLU, and the tanh function for sigmoid.

Regression

One of the reasons that have made NNs so popular is their flexibility. Although not likely to yield the best results, the same network can be used for both classification and regression with only minor adjustments to the output layer. For regression it is common that the output layer has no activation function (Pedregosa et al., 2011), thereby predicting continuous values. The neural network chosen in this project work had two hidden layers with x and y nodes respectively.

The parameters η and λ were tuned by doing a grid search (figure 6.8). By direct comparison of figures 6.10a and b, it is clear that OLS describes the underlying Franke function better than the Neural Network, but it is also clear that the network picks up in some of the larger structures. Evaluating the final k-fold cross-validated MSE and figure 6.9 also shows that the the error moves towards the noise (0.01 for 0.1 noise factor). Some possible ways to improve the results could be to add more layers, tuning the layer sizes and experimenting with different activation functions.

Conclusion

Through this project two methods have been tested for classification, namely logistic regression and Neural Networks. These were tested on the original and downsampled credit card dataset. Both the Neural network and logistic regression show similar results both in terms of accuracy and F1 score. However, there is a large discrepancy between the original skewed data and the downsampled data. Whereas the accuracy is higher for the skewed data, the F1 score is lower, indicating that algorithms are biased towards predicting zeros. To ensure stability, k-fold cross-validation was used as resampling technique.

OLS regression yielded the best score in terms of MSE and R2 score for the regression case. However, the neural network yields promising results, and the results might improve with further tuning. As stated earlier, one of the advantages of neural networks are their flexibility. However, this can be seen as a double edged sword due to the large amount of tuneable hyperparamters. Moreover, this implies that it is almost impossible to find the "perfect" network structure, and a cost effective method for choosing the best hyperparameter is hard to come by.

Bibliography

Leon Derczynski. Complementarity, f-score, and NLP evaluation. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 261–266, Portorož, Slovenia, May 2016. European Language Resources Association (ELRA). URL https://www.aclweb.org/anthology/L16-1040.

Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2017.

Morten Hjorth-Jensen. Data analysis and machine learning: Logistic regression, 2019a. URL https://compphysics.github.io/MachineLearning/doc/pub/LogReg/html/LogReg.html.

Morten Hjorth-Jensen. Data analysis and machine learning: Neural networks, from the simple perceptron to deep learning, 2019b. URL https://compphysics.github.io/MachineLearning/doc/pub/NeuralNet/pdf/NeuralNet-minted.pdf.

Che-hui Lien I-Cheng Yeh. The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. *Physics reports*, 2007.

Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre GR Day, Clint Richardson, Charles K Fisher, and David J Schwab. A high-bias, low-variance introduction to machine learning for physicists. *Physics reports*, 2019.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Astrid Tesaker, Vemund Thorkildsen, and Sofie Tunes. Project 1: Regression analysis and resampling methods, 2019. URL https://github.com/VemundST/Project1.

Unknown. Learning rate, 2019. URL ://towardsdatascience.com/smart-discounts-with-logistic-regression-machine-learning-from-scratch-part-i-3c242f4ded0.