FYS-STK4155 - Project 2
Classification and Regression - from Linear and Logistic Regression to Neural Networks

Hannah Lovise Okstad Ekeberg　　Idunn Aamnes Mostue

November 2019

All code and data used in this project can be found at:
https://github.com/HannahEkeberg/FYS-STK4155/tree/master/Project2

Abstract

In this project, we wanted to analyze how well logistic regression and neural network perform on a binary classification problem. We analyzed a dataset with creditcard-details from a bank i Taiwan, and wanted to predict whether a cardholder would be able to pay back. We used a stochastic gradient descent in a logistic regression solver and a neural network to find accuracy scores. For logistic regression, we got an accuracy score of 0.7057 on the validation data, and for neural network, we got bad results (suspecting an error in the code), however Scikit-Learn's algorithm gave 0.7022 on the validation data. We applied the neural network code in regression-analysis on the Franke function to see how well neural network perform in linear regression, and got a mean squared error of 0.0496. In comparison, Lasso with $\lambda = 10^{-5}$ gave a mean squared error of 0.0149.

Contents

1 Introduction

In this project we try to recreate some of the results done by Yeh & Lien (2007) [1]. In Taiwan 2006, banks over-issued cash and credit cards to people who were not able to pay back. With different cardholders as targets, we want to look at whether or not the cardholders would be able to pay back, based on different parameters. Properties of logistic regression and neural networks will be investigated on the creditcard dataset, from an important bank in Taiwan looking from april-september 2005 (taken from UCI). We will also look on how the neural network performs on a linear regression problem, the Franke function , and compare that result with error from OLS, Ridge and Lasso to conclude what gives the best estimation best on mean squared error.

With this report we wanted to answer the following questions:

1. Is there any difference of classification accuracy between the two datamining techniques logistic regression and neural network?
2. Could the estimated probability of default produced from data mining represent the real probability of default?

In order to answer these questions, we have estimated precision scores on the two different methods, and looked on how well the models classifies on behalf of the confusion matrix.

This report proceeds in six chapters. Chapter 2 provides theory on the regression models and neural network used for the analysis, as well as optimization and accuracy validation techniques. In chapter 3 we introduce the credit card data, and explain how we implemented the machine learning methods for for both classification and regression analysis. Further, in chapter 4 we present our main results, before discussing our main findings in chapter 5. Lastly, chapter 6 give a short conclusion of the project.

2 Theory

In this chapter we introduce some theoretical background on two regression methods used in our analysis in section 2.1. Further, in section 2.2 we touch on how to optimise the output of our models. Section 2.3 introduce some background theory on artificial neural networks. Lastly, section 2.4 introduce methods for accuracy validation.

2.1 Regression

2.1.1 Linear regression revisited

Linear regression models are used to predict data for a continuous variable by learning the coefficients of a functional fit [2]. In cases where linear regression are applicable, there is a functional variable y which is linearly dependable on independent variables X. Regression finds a predictable y-value, given by optimization parameters β:

$$\hat{y} = \hat{X}\hat{\beta} \tag{1}$$

Here, we use the three most common linear regression models, namely Ordinary Least Square (OLS), Ridge and Lasso, in order to find the optimal parameters of beta - minimizing the cost function. We define a general cost function - The Mean Squared Error (MSE):

$$C(\beta) = \sum_{i=0}^{N}(y_i X_i \beta)^2 \tag{2}$$

To avoid overfitting for large parameters of β, we introduce a regularization (penalty) parameter λ. For Ridge regression, the sum of all β^2 are added, and for Lasso, the sum of all β are added to the cost function. The unique solution for the optimal parameters of β for OLS and Ridge are then respectively presented as:

$$\hat{\beta}_{\text{OLS}} = (\hat{X}^T\hat{X})^{-1}\hat{X}^T\mathbf{y} \tag{3}$$

$$\hat{\beta}_{\text{Ridge}} = (\hat{X}^T\hat{X} + \lambda I)^{-1}\hat{X}^T\mathbf{y} \tag{4}$$

For Lasso, there is no unique solution so we use the inbuilt class from scikit-learn[1].

[1]Description of Scikit-Learns inbuilt class for the regression model Lasso can be found at: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html

2.1.2 Logistic regression

The following paragraph is based on the lecture notes *'Data Analysis and Machine Learning: Logistic regression'* by Morten H. Jensen[2].

In comparison to linear regression where the aim is to predict data for continuous variables, one also have cases where we wish to predict data of discrete variables. In such cases, the dependent variables y_i only take values from $k = 0, ..., K-1$ classes, and the goal is to predict the target classes from the independent variables X.

The most simple case is with a binary outcome with two possible classes; $y_i = \{0, 1\}$, for instance "yes" or "no". To get such discrete outputs, one need some function that can map the output and give values of the possible classes $\{0, 1\}$. Such models can be 'hard classifiers' or 'soft classifiers'. In this project we will discuss the latter. The most common of soft classifiers is the Logistic function, which gives the probability of an outcome rather than an actual value.

Through Logistic regression the probability of the independent variable X_i to be in either of the classes y_i is usually represented by the Sigmoid function:

$$p(x) = \frac{1}{1+e^{-x}} \tag{5}$$

For our simple case of binary classification with two optimization parameters, $\hat{\beta} = \{\beta_1, \beta_2\}$, the probabilities for $y = 1$ or $y = 0$ from equation 5 becomes:

$$p(y_i = 1|X_i, \hat{\beta}) = \frac{1}{1+e^{-(\beta_0+\beta_1 X_i)}} \tag{6}$$

$$p(y_i = 0|X_i\hat{\beta}) = 1 - p(y_i = 1|X_i, \hat{\beta}) \tag{7}$$

The Maximum Likelihood Estimation (MLE) can be used in order to define the total likelihood for all possible outcomes of a data set. The total likelihood for all possible outcomes given a binary classification can be approximated as the multiplication of each individual probability outcome from equations (6) and (7):

$$P(\{(X, y)\}, \hat{\beta}) = \Pi_{i=1}^{n} p(y_i = 1|X_i, \hat{\beta})_i^y \cdot p(y_i = 0|X_i\hat{\beta})^{y_i - 1} \tag{8}$$

From equation (8) we obtain the log-likelihood and express it as the cost/loss function:

$$C(\hat{\beta}) = \sum_{i=1}^{n} (y_i \log(p(y_i = 1|X_i, \hat{\beta}) + (1-y_i)\log(1 - p(y_i = 1|X_i, \hat{\beta})) \tag{9}$$

Through logarithmic properties we can rewrite it further, and note that the cost function now is only the negative log-likelihood, also referred to as the cross entropy:

$$C(\hat{\beta}) = -\sum_{i=1}^{n} [y_i(\beta_0 + \beta_1 X_i) - \log(1 + e^{\beta_0+\beta_1 X_i})] \tag{10}$$

2.2 Optimization

When fitting our model, we wish to find values of β that minimizes our cost function[4]. One method is to use gradient descent, which uses the gradient of the cost function with respect to the variables β in order to find the minimum cost.

$$\nabla C(\hat{\beta}) \tag{11}$$

In our case of a negative-loglikelihood, we have a convex function of weights $\hat{\beta}$ for positive real numbers. Gradient descent starts by initializing random parameters β. When the derivative of the variables

$$\frac{\partial C(\beta)}{\partial \beta} \tag{12}$$

is negative, it will move in the direction in which it approaches the minimum. The step size is given by

$$\text{stepsize} = \frac{\partial C}{\partial \beta} \cdot \eta \tag{13}$$

where η is the learning rate. The step size gives larger steps when the optimizing parameters give a prediction far away from the minimum, and smaller steps when it approaches the minimum. The gradient descent stops when it reaches a tolerance, and the maximum number of steps is reached, called epochs. When handling large sets of data as well as many parameters, gradient descent can be inefficient. Instead we can use *stochastic gradient descent*.

Stochastic gradient descent is useful for large sets of data. For n data points one can divide the data set into a number of n/M minibatches, M. Then, we only use the sum of a few random chosen minibatches in each step. This results in a more stable estimate in fewer steps, as well as it is more cost effective to program.

2.3 Artificial Neural Networks

The main idea behind Neural Networks in Machine Learning is that we do not want to tell the computer how to solve a problem, but rather train it by observational data so it can figure out its own solution to the problem given[7]. A mathematical model of the neural network was introduced in 1943 by McCullon and Pitts, which laid the fundamental idea of the a Artificial Neural Network (ANN). ANN is a machine learning technique similar to the biological neural network that make up our brain (Marsland, chapters 3.2-3.3 [5]). In the 1950s-1960s Frank Rosenblatt developed the Perceptron Artificial Neuron. This neuron is a simple, single layer linear classifier, which takes several binary inputs and produces a single binary output. Rosenblatt also introduced an added real numbered weight, expressing the importance of the respective input to the output, which in turn is determined by whether the weighted sum $\sum_j w_j x_j$ is less or greater than some chosen threshold value.

$$\text{Output} = \begin{cases} 1, \text{if} \sum_j w_j x_j \leq \text{threshold} \\ 0, \text{if} \sum_j w_j x_j \geq \text{threshold} \end{cases} \tag{14}$$

By setting $x \equiv \sum_j w_j x_j$ and move the threshold to the other side of the equation and replacing it with a bias $b \equiv -\text{threshold}$ one can simplify the above equation for:

$$\text{Output} = \begin{cases} 1, \text{if}(z) \leq 0 \\ 0, \text{if}(z) \geq 0 \end{cases} \tag{15}$$

where $z = w \cdot x + b$.
We then see from equation (15) that we can modify the outputs by changing the weights and biases. However, a small change in bias or weight may cause the output of the preceptor to flip and potentially change the behaviour of the network [7]. Instead, we want to introduce some activation function. In this project we focus on the *sigmoid*, *hyperbolic tangent* and *rectified linear unit* (ReLu).

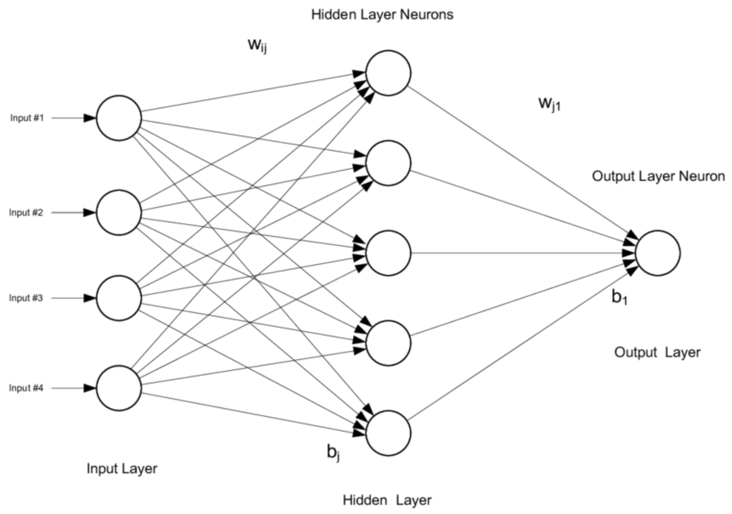


Figure 1: Structure of a Artificial Neural Network. The network consist on an input layer, one hidden layer, and an output layer. (From [6])

The Perceptron is the simplest example of a ANN, but in this project we will consider a Multi Layered Perceptron network (MLP). As we can see from Figure 1 the structure of a MLP is similar to the single Perceptron, but in addition it also contains 'hidden' layers in between the input and the output layer. One can have multiple hidden layers with n neurons in each layer, where each neuron from one layer will get input from all neurons of the previous layer and give its output to all neurons on the next layer. Such a structure where output from one layer is used in creating the input for the next one is called a *feedforward* neural network. This structure cause neurons to fire for a set amount of time, then these signals can cause new neurons to fire, and so on.

As we first mentioned, we want to train the computer with some observational data, for it to solve a given problem by its own. Therefore, we assign it with some date for training and some data for validation. The goal is to make it produce some output approximated as close as possible to the actual target data. To quantify the performance we validate the output data against the target data through some cost function C(w,b), dependent on earlier weights and biases of the network. This cost function will give us some error that we want to adjust so it becomes as small as possible. We can do so by a gradient descent method called *backpropagation*, where we apply the chain rule working our way backwards through the network and upgrade our weights and biases along the way.

By defining the weighted sum $z_j = \sum_{i=1}^{n} w_{ij} x_i + b_j$ for the jth neuron in our MLP, with a chosen activation function $f(z_j) = a_j$, we can write out the changes in the cost as functions of weights (16) and biases (17):

$$\frac{\partial C}{\partial w_{jk}^l} \tag{16}$$

$$\frac{\partial C}{\partial b_j^l} \tag{17}$$

Then we introduce

$$\delta^l = \frac{\partial C}{\partial z_j^l}, = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z') \tag{18}$$

$$\delta^L = \frac{\partial C}{\partial a_j^l} \sigma'(z_j^L) = \nabla_a C \odot \sigma'(z') \tag{19}$$

where δ^l is the error in the jth neuron in the lth layer, and δ^L is the error in the output layer. $(\frac{\partial C}{\partial a_j^l})$ tells us how fast the cost function is changing as a function of the jth output activation, and $\sigma'(z_j^L)$ is how fast the activation function σ is changing at z_j^L.

So, in principle we set a^1 corresponding to the input layer. Then we feed forward for each $l = 1, 2, ..., L$, and further compute $z^l = w^l a^{l-1}$ and $a^l = \sigma(z^l)$. From this we can compute the output error (19), and then backpropagate the error for each $l = L-1, L-2, ...$ and compute δ^l. Finally we calculate the gradient of the cost function and see how it changes at the weights (20) and biases (21):

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{20}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{21}$$

2.4 Accuracy Validation

When predicting data, we need some way of validating the accuracy of our predictions. In this project we will use Mean Squared Error (MSE)[2] for validation of our continuous data. For data of discrete variables we predict targets of different classes and use confusion matrices and accuracy score as tools for validation. This next section is based on the paper *'Beyond Accuracy, F-Score and ROC: A Family of Discriminant Measures for Performance Evaluation'* by Marina Sokolova et al.[8].

Confusion Matrix

A confusion matrix give us some statistics about the predicted values which we can use in validation of our model. Figure 2 shows an example of a confusion matrix of target classes zero and one. The first value of the matrix indicates the number of predicted zeros that where true zero, the second is the number of predicted ones that where false one, the third is the number of predicted zeros that where false zero, and the last one is the number of predicted ones that where true one.

Accuracy Score

[2]For more theory on MSE, please check out Project 1 at https://github.com/HannahEkeberg/FYS-STK4155/tree/master/Project1

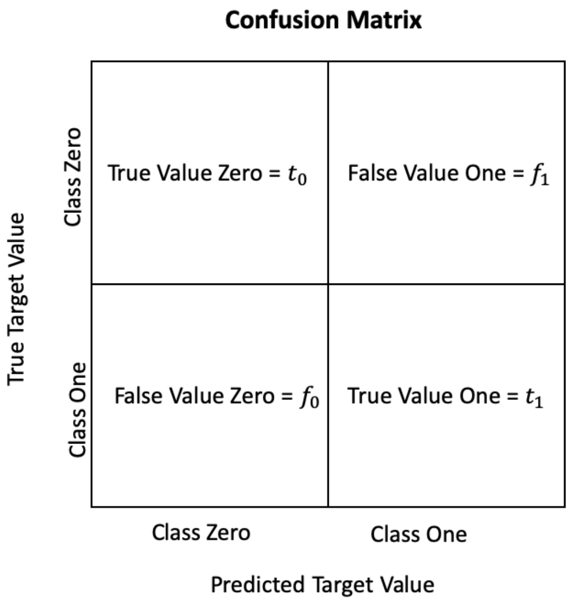


Figure 2: Confusion Matrix of target class Zero and One

As we can see from equation (22) the accuracy score is a measure of how many correct target values the model predict divided by the total number of targets. A perfect classifier will have an accuracy equal to 1, hence we want our accuracy score to be as close to 1 as possible. However, it is important to remember that this score is only a measure of the total correct predictions, and does not tell us anything about the accuracy of each target class individually.

$$Accuracy = \frac{t_0 + t_1}{t_0 + t_1 + f_0 + f_1} \tag{22}$$

Recall and Precision

To get a better grip of what is actually going on in the confusion matrix, we can calculate the *Recall* and *Precision* for each of the classes. Focusing on one class at a time, the recall gives a measure of the models ability to recognize the true class. Equation (23) and (24) gives the recall value for class zero and one respectively. And the precision is a measure of the models ability to predict only the relevant target values. In other words, when it predicts a target to be of a category, how often is it true. Equation (25) and (26) gives the precision value for class zero and one respectively.

$$Recall\ class\ zero = \frac{t_0}{t_0 + f_1} \tag{23}$$

$$Recall\ class\ one = \frac{t_1}{t_1 + f_0} \tag{24}$$

$$Precision\ class\ zero = \frac{t_0}{t_0 + f_0} \tag{25}$$

$$Precision\ class\ one = \frac{t_1}{t_1 + f_1} \tag{26}$$

3 Methods

This chapter give a description of the credit card data used and updates of the data made for the classification analysis in section 3.1. In section 3.2 we briefly introduce the Franke function used for the continuous data analysis. Section 3.3, 3.4 and 3.5 explain the implementation of the methods of logistic regression, neural network and linear regression analysis respectively. Lastly, section 3.6 give a description of our source code.

3.1 Credit card data

In this project we study some credit card data from a bank in Taiwan, containing payment data from 2005. The data provide explanatory information about the credit card customers, such as gender(X2), education(X3), maritial status(X4) and age(X5), as well as the amount of given credit(X1), history of past payments(X6-X11), amount of bill statements(X12-X17) and amount of previous payments (X18-X23). Lastly, the data include a real probability of 'default payment for the next month for each customer'(Y) indicated by *zero* for 'no default payment the next month', and *one* for 'default payment the next month'. The original data provide 30,000 data points [1].

We used the 23 explanatory variables (X1-X23) as our independent variables X, and considered the binary variables of default payments as our target variable Y. Further, we distinguished between continuous independent variables, such as age, and categorical independent variables, such as education. Customer data where there where no history of past payments and no amount of bill statements where deleted, as we did not want our model to train on this data. We considered any case where a categorical independent variable where accustomed a value not included its respective indicator variables as typo in the data. Consequently, these credit card data where removed. Through *OneHotEncoding* in ScikitLearn[3], each category of our categorical data where divided into a number of columns co-responding to the number of its indicator variables. Only one attribute in each of the new columns equal *one* (hot), while all the others equal *zero* (cold).

Thereafter, when exploring the balance of the target data we noted an imbalance of our classes with a ratio of 'no default payment' to 'default payment' of 79:21, as seen in Figure 3. As this imbalance in target data may cause for overfitting of the 'no default payment' group, we finalised our data by randomly picking out a number of credit card data of 'no default payment' target co-responding to the number of credit card data of 'default payment', giving a balanced ratio of 50:50. After this final adjustment we were left with 7134 data points. Lastly, the data where randomly divided into one group of training data(80%), and one group of validation data(20%).

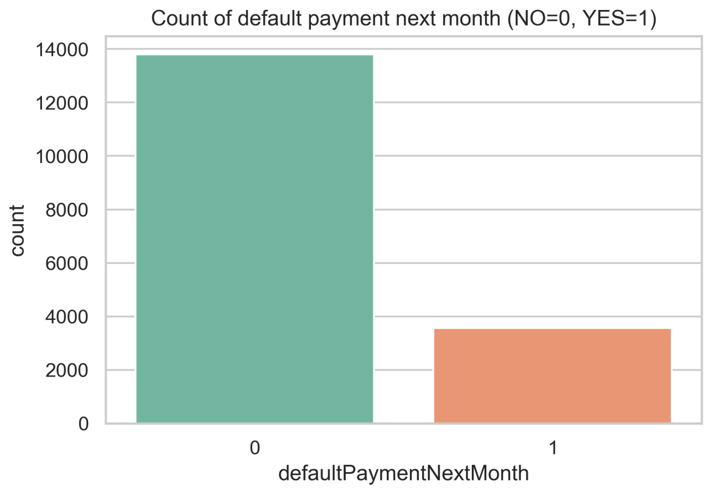


Figure 3: Counter plot of the balance of the target classes. The figure shows an imbalance between the number of 'no default payment': (NO=0), and 'default payment': (YES=1), with an 79:21 ratio.

3.2 Franke function

The Franke function[4] was used initializing two arrays of length 1000, containing random number between 0 and 1. We added some normally distributed noise to the function, of order 0.0-0.1. The design matrix corresponding to the Franke function is of polynomial degree 10.

3.3 Logistic regression

For the logistic regression code, we use a stochastic gradient descent in order to find the optimal parameters to estimate the target vector. The stochastic gradient descent first splits up the data into batches, then run the

[3]Description of Scikit-Learns inbuilt class for OneHotEncoding can be found at: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html

[4]For more information about the Franke function, please see Project one at https://github.com/HannahEkeberg/FYS-STK4155/tree/master/Project1

code iterating over a certain number of epochs, where the gradient is updated for each batch. The negative log-likelihood (equation (7)) was used as cost function and Sigmoid function (equation (5)) was used as activation function. We compare our result with Scikit-Learn's algorithm for logistic regression. We used LBFGS as solver in the Scikit-Learn prediction, as the stochastic average gradient descent solver (SAG) did not converge. However, this unabled us to see how the model varied with learning rate.

3.4 Neural Network

For the neural network, we made a class *dense* which takes an input array and an output array, make initial weights and biases, and return the weighted sum. In the *NeuralNetwork* class, we define the number of hidden layers and number of nodes in each layer. By iterating over the layers we feed forward, before backpropagating through equation (18)-(21).

For classification, we chose Sigmoid as our activation function for the output layer, since we wanted binary outputs. Activation functions of the hidden layers are defined in the code and can be determined specifically for each layer. For regression, Sigmoid is not suitable as activation function for the output layer, as we are dealing with continuous data. Instead we set ReLu as default for all the layers in the regression analysis. The negative log-likelihood function (equation (7)) was used for the classification problem, and MSE for regression (equation (2)).

We suspect an error within our neural network code, which we were unable to debug. Consequently, we do not trust the results obtained from the neural network code, and should thus not be treated as valid results. We set number of hidden layers to be 3, with nodes 50, 50, and 10. We compared to Scikit-Learn's algorithm for MLP for both classifier on credit card data and regressor on the Franke function.

3.5 Regression

Regression uses linear regression techniques OLS, Ridge and Lasso to estimate data, and how well it performs with MSE and R^2. Each method is compared to Scikit-Learn, however, there seem to be an error here as well, due to the data results looking too much alike.

3.6 Source code

The source code we have used is in the folder *ProgramMaterial* on GitHub.

- main.py - Main function where all code is run
- scores.py - accuracy score, confusion matrix, MSE and R^2
- cost_act_funcs.py - cost functions (negative log-likelihood and MSE), activation functions (Sigmoid, tanh, ReLu) for the neural network.
- creditcard_data.py - Sorting and balancing the credit card data
- Franke_function.py - Franke function and design matrix
- logistic_regression - Logistic regression code. Scikit-Learn's logistic regression algorithm is also included.
- neural_network - Neural network code. Scikit-Learn's algorihm is also included - both *regressor* and *classifier.*
- regression.py - a slightly modified program from project 1, added Scikit-Learn's algorithm for comparison.

4 Results

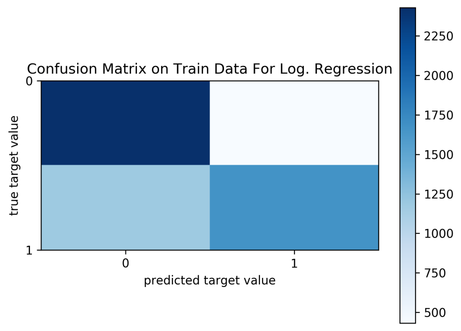
This chapter presents the results of our analysis. In section 4.1 you find the results of our classification analysis on the credit card data, while section 4.2 presents the results of our analysis on regression using data provided from the Franke Function.

4.1 Classification

Figure 5 shows the cost of the logistic regression as a function of epochs, with learning rate set to 0.01, batchsize set to 128. Table 1 shows the accuracy for batchsizes of $32, 64, 128$, and learning rates ranging from $10^{-5} - 10^{1}$. Figure 4 shows the confusion matrices for the best result: learning rate 0.01 and batchsize 128, for train data (4a) , validation data (4b) and for ScikitLearn (4c), with the corresponding values in the caption.

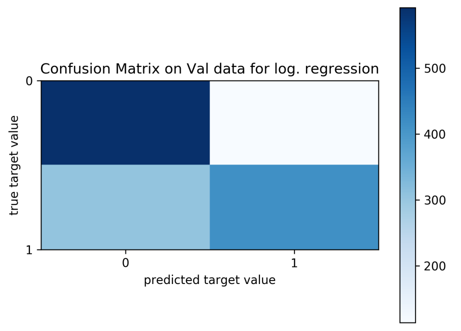
Figure 6 shows the cost of the Neural Network as a function of epochs for Sigmoid (6a), tanh (6b), and ReLu (6c) as activation functions, with learning rate 0.01 and batchsize 128. Table 2 shows the different accuracy scores for learning rate set to $10^{-5}, 10^{-3}, 10^{-1}$, for batchsize 128 and 256. Accuracy scores for ScikitLearn's Neural Network is also included for various learning rates. Figure 7 shows the confusion matrices for the best result: learning rate 0.001 and batchsize 128, with tanh as activation function for the hidden layer, with the corresponding value provided from ScikitLearn's Neural Network algorithm.

The precision and recall score for the methods which gave the best accuracy score for the validation data; ScikitLearn's Neural Network algorithm with tanh as activation function with learning rate 0.001, and logistic regression with learning rate 0.1 and batchsize 128 are given in table 3.



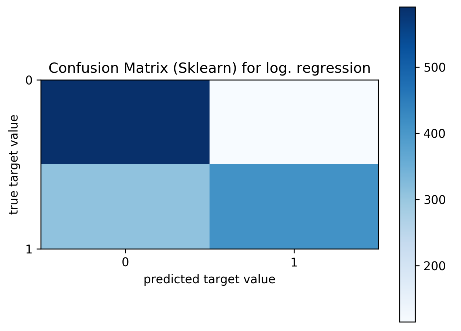
(a) Trainig confusion matrix with accuracy score 0.7174

$$\begin{bmatrix} 2428 & 433 \\ 1180 & 1666 \end{bmatrix} \tag{28}$$



(b) Validation confusion matrix with accuracy score 0.7057

$$\begin{bmatrix} 592 & 114 \\ 306 & 415 \end{bmatrix} \tag{30}$$



(c) scikit-learn confusion matrix:

$$\begin{bmatrix} 591 & 115 \\ 306 & 415 \end{bmatrix} \tag{32}$$

Figure 4: Confusion matrices with the best accuracy score for Logistic regression.

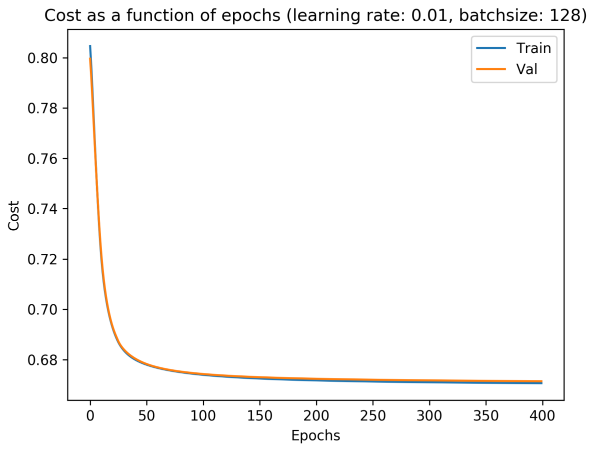


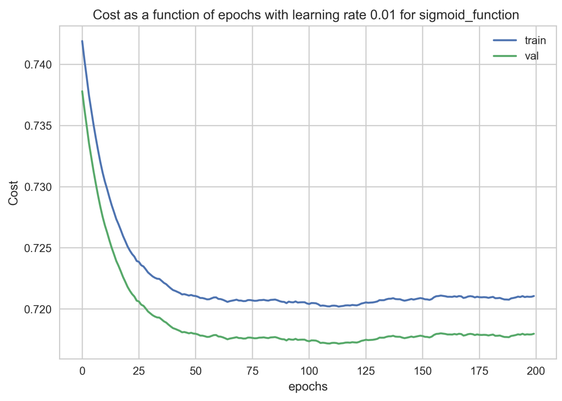
Figure 5: The cost as a function of epochs in logistic regression. Learning rate was 0.01, batchsize was 128, and number of epochs was set to 400.

Table 1: Accuracy scores for batchsizes 32, 64 and 128, with learning rates ranging from $10^{-5} - 10^{1}$. From Scikit-Learn we got an accuracy score of 0.7022. Batchsize of 128 and learning rate of 10^{-1} gives the highest accuracy score of 0.7174 for training data and 0.7057 for validation data.

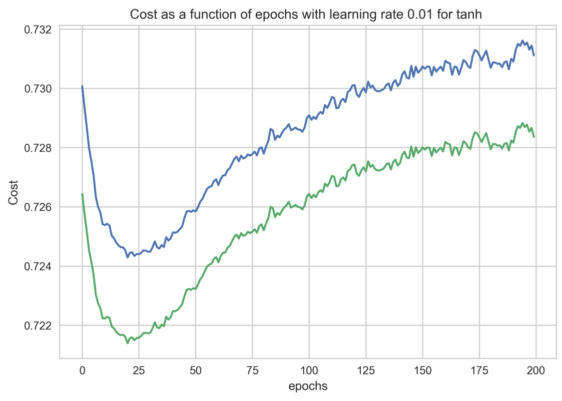
Batchsize	32		64		128	
	train	val	train	val	train	val
$\eta = 10^{-5}$	0.4778	0.4779	0.4773	0.4856	0.4770	0.4870
$\eta = 10^{-4}$	0.4972	0.4926	0.4747	0.4800	0.4763	0.4730
$\eta = 10^{-3}$	0.6990	0.6804	0.6639	0.6433	0.5924	0.5753
$\eta = 10^{-2}$	0.7123	0.7015	0.7093	0.7001	0.7065	0.6917
$\boldsymbol{\eta = 10^{-1}}$	0.7154	0.7029	0.7161	0.7057	**0.7174**	**0.7057**
$\eta = 10^{0}$	0.6897	0.6566	0.7021	0.6741	0.7114	0.6889
$\eta = 10^{1}$	0.6573	0.6412	0.6741	0.5999	0.6746	0.6692

Table 2: Accuracy scores for batchsizes 128 and 256, with learning rates $10^{-5}, 10^{-3}, 10^{-1}$, and for Sci-kit learn's own alogirithm for neural network with Sigmoid as solver, nodes: [50,50,10], and max iterations=1000.

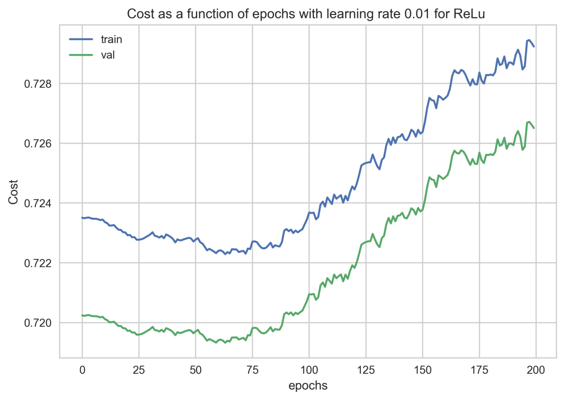
Learningrates	Batchsize 128		Batchsize 256		Sklearn	
	train	val	train	val	train	val
Sigmoid						
$\eta = 10^{-5}$	0.5000	0.4999	0.4992	0.5033	0.7249	0.7057
$\eta = 10^{-3}$	0.5002	0.4990	0.4999	0.5004	0.7249	0.7050
$\eta = 10^{-1}$	0.5003	0.4986	0.5003	0.4987	0.7249	0.7022
Tanh						
$\eta = 10^{-5}$	0.4987	0.5052	0.5013	0.4947	0.7261	0.7022
$\eta = 10^{-3}$	0.5013	0.4947	0.4987	0.5053	0.7265	0.7022
$\eta = 10^{-1}$	0.5002	0.4991	0.5013	0.4917	0.7256	0.7015
ReLu						
$\eta = 10^{-5}$	0.4996	0.5017	0.4987	0.5053	0.7249	0.7057
$\eta = 10^{-3}$	0.4993	0.5029	0.4992	0.5033	0.7249	0.7049
$\eta = 10^{-1}$	0.5002	0.4991	0.5003	0.4989	0.7249	0.7022



(a) Sigmoid



(b) tanh

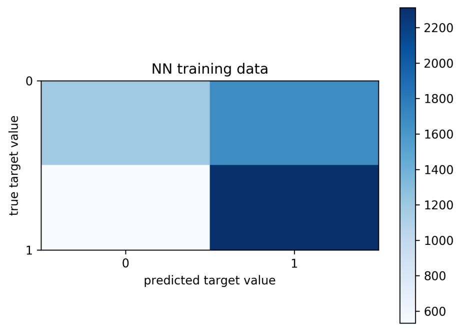


(c) ReLu

Figure 6: Cost functions using Sigmoid, tanh and ReLu as activation functions for the hidden layers.

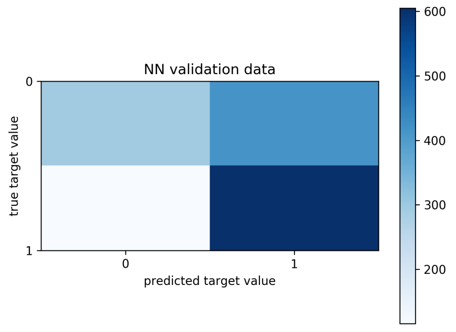
Table 3: Precision and recall of predicting zeroes and ones for Neural Network (using ScikitLearn for validation data for tanh as activation function, learning rate 10^{-3}) and Logistic Regression (using the validation data, learning rate 0.1, batchsize 128.

	Neural Network	Logistic Regression
Precison true 0	66.32%	65.92%
Precision true 1	76.15%	79.37%
Recall true 0	80.88%	83.85%
Recall true 1	59.78%	57.56%



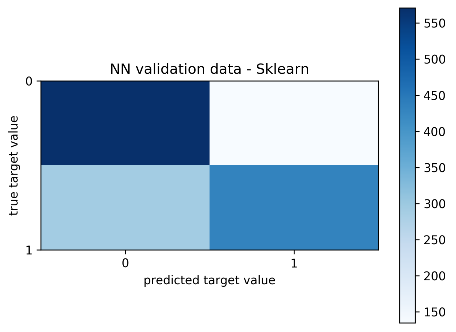
(a) Training confusion matrix with accuracy score 0.4995:

$$\begin{bmatrix} 1187 & 1674 \\ 533 & 2313 \end{bmatrix} \tag{34}$$



(b) Validation confusion matrix:

$$\begin{bmatrix} 292 & 414 \\ 116 & 605 \end{bmatrix} \tag{36}$$



(c) scikit-learn confusion matrix:

$$\begin{bmatrix} 571 & 135 \\ 290 & 431 \end{bmatrix} \tag{38}$$

Figure 7: Confusion matrices with tanh as activation function, and learning rate 0.001, for the Neural Network.

4.2 Regression

Table 4 shows the different Mean squared error as functions of hyperparameters, λ, η.

Table 4: Mean squared error for OLS, Ridge, Lasso (as functions of hyperparameter λ) and for Neural Network (as function of hyperparameter η). ScikitLearn is also included for comparison.

Hyperparameters	Analytical method		Sklearn	
	train	val	train	val
OLS				
$\lambda = 0$	0.0090	0.0108	0.0090	0.0108
Ridge				
$\lambda = 10^{-10}$	0.0090	0.0108	0.0090	0.0108
$\lambda = 10^{-5}$	0.0102	0.0108	0.0090	0.0108
$\lambda = 10^{-3}$	0.0115	0.0108	0.0090	0.0108
$\lambda = 10^{-1}$	0.0161	0.0108	0.0090	0.0108
Lasso				
$\lambda = 10^{-10}$	0.0153	0.0149	0.0121	0.0124
$\lambda = 10^{-5}$	0.0154	0.0149	0.0121	0.0124
$\lambda = 10^{-3}$	0.0238	0.0149	0.0121	0.0124
$\lambda = 10^{-1}$	0.1014	0.0149	0.0121	0.0124
NN				
$\eta = 10^{-5}$	0.1265	0.1168	0.0512	0.0497
$\eta = 10^{-3}$	0.1077	0.1013	0.0553	0.0496
$\eta = 10^{-1}$	0.0869	0.0995	0.0490	0.0560

5 Discussion

This section provides a critical run through and discussion on our main findings.

5.1 Costfunctions

In the classification problem, we want to have a cost function which produces high values when the classification is low, which is why we use the negative log-likelihood function as cost function in the Logistic Regression and Neural Network code. In regression models, we do not want a measure of how well the model classifies, but how well the model predicts new data, which is why we use the MSE for regression problems.

5.2 Learningrate, epochs and batch size

For the Logistic Regression code, we found 120 epochs to be a good number of epochs, due to the slight increase in cost of the validation data (figure (5)). The increase in cost-value for the validation data shows sign of overfitting to the training data. Table 4 represents the accuracy scores generated by the Logistic Regression code, for learning rates $\eta = 10^{-5} - 10^{1}$, and batchsizes 32, 64 and 128. Ideally, we would have made a learning rate schedule, instead of a constant learning rate. For the Neural Network code, we tried to plot the cost as a function of epochs for Sigmoid (6a), tanh (6b) and ReLu (6c). The results gave a clear indication that we have a bug in our Neural Network code, and does not give an indication of how many epochs we should run with. Therefore, we had chose an amount of runs and decided on 100. Table 2 represents the accuracy scores for the Neural Network code with learning rates $\eta = 10^{-5}, 10^{-3}, 10^{-1}$, and batchsizes 128 and 256, along with Scikit-Learn's neural network algorithm for comparison. Although the results obtained from the neural network code are poor, the results from Scikit-Learn neural network mathces the results obtained from Logistic Regression (table 1).
Due to the poor results obtained from the Neural Network code, we did not get to run the tests we wanted to. Ideally, we would test for various number of hidden layers and nodes. We tested for different learning rates but excluded the values from the report as they did not give any additional information.

5.3 Accuracy score

Learning rate of 0.1 and batchsize 128 gave the best accuracy score (0.7174 for training data and 0.705 for validation data). In comparison, the accuracy score from Scikit-Learn was 0.7022. This is a relatively low score, which can be due to the small dataset (in order to balance the data we decreased the number of datapoints from 30.000 - $\sim$7.000). The fact that Scikit-Learn's algorithm gives similar result indicates that the model is quite good.

Tanh as activation function and learning rate 0.001 gave the best result for Scikit-Learn's algorithm on Neural Network with an accuracy score of 0.7265 for training data and 0.7022 for the validation data.

Figure 4 shows the different confusion matrices for train data (4a), validation data (4b) and Scikit-Learn (4c) for learning rate 0.1 and batchsize 128. Figure 7 shows the different confusion matrices for train (7a), validation (7b) and Scikit-Learn (7c) for tanh as activation function, learning rate 0.001 and batchsize 128.

For both logistic regression and for Scikit-Learn's Neural Network code, we get a reasonable result. Both models does a good job in predicting true zeroes, and a slightly poorer job predicting true ones. The prediction of false ones is very low, while predicting false zeroes is lower than predicting true ones. For the Neural Network code, it is still clear that the network is predicting wrong.

Table 3 shows the precision and recall of true zeroes and true ones from the confusion matrices for the Neural network (Scikit-Learn validation data) and logistic regression (validation data). Neural network generally scores higher, but they perform similarly on all categories. Both models scores fairly poor on precision, meaning that it is quite often predicting wrong. Both models score slighly better for true ones. Both models scores relatively high on recalling true zeroe, while both models does a poor job recognizing the true ones.

5.4 Neural Network and regression

For regression, the cost function was changed from negative log-likelihood to MSE. Table 4 shows the Mean squared error for OLS, Ridge and Lasso as functions of hyperparameter λ, and regression using the Neural Network as function of learning rate η. From Scikit-Learn, calculating the MSE, we get identical results, which indicates that the prediction of y is not updated in the Scikit-Learn algorithm. Yet, comparing the values obtained from our own regression code for Ridge, Lasso and OLS, and Scikit-Learn's Neural Network code, it seems as the simple linear regression methods perform better than the Neural Network. Lowest value being 0.0553 for training data and 0.0496 for validation data with a learning rate of $\eta = 0.001$. For lower values of λ, the mean squared error for the validation data seems to be overfitted. Due to weird results in our code, we excluded the R^2 score from the results. Ideally, we would have included them.

6 Conclusion

In this project, we wanted to look if there is any difference in classification accuracy between Logistic regression and Neural Network, and if the estimated probability of default produced represent the real probability. Looking at the Logistic regression, we got an accuracy score of 0.7057 for the validation data, and comparing to ScikitLearn's Neural network algorithm, which gave 0.7022 for validation data, the two methods seem to perform similarly (table 3). Logistic regression with 120 epochs, batchsize 128 and learning rate $\eta = 0.1$ gave the best result, and for Neural Network, we conclude that it can not be evaluated critically, and that we should run for various layers and nodes once the program runs correctly. We also studied how Neural network perform in regression, where we used the Franke function. The results obtained from this (table 4) suggests that linear regression models OLS, Ridge and Lasso perform better than the Neural Network. From table 3, it seems as if logistic regression and neural network perform similar, but neural network slightly better. An accuracy score of 70 % does not represent the real probability, and this model should perhaps not be used to predict default payment.
The article done by Yeh & Lien (2017) suggests that the neural network gives a better result, but we do not want to conclude on behalf of our data analysis.

References

[1] Yeh, I-Cheng, Lien, Che-Hui (2007). *"The comparisons of data mining technicalities of default of credit card clients"*. https://doi.org/10.1016/j.eswa.2007.12.020

[2] Hjort-Jensen, Morten. *"Data analysis and Machine Learning: Logistic Regression"*. https://compphysics.github.io/MachineLearning/doc/pub/LogReg/html/LogReg.html (accessed 27.10.19).

[3] Akinkunle, Allen. *"Deriving Machine Learning Cost Functions uding Maxiumum Likelihood Estimation (MLE)"*. Published: 24.3.19. http://allenkunle.me/deriving-ml-cost-functions-part1 (accessed 27.10.19).

[4] Hjort-Jensen, Morten. *"Data Analysis and Machine Learning lectures: Optimization and Gradient Methods"*. https://compphysics.github.io/MachineLearning/doc/pub/Splines/html/Splines.html (accessed 27.10.19)

[5] Marsland, Stephen. "Machine Learning- *An Algorithmic Perspective*", second edition. Chapman & Hall/CRC.

[6] Codes in MATLAB for Training Artificial Neural Network using Particle Swarm Optimization - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/A-typical-artificial-neural-network-ANN_fig1_305325563. (accessed 24.10.19)

[7] Michael A. Nielsen, *"Neural Networks and Deep Learning"*, Determination Press, 2015. Available from: http://neuralnetworksanddeeplearning.com/index.html. (accessed 22.10.19)

[8] Sokolova, Marina and Japkowicz, Nathalie and Szpakowicz, Stan, *"Beyond Accuracy, F-Score and ROC: A Family of Discriminant Measures for Performance Evaluation"*, Springer Berlin Heidelberg, 2006.