# FYS-STK3155/4155 Project 2: Comparisons of machine learning methods on classification and regression analysis

Michael Bitney and Magnus Grøndalen

November 13, 2019

**Abstract**

In this project we have looked at two cases of data processing, which were classifying the credit card default data[1] and doing a regression fit of the Franke function. For classification we used Logistic Regression with a regular gradient descent, our own neural network with a regular gradient descent and Scikit-Learns neural network. We saw that the neural networks performed better than the Logistic Regression, with Scikit-Learns version performing the best with an $R^2$-score of 0.823. However, there are improvements to be made in order to increase the model by using a stochastic gradient descent, mini-batches and cross-validation. With the Franke function we used the Ordinary least squared method (OLS), our neural network code with a regular gradient descent and Scikit-Learns neural network with one and four layers. We saw that the Ordinary least squared performed the best by a good margin with an $R^2$-score of 0.978, except for the Scikit-Learn four-layer neural network which gave an impressive $R^2$-score of 0.9997. The OLS required less time to run with an easier algorithm, so we concluded that it's generally better, but if one has the accuracy of the model as the main priority, the multilayered neural network is better suited.

The material for this project can be found in our GitHub-repository at:
https://github.com/michaesb/machine_learn_2

## 1 Introduction

Machine learning is today a growing field, with numerous breakthroughs that makes it a desirable method for handling big data in numerous types of industry. For example classifying credit scores is a big task with huge amount of data sets, which was previously done manually by employees. A bank needs to take a risk every time they make a loan and too much bad loans is what happened in the economic crisis in 2008. This shows the importance of classifying the data properly and we look into how machine learning can tackle this problem.
Here we will look at credit card default data and will first look at it using Logistic Regression and then look at how accurate this model is. We then train a neural network with the same data and look for the difference in performance. We will look at how the neural network performs on the Franke function and compare this to Linear Regression. Our goal is see how well these perform on a given data set and to see which method is the best for the given case.
We have structured this paper in Theory, where we explain the theory behind our data sets and methods used, Results for where we explain our parameter values and display the scores for the different methods. We then discuss the results and compare methods in Discussion and then come with our Conclusion.

## 2 Theory

### Regression Methods

A regression method is a method using a statistical process to determine the output based on the relationship between one or multiple input parameters. Logistic Regression is used in cases where the outputs are different classes or categories, like yes/no and mouse/cat/dog. Linear Regression is used in cases where we try to fit a linear function based on the inputs.

[1]Credit card default data set in [6]

## Linear Regression

For more information on how Linear Regression works, we invite the reader to check out the previous report on Linear Regression Methods [5].

## Logistic Regression

Logistic Regression is a statistical model that lets you perform classification on data sets. Here we have design matrix $X$, being all the samples for different variables, with a coefficient $\beta$ and equation (2) that will be our model (2).

$$y_{\text{model}}(\beta) = \begin{cases} 1 & \textbf{if} \;\; \text{sigmoid}(X \cdot \beta) \geq 0.5 \\ 0 & \textbf{if} \;\; \text{sigmoid}(X \cdot \beta) < 0.5 \end{cases}$$

(1)

In order to find our beta here we need a cost function, which is equation (6). With this error function, we can use Newtons iterative method/gradient descent (3) to find the minimum of the cost function.

$$\beta_{i+1} = \beta_i - \frac{dC(\beta_i)}{d\beta} \Big/ \frac{d^2C(\beta_i)}{d\beta^2}. \tag{2}$$

If we derive the derivatives of the cost function we get:

$$\beta_{i+1} = \beta_i - (X^T W X)^{-1}(X^T(\hat{p} - \hat{y})).$$

Since $(X^T W X)^{-1}$ is a very heavy procedure for the computer, especially since it need be done for every iteration, we will replace it with a variable $\gamma$.

$$\beta_{i+1} = \beta_i - \gamma X^T(\hat{p} - \hat{y}) \tag{3}$$

Equation (3) demands that we find an optimal value for $\gamma$. If $\gamma$ is too low, we would need a very high number of iterations to find the minimum, but if $\gamma$ is too high, we might skip a minimum point and not get the optimal value for $\beta$.

## Neural Network

A neural network is a machine learning technique with inspiration from the brain. It uses so-called neurons, or nodes, that is a weighted sum of its input neurons. We have an input layer of neurons, an output layer of neurons, and then we have the hidden layers of neurons. A neural network will always have one input layer and one output layer, but might have an arbitrary number of hidden layers. Each of the neurons in one layer is connected via weighted edges to the neurons in the next layer. The goal of the neural network is to find weights that produce a wanted output. In figure 1 we can see the behavior of a neuron. This is further explained in the paragraph below. Lets take a look at the components of a neural network.

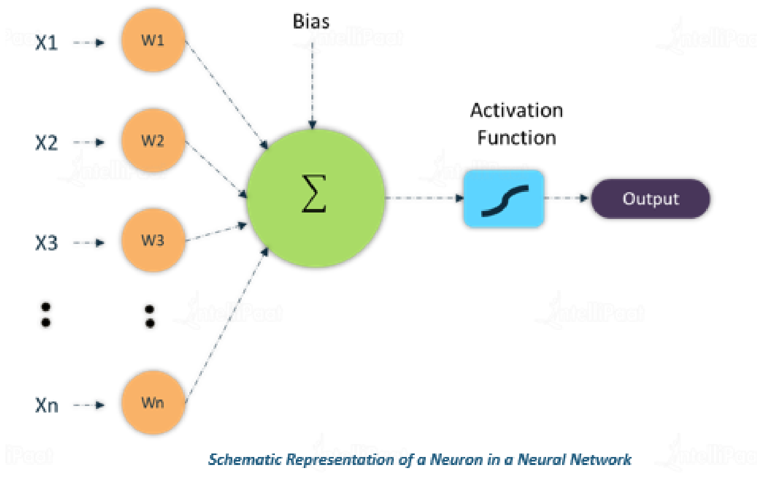


Figure 1: Figure showing the basics of a neuron in a neural network. Credit: https://intellipaat.com/

As stated above, a neural network has one input and one output layer, and an arbitrary number of hidden layers with an arbitrary number of neurons in each layer. Each neuron in one layer receives signals from all the neurons in the previous layer, and sends a signal to all the neurons in the next layer, much like a biological system. For a specific neuron $y$ we have that

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right), \tag{4}$$

where $n$ is the amount of neurons in the previous layer, $x_i$ is the value of the $i$-th neuron in the previous layer, $w_i$ is the weight of the "wire" between neuron $x_i$ and itself. At last, the $f()$ denotes the activation function of the neuron. This is the output of a neuron, and there exists multiple different activation functions for machine learning. Examples are the sigmoid function, the Rectified Linear Unit (ReLU) and the softmax function, and they are normally used in different scenarios. We will only look at the sigmoid function in this project. You can see the equation for the sigmoid function in (5). In figure 2 you can see the sigmoid function plotted with the step function. The step function is what we would like to approximate, but because we need the activation functions derivative, we choose the sigmoid.

$$f(t) = \frac{e^t}{1+e^t} = \frac{1}{1+e^{-t}}. \tag{5}$$

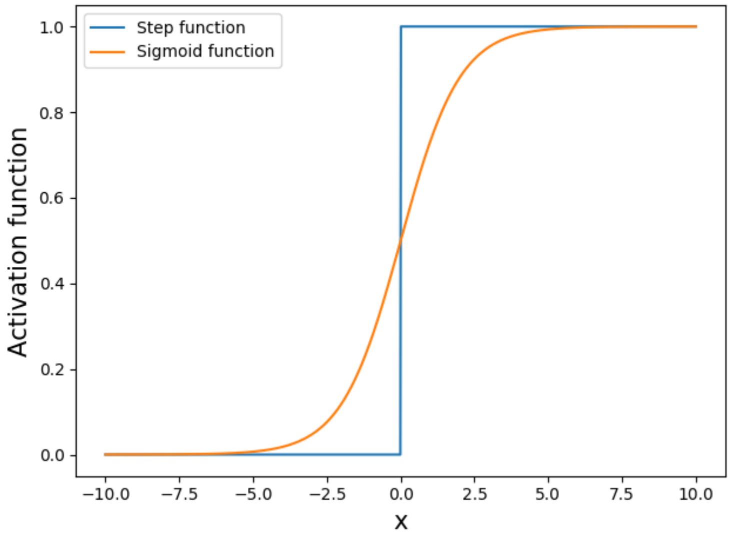


Figure 2: Figure showing the step function and the sigmoid function.

The method that takes the input variables and produces an output is called the feed forward method. It is responsible for adding the weighted sum forward through the network for each layer and neuron, through the activation functions, finally producing an output. In figure 1 you can see a visualization of this process for one neuron.

To validate whether our neural network is performing well, we need some kind of function that evaluates the fitness, or the error of the output from our neural network. This function is called the cost/loss function, and this is the function that we are trying to minimize. I.e. we want to optimize the weights and biases to minimize the cost function. There exists a variety of cost functions to choose from, but the simplest function is the Mean Squared Error (MSE) as you can see in equation (29). This cost function is probably the most popular and used cost function, but there is another cost function that is especially good in binary classification problems. That cost function is called the cross-entropy loss function, or the log likelihood, and the equation can be seen in (6).

$$C(y, \hat{y}) = -\sum_{i=1}^{n} \left(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)\right). \tag{6}$$

Where $y$ is the wanted output data, and $\hat{y}$ is the probability output predicted by our neural network.

When we have all these components as described above, i.e. a cost function, activation functions and feed forward method, we need to find a way to improve the performance of the neural network. This is done with something called backpropagation. It uses the errors at the output neurons, and "propagates" the error backwards through the network. To do this we need some kind of method that lets us change the weights to minimize the error from the cost function. One optimizer is the gradient descent, it lets you move the weights in the direction that decreases the cost function. First we look at the derivative of the cost function, using the cross-entropy loss in equation (6).

$$\frac{\partial C}{\partial \hat{y}} = -\sum_{i=1}^{n} \left(\frac{y_i}{\hat{y}_i} - \frac{1 - y_i}{1 - \hat{y}_i}\right). \tag{7}$$

To explain the backpropagation, we first make some abbreviations:

- $\hat{y}$: Output layer after activation
- $z_o$: Output layer before activation
- $w_o$: Weights for output layer
- $b_o$: Bias for output layer
- $a_h$: Hidden layer after activation
- $z_h$: Hidden layer before activation

- $w_h$: Weights for hidden layer
- $b_h$: Bias for hidden layer
- $X$: Input layer

We then propagate backwards using the chain rule. We take one step backwards until we find the weights and biases. First we look at the activated output layer $\hat{y}$, which is equated using the activation function, such as the sigmoid function.

$$\hat{y} = \text{sigmoid}(z_o) = f(z_o). \tag{8}$$

Since we now are seeing the $z_o$, we want to describe the cost function as a derivative of this quantity, and we do so by derivating $\hat{y}$ by $z_o$ and then using the chain rule.

$$\frac{\partial \hat{y}}{\partial z_o} = f(z_o) \cdot (1 - f(z_o)) = \hat{y} \cdot (1 - \hat{y}). \tag{9}$$

$$\frac{\partial C}{\partial z_o} = \frac{\partial C}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_o}. \tag{10}$$

Then we look deeper into $z_o$, by seeing how we equate it.

$$z_o = w_o \cdot a_h + b_o \tag{11}$$

We already now see the weights and biases for the output layer, and we can find the derivatives of the cost-function with respect to them. So we have $\partial C/\partial z_o$, so we need the $\partial z_o/\partial w_o$ which is just $a_h$, so we get

$$\frac{\partial C}{\partial w_o} = \frac{\partial C}{\partial z_o} \cdot a_h. \tag{12}$$

For the bias we just need the $\partial z_o/\partial b_o$, which is 1, and we get

$$\frac{\partial C}{\partial b_o} = \frac{\partial C}{\partial z_o} \cdot 1. \tag{13}$$

Now we have the derivatives for both the weights and the biases in the output layer, so we now need to go deeper to find the derivatives for weights and bias in the hidden layer. We then need to look into $a_h$, and what it's made of.

$$\frac{\partial z_o}{\partial a_h} = w_o. \tag{14}$$

We then again use the chain rule to find the impact $a_h$ has on the cost function, and do the same as we did in the previous steps for $\hat{y}$.

$$\frac{\partial C}{\partial a_h} = \frac{\partial C}{\partial z_o} \cdot w_o \tag{15}$$

$$a_h = f(z_h) \tag{16}$$

$$\frac{\partial a_h}{\partial z_h} = f'(z_h) = a_h \cdot (1 - a_h) \tag{17}$$

Using the chain rule once again to find the impact of $z_h$ on the cost-function.

$$\frac{\partial C}{\partial z_h} = \frac{\partial C}{\partial a_h} \cdot \frac{\partial a_h}{\partial z_h} \tag{18}$$

$$z_h = w_h \cdot X + b_h \tag{19}$$

Finally, we found the weights and biases for the hidden layer, and we just repeat what we did at the output layer for the weights and bias.

$$\frac{\partial z_h}{\partial w_h} = X \tag{20}$$

$$\frac{\partial C}{\partial w_h} = \frac{\partial C}{\partial z_h} \cdot X \tag{21}$$

$$\frac{\partial C}{\partial b_o} = \frac{\partial C}{\partial z_o} \cdot 1 \tag{22}$$

Now we have found all the derivatives we need to update the weights and biases for the neural network. We also need to decide how much we want to change the weights and biases based on its derivatives, and we do that by using a learning rate $\gamma$. The final equations for the optimizing is

$$w_o = w_o - \gamma \cdot \frac{\partial C}{\partial w_o}, \tag{23}$$

$$b_o = b_o - \gamma \cdot \frac{\partial C}{\partial b_o}, \tag{24}$$

$$w_h = w_h - \gamma \cdot \frac{\partial C}{\partial w_h}, \tag{25}$$

$$b_h = b_h - \gamma \cdot \frac{\partial C}{\partial b_h}. \tag{26}$$

$$\tag{27}$$

Neural networks are often used for classification problems, but can also be used for regression analysis. Depending on whether you're solving classification or regression problems, there are some differences on how to set up your neural network. Lets take a look at the differences.

#### Classification vs. Regression

Using a neural network for classification problems means that you want the neural network to classify inputs as the correct category. This means for example classifying whether an image is showing a dog, a cat or a mouse. If the output from your data says that dog is equal to 0, cat is equal to 1 and the mouse is equal to 2, it is often favored to create three output neurons that is either 0 or 1, instead of one output neuron between 0 and 2.

Neural network for regression means you want to fit a function, or at least something linear or sort of continuous with your neural network. This means having just one output neuron. For example you could estimate real-estate housing prices with different attributes like size, location, number of bedrooms and so forth.

As already stated, the choice of cost function often vary between classification and regression problems. An all-round good cost-function is the Mean Squared Error in equation (29), and is used in both classification and regression. A popular cost-function for classification is the cross-entropy cost-function as you can see in equation (6). The backpropagation also needs to adapt to the cost-function of choice. To help decide whether the neural network is performing well, we have something called a score. For classification the score is calculated using the accuracy-score function

$$\text{Accuracy} = \frac{\sum_{i=0}^{n-1} I(\hat{y}_i = y_i)}{n}. \tag{28}$$

Where $\hat{y}$ is our predicted output, $y$ is our targets, and I is the indicator function, 1 if $\hat{y}_i = y_i$ and 0 otherwise. This score-function doesn't apply to regression analysis, where we rather use something called the $\mathrm{R}^2$ score function, as described in the equation (30). More on this topic in the section below.

## Testing and modeling error for Franke

#### MSE and $\mathrm{R}^2$-score

In order to test our regression models, we will use multiple methods to determine how well our models fit the data. Here we use an expression of the Mean Squared Error. This tells us the average of all the errors squared, which gives us an idea on how good a model is.

$$MSE(\hat{y}, \tilde{y}) = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2 = \mathbb{E}[(y - \tilde{y})^2] \tag{29}$$

Here the $\tilde{y}_i$ is the model, $y_i$ is the data, and $\mathbb{E}[]$ is the expectation value of an expression.

We will use an $\mathrm{R}^2$-score function, which tells us how well the fitted line is to the data. The best score is 1 and the worst score is $-\infty$. An $\mathrm{R}^2$ score of 0 is a straight line at the mean, which is considered noise. See equation (30) for the formula.

$$R^2(\hat{y}, \tilde{y}) = 1 - \frac{\sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2}{\sum_{i=0}^{n-1} (y_i - \bar{y})^2} \tag{30}$$

### Franke function

The Franke function is a common test for regression analysis with two inputs.

$$f(x,y) = \frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) \tag{31}$$

$$+\frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) \tag{32}$$

$$+\frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) \tag{33}$$

$$-\frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right). \tag{34}$$

This function is a common test for regression analysis and you can see the shape function in plot 3

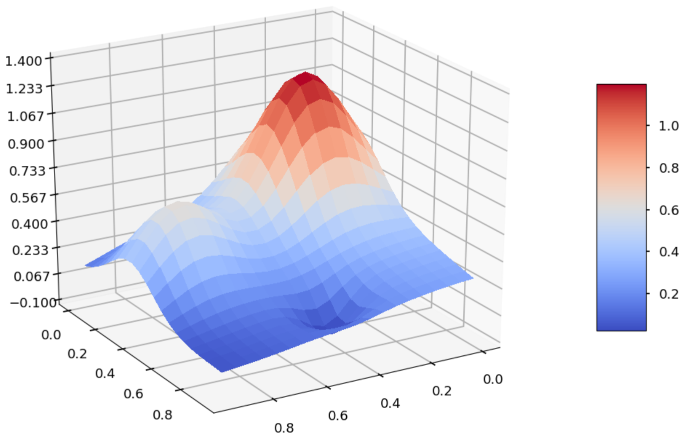


Figure 3: This plot shows the Franke function in a 3D plot

### Credit Card data

Credit card data is used in economics to determine if an individual is suited to receive a loan from a bank. A loan is at a risk for the bank, although the bank will demand more than the loaned amount, because the borrower might not be able to pay back the loaned amount, because of numerous reasons. A failure to pay back the loan is called a default. Credit Card data gives us data so we can look for certain indicators of whether the person will default on the loan. With machine learning techniques we can use this data to predict who will default on the loan.

## 3 Results

### Data processing: Credit Card data

For our logistic Regression and Neural net we have used the Credit Card data. This is credit card information from individuals from Taiwan and it shows information about an individual and whether they pay their bill or default. This data was retrieved from the site in [6] and is a common example for Machine Learning Algorithms. We have a description of the data given on the site and in our repository. We saw however that in the data there were certain classifications that was not mentioned in the description. We decided to remove these individuals from the the data set and focus on the ones we can identify. We then have roughly 23 000 individual samples to process. The data contain attributes like gender, payment history, education and more. These attributes is the information we feed to our method and the output will be either 1 for default or 0 for not default.

### Logistic Regression

We constructed our Logistic Regression function using Gradient descent, but we needed to adjust the parameters learning rate and number of iterations to find a good model for the credit card data. We plotted multiple learning rates to try to find the best one in graph 4.

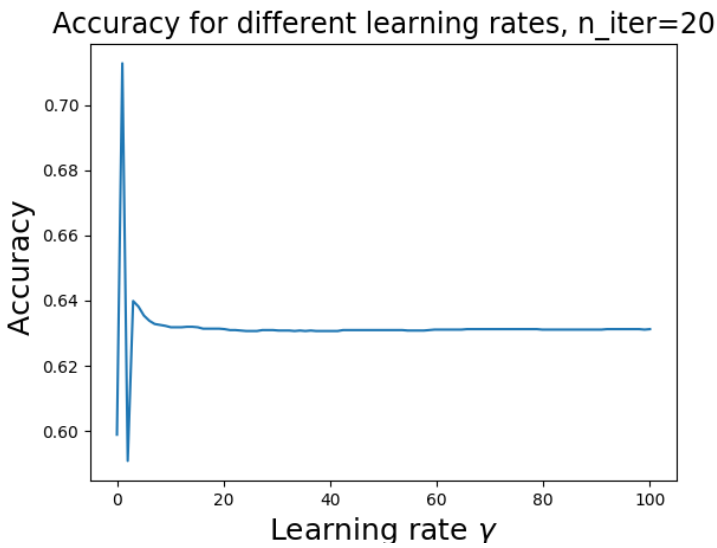


Figure 4: Here we have plotted the accuracy vs. the learning rate with 20 iterations in order to plot for many learning rates. Note the spike between 0 and 5.

We saw a spike between 0 and 5 and decided to zoom on this area in graph 5.

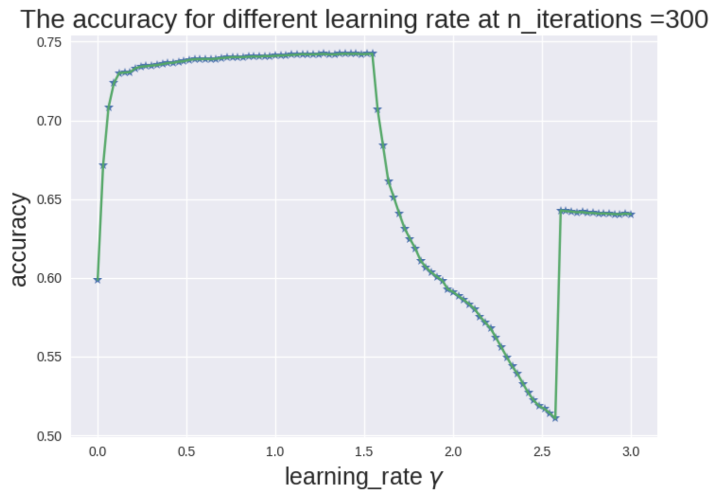


Figure 5: Here we see the a zoomed up version of graph 4 but with a higher iterations at 200.

To ensure that the learning rate was optimal for different iterations, we did a heatmap for different iterations an different learning rates.

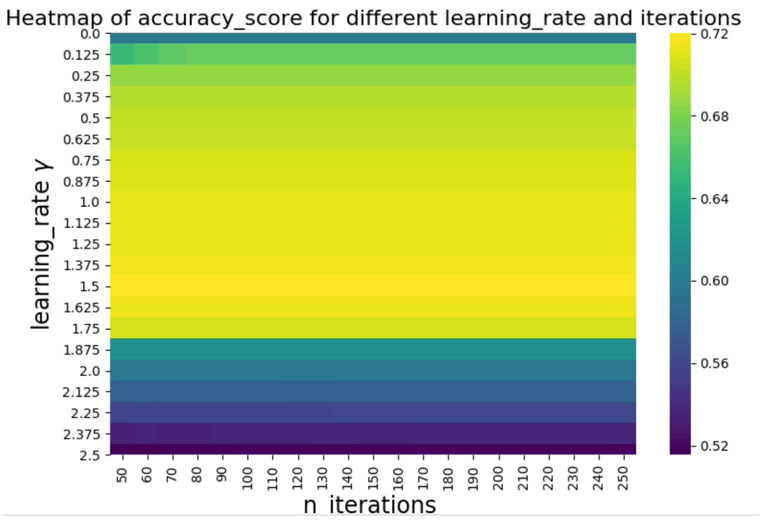


Figure 6: This is a heatmap of the accuracy where the x-axis is the learning rate and y-axis is number of iterations.

We saw that the optimal value for the learning rate is in the interval [1.3,1.5] and used this learning rate in graph 7.

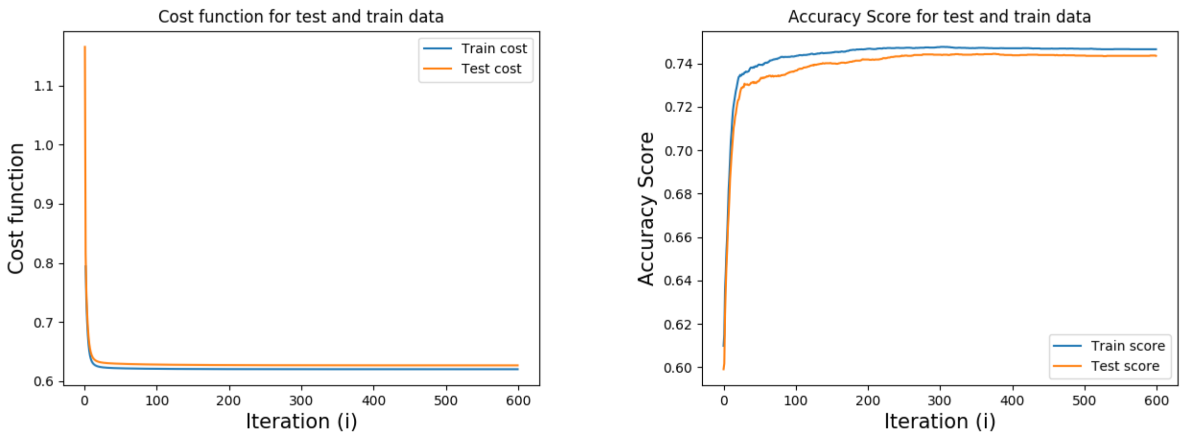


Figure 7: Figure to the left shows the cost/loss through the iterations of the gradient descent, and the figure to the right shows the accuracy scores throughout the gradient descent

## Neural Network for Classification

We made our neural network with only one hidden layer. We used the gradient descent method to find the minimum point using the cross-entropy cost function. We found it difficult to find the optimal parameter values, since they all impact it in a different way and it didn't appeared to have any obvious pattern. After trial and error we found the best learning rate to be at 0.0001 and number of neurons/features at 80.

In figure 8 we used a learning rate that was way too big, but in figure 9, we see the loss is gradually getting better, as well as the accuracy score.

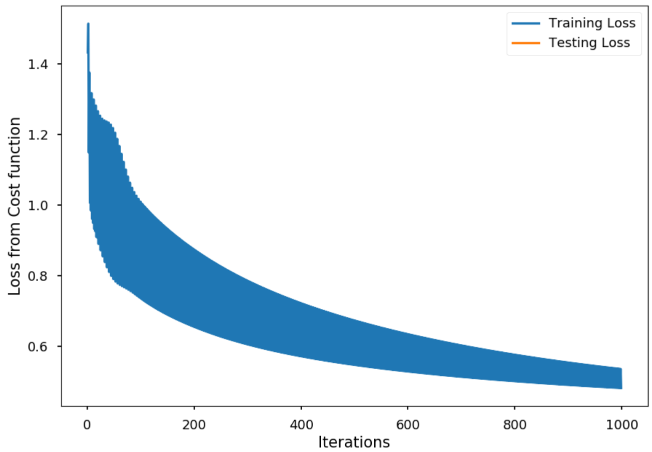


Figure 8: In this figure we see the Cost function through the gradient descent with a non-optimal learning rate.

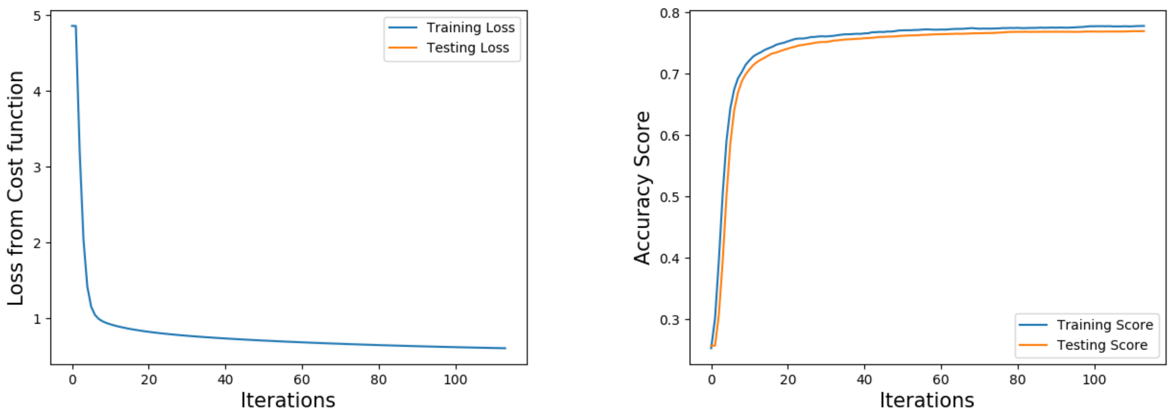


Figure 9: This is the performance for the Neural Network on the credit card data. Figure to the left shows the cost/loss through the iterations of the gradient descent, and the figure to the right shows the accuracy scores throughout the gradient descent. We used a learning rate of 0.00001 in these figures.

## Neural Network for Regression

In the regression analysis using our neural network, we only used one hidden layer, and varied the number of neurons, the $L2$ penalty and the learning rate. We used the Mean Squared Error as our cost-function, and the sigmoid function again as the activation functions for all the layers. We used the regular gradient descent also in this neural network, but you should consider using a stochastic gradient descent or mini-batches. In figure 10 we used a high learning rate, and in figure 11, we decreased the learning rate.

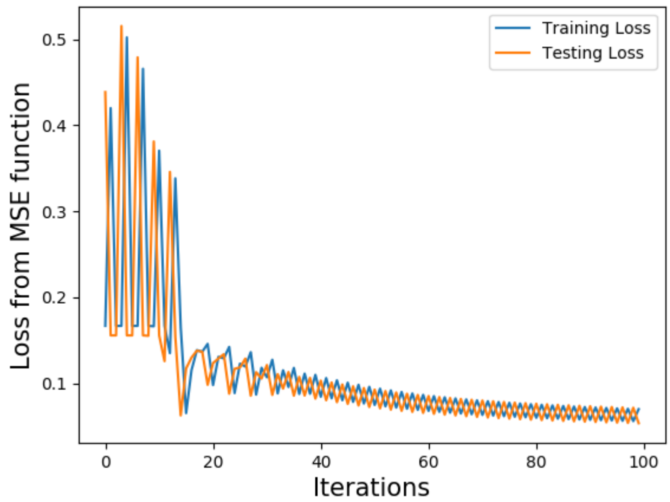


Figure 10: In this figure we see the learning process for our neural network for regression. Here we are using a non-optimal learning rate.

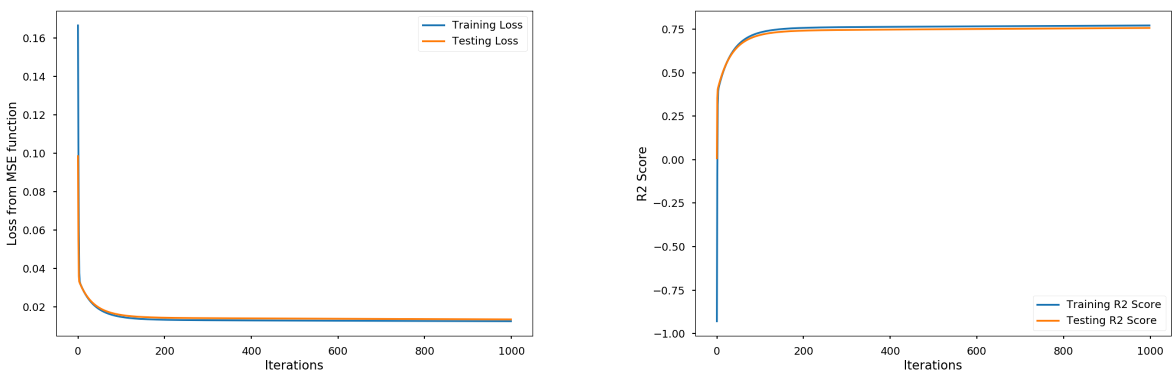


Figure 11: In this figure we can see the learning process of using gradient descent in our neural network, with the cost-function being on the left, and the accuracy score on the right.

## Comparison

In order to compare our neural network, we utilized the Scikit-Learn package to create a neural network. The difference between the package we used from Scikit-Learn and our own neural network code, is that Scikit-Learns neural net is not limited to one layer and can use an adaptive learning rate. We will compare the Scikit-Learn package with one layer and multiple layers.

Classification

| Method | Accuracy |
|---|---|
| Logistic Regression | 0.743 |
| Logistic Regression Scikit-Learn | 0.817 |
| Neural Network | 0.819 |
| Neural Network Scikit-Learn (single layer) | 0.823 |
| Neural Network Scikit-Learn (4 layer) | 0.822 |

Table 1: Here's the accuracy and score for the Credit card data for multiple methods. Note that Scikit-Learn has only an initial learning rate and will be adjusted as the program progress.

To decide how well our classification methods behaved, and for comparing the different methods, we decided to look into the cumulative gain charts. We made use of Scikit-Plots method `plot_cumulative_gain` to find these figures in 12. By finding the area between the baseline graph and our "Default"-line, relative to the area between the baseline graph and the "best" graph, we can compare the different classification techniques.

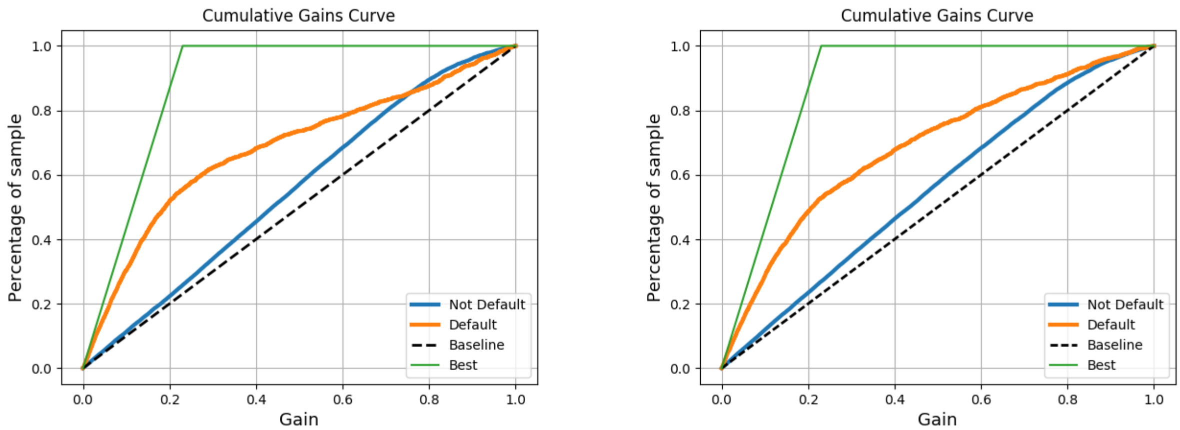


Figure 12: These figures show us the cumulative gain graph of our Logistic Regression and our Neural Network respectively. The area for the Logistic Regression was 0.478 and the area for our Neural Network was 0.559

Here we use OLS (Ordinary Least squares method) from the Linear Regression report [5] as it performed the best on the Franke function.

Regression

| Method | $R^2$-Score | MSE |
|---|---|---|
| OLS | 0.978 | 1.98e-3 |
| Neural Network | 0.821 | 0.040 |
| Neural Network Scikit-Learn (Single layer) | 0.974 | 0.022 |
| Neural Network Scikit-Learn (four layers) | 0.9997 | 1.25e-3 |

Table 2: Here the mean squared error and the accuracy for different methods on the Franke Function. They were each given a mesh grid of 50 x 50 points and all have been spent time to find the best score by adjusting the parameters. Note that the Neural network code that we designed has only one hidden layer and Scikit-Learn has the option to use one or multiple layers. Note that the four layer Neural Network needed a lot of time to run. A seed value was used to get the same values for all methods

# 4 Discussion

## Logistic Regression

We can see that our Logistic Regression works based on the plots in figure 7. Here we see that as the program progresses the cost function decreases and the accuracy increases. This indicates that the program is adjusting the $\beta$ -values to minimize the cost function, which is the desired behavior. We can also see that the test and train values are very close to each other in figure 7. We can see on the cost function plotted, that train is below test, which is due to the fact that the model is being trained using the training set, and the test data is unseen to the model. We can also see that the accuracy is better for train than for test as well, which is expected. We might get some overfitting, since train has a higher accuracy, but it seems to be very low in this case, since test and train are very close to each other. It could worth looking into doing cross-validation on the data, to avoid overfitting in the data, especially if one intends to run for more iterations.
We used our learning rate from the heatmap in figure 6 which gives us a model of an accuracy of roughly 0.75. It is possible that we have missed a better value for the learning rate somewhere, because we did a very low-resolution sweep of the parameters. So it's likely that this result can be improved upon with a better a machine with better computational power. One thing to consider is that our iterative method might be stuck in a local minima. In order to avoid this we can use mini-batches and a stochastic gradient to avoid this local minima trap. It's difficult if not impossible to see with a regular gradient if this happens, so this shows more potential to use a stochastic gradient descent instead to find a better model.

For deciding the error of our Logistic Regression classification, we used the cross-entropy as our cost-function. We used the cross-entropy without any regularization terms, such as $L_1$ and $L_2$ penalties that is used in Ridge and Lasso regression. We would suggest adding regularization terms to the cost-function, as this might improve the model.

## Neural network for classification

In graph 9 the cost function decreases and accuracy increases, which tells us that the method is functional. We also plotted the cost function for an non-optimal learning rate in graph 8. This to show how it oscillates over a minima point and it's likely due to the learning rate being to large.This doesn't happen when using a more optimal learning rate in graph 9, which again tells us we have a good value for the learning rate.

Our artificial neural network for classification turned out to get an accuracy score of about 0.8, which is pretty good considering we only had one hidden layer. As stated, we went for one hidden layer, and used the sigmoid activation function in both the hidden layer and the output layer. Because of the vanishing gradients problem by using the sigmoid function, one could try using another activation function, such as the softmax or the ReLU. A reason for using the sigmoid function in our case, is that we want an output between 0 and 1, e.g. a probability of our output.

We used the standard gradient descent in this neural network, and it is clear this has its limits. We would suggest using a stochastic gradient descent or mini-batches for a more stable performance. To validate whether our errors were accurate, we probably would have benefited from using some form of Cross-Validation such as the Kfold.

When tuning the variables, such as the amount of neurons in the hidden layer, regularization term $L_2$ and the learning rate, we tried a lot of different values until we were happy with the result. To make it easier to understand what the variables could be tuned to, we could have performed a grid search of some selected area for the learning rate and the regularization parameter.

## Neural network for regression

We can see in graphs 11 that here as well the cost function goes down and accuracy goes up when we run it. This shows the neural net is working and we can compare it to graph 10, where we used a non-optimal learning rate. We can see it oscillates over a minima, due to a too high learning rate.

The neural network we made for performing regression analysis did not perform quite as well as our neural network for classification. The $R^2$-score of our method had a maximum of 0.821. Compared to our OLS from project 1 [5], which had an $R^2$-score of 0.978, and Scikit-Learns neural network with 0.9997, we see that our neural network didn't stand a chance against other methods. In this method, we also used only one hidden layer, which might be a significant reason for the bad performance of this network.

As the other classification methods above, we used the gradient descent and would have benefited from using a more stable and advanced optimizer such as the SGD or mini-batch optimizer. We still used the sigmoid activation function, so we scaled the input from the Franke function to be within 0 to 1.

The tuning of parameters was also done using trial and error, so this could also benefit from doing a grid search. We however don't have the computational strength to perform a good sweep of the values, so we went for intuition instead.

### Comparison of methods

In table 1 we have displayed the accuracy score for each of the methods. We can see that our logistic regression performed worst out of all the methods. Scikit-Learns Logistic Regression performed much better. We couldn't specify the learning rate in the Scikit-Learn package, which might infer it might have an adaptive learning method or some more advanced method underneath the hood. Unfortunately it is a bit of a black box, but it gave us a better result than our own package.
Our Neural Network however, performed much better than the our logistic regression. This was using with a single layer and shows promise, despite there being numerous possible improvements that could be made. The Scikit-Learn package performed better than our own code by a small amount. Here we could see that adding more layers to the network did not improve the performance. However this was not thoroughly looked into, because of the sheer number of permutations. We then might have missed a good parameter value here and doing a systematic grid search could gives us better parameter values.

We can see in the cumulative gain graph 12 that Logistic regression has a higher increase in the beginning. This could indicate that Logistic Regression is better at identifying the default cases with higher probability. However, neural network seemed to perform better at the non-default cases with lower probabilities and was all around better for all cases. However this difference is hard to see in the graph, but is an interesting effect that is worth looking into. We calculated the area under the curve as well, with neural network being higher than the Logistic Regressor. This fits well with what the accuracy score implies.

In the previous report [5], we looked at the regression method OLS and you can see the comparison in table 2. Here we can see that OLS performs better than the neural networks, than our Neural Network and Scikit-Learn Network (single layer). OLS is a simple algorithm, yet managed to get a better value. However the four layer Neural Network got a score of 0.9997. This method had a long execution time, but turned out to be a very accurate model. Unfortunately we could not adjust the parameters due to computational constraints, but it shows that with the Franke function neural Networks can be very accurate. OLS might be preferable because of simplicity and time usage, but if accuracy is the main concern, the four layer Neural network would be better suited.

## 5 Conclusion

We created a Logistic Regressor and two neural networks; one for classifying data and for doing regression analysis, and then compared it to Scikit-Learns package. For the credit card data we found that the neural network performed better than the Logistic Regression. The Scikit-Learn version of the Neural Network performed the best here with an accuracy score of 0.823.

On the Franke function, Ordinary least squared performed better than the regular neural network, except a four layer Scikit-Learn Neural Network which gave us an accuracy score of 0.9997, although this took a large amount of time. We concluded that OLS is better than Neural network, despite having a slightly lower score due computational strength needed and simplicity. If accuracy is of high importance then a multilayered neural network might be better suited.

We recommend trying a stochastic gradient descent with mini-batches on the neural network and using cross-validation to check for overfitting on the iterative method. We also recommend looking into our neural network with high-performance computers and performing grid search in order to find better parameters, to improve the models further.

## References


[1] Lecture Notes made by Morten Hjorth-Jensen in class Fys-Stk4155/3155 at uio
`https://compphysics.github.io/MachineLearning/doc/web/course.html`

[2] Hastie Tibshiran Friedman; The Elements of Statistical Learning; Second edition; Springer 2009

[3] Géron, Aurélien; Hands-On Machine-Learning with Scikit-Learn & TensorFlow; First edition; O'Reilly 2017

[4] Yeh, Lien; The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients;

[5] Grøndalen Bitney; Project Regression for the subject FYS-STK3155/4155
https://github.uio.no/michaesb/ml_project1_mms/blob/master/report_fysstk_magnubgr_michaesb.pdf

[6] I-Cheng Yeh; default of credit card clients Data Set; UCI Machine Learning Repository
https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients

## Acknowledgements

Thanks to Morten Hjorth-Jensen and student teachers in FYS-STK-4155/3155 for help during this project.