FYS-STK4155 – Project 2 on Machine Learning

Classification and Regression

Viggo Wetteland

November 8, 2020

Abstract

In this project, we have studied in detail both classification and regression problems by developing our own feed-forward neural network code. We have compared the neural network's performance against OLS and Ridge regression algorithms on a regression problem, and against a logistic regression algorithm on a classification problem. We also studied Stochastic Gradient Descent and various activation functions like RELU, Sigmoid, and Softmax in our codes. For the regression problem, we found that the simplest model, OLS, performed best because the other models overcomplicated the problem. For the classification problem, we found that the FFNN code had the highest potential and performed better, but required a bit more work and fine-tuning of the parameters than the logistic regression algorithm.

1 Introduction

The goal of this project is to study classification and regression problems and compare various machine learning algorithms on the same datasets. We will develop a logistic regression code and a feed-forward neural network (FFNN) code in addition to a modification of our old OLS and Ridge regression algorithms from project 1 [9]. The Franke function data will represent our regression problem, and an MNIST set of handwritten numbers from 0 to 9 will be our classification problem. Stochastic Gradient Descent (SGD) will be added to the OLS and Ridge methods, and to the logistic regression code. The modified OLS and Ridge regression codes will be tested on the Franke function data and will be compared with the FFNN code on the same dataset. For the classification problem, we will compare the FFNN code and the logistic regression code. The cost functions used to quantify the error between predicted values and expected values will mainly be MSE for the Franke data and accuracy score for the MNIST data. While the linear regression on the simulated terrain dataset will be handled much like how it was done in project 1, the new methods introduced will handle the classification problem of the MNIST dataset. Instead of trying to fit the model to a terrain, we will here train our models to predict these handwritten numbers correctly with as high a success rate as possible.

To compare the various models, we will study and optimize hyper-parameters like the learning rate, the number of mini-batches in the SGD, epochs, and L2 regularization. Different activation functions like the sigmoid, RELU, and leaky RELU function will also be compared in the FFNN model. After we attempt to

optimize each model for the regression and classification problems we will give a critical evaluation of the various algorithms of their pros and cons in the regression case and in the classification case, and declare which algorithm works best on these two types of data.

This report is structured with an abstract, introduction, method section, results and discussion section, conclusion, appendix and references.

2 Method

2.1 The Stochastic Gradient Descent implementation

We will now start off by updating our standard ordinary least squares (OLS) and Ridge regression codes from project 1. The matrix inversion algorithm will be replaced by our own SGD mini-batch code. The objective of Gradient Descent is to descend a slope to reach the lowest point on that surface. The Stochastic part introduces the randomness of the algorithm by only evaluating one random datapoint instead of all data points at each step. It is obvious that each iteration will be less accurate when only evaluating one data point at a time, so we will instead evaluate a small group of random data points, merging the slow and accurate Gradient Descent method with the much faster Stochastic method [1].

We will study the learning rate, epochs, and mini-batch parameters to see how they affect the model's precision. The learning rate is a flexible parameter that heavily influences the convergence of the algorithm. Larger learning rates make the algorithm take huge steps down the slop and it might jump across the minimum point thereby missing it. A learning rate that is too small may never converge or may get stuck in a local minimum (a suboptimal solution). We will make use of a learning schedule function in the first part of the project that goes like $\frac{t_0}{t}$ where t increases with each iteration. The number of epochs is a hyperparameter of gradient descent that controls the number of complete passes through the training dataset.

Mathematically, the Gradient Descent variations discussed here can be expressed in three ways. Firstly with standard Gradient Descent, secondly with Stochastic Gradient Descent and then thirdly with mini-batch Gradient Descent.

$$w = w - \eta \nabla_w Q(w) \tag{1}$$

$$w = w - \eta \nabla_w Q\left(x^{\{i\}}, y^{\{i\}}; w\right) \tag{2}$$

$$w = w - \eta \nabla_w Q\left(x^{\{i:i+b\}}, y^{\{i:i+b\}}; w\right) \tag{3}$$

where the parameter w that minimizes Q(w) is to be estimated. η is the learning rate (step size). The expression $i: i + b$ refers to the mini-batch for each iteration where b is the batch size. Equation 3 corresponds to mini-batch Gradient Descent and is the one we will implement in the OLS and Ridge regression algorithms.

2.2 The Feed Forward Neural Network code on Franke function data

Our next objective is to compare the OLS and ridge regression methods with our FFNN code. A feedforward neural network is a biologically inspired classification algorithm. It consists of a (possibly large) number of simple neuron-like processing nodes, organized in layers. Every node in a layer is connected with all the nodes in the previous layer. These connections are weighted differently. Data enters at the inputs and passes through the network, layer by layer until it arrives at the outputs. When there is no feedback between layers it is called a "feedforward" neural network [2]. In figure 1 below, we see an example of a 2-layered network.

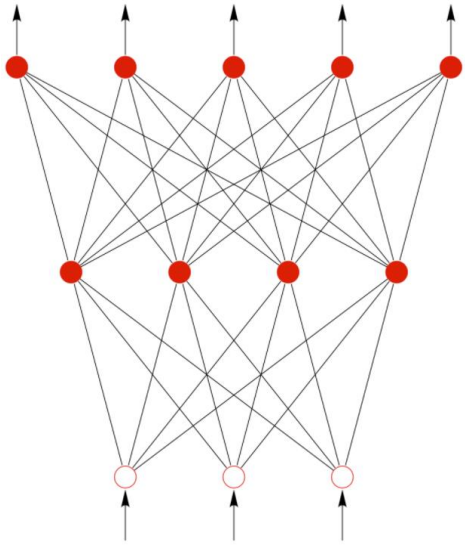

Figure 1: Illustration of a feedforward neural network. From top to bottom: an output layer with 5 units, a hidden layer with 4 units, and then 3 inputs.

Two important new learnable parameters in a machine learning model like this are weights and biases (commonly referred to as w and b). When the inputs and transmitted between neurons, the weights are applied to the inputs along with the bias.

$$Y = \sum(weight * input) + bias \tag{4}$$

Weights control the signal (or the strength of the connection) between two neurons. In other words, a weight decides how much influence the input will have on the output. Biases, which are constant, are an additional input into the next layer. The bias unit guarantees that even when all the inputs are zeros there will still be an activation in the neuron [3]. An example of how to do this numerically is like this:

```
def feed_forward_train(X):
    z_h = np.matmul(X, hidden_weights) + hidden_bias
    a_h = sigmoid(z_h)
    z_o = np.matmul(a_h, output_weights) + output_bias
    return a_h, z_o
```

where we have included the sigmoid function $S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$ which returns a value between 0 and 1 based on the input values. If the input value is closer to 0 than 1, it will return 0 and vice versa. The reason for using this function is for example for when we want an output of either 0 or 1, but not 0.25 or 0.75. So in practice, the weights that are of lesser importance will get the value 0 + bias and the more important weights will get the value 1 + bias.

In our FFNN code, we will also add backpropagation after we've computed the output error. This is basically by computing backward into the layers again. By computing the errors in each layer backward we update the weights and biases using gradient descent. Below is an example of how to do this numerically:

```
def backpropagation(X, Y):
    a_h, z_o = feed_forward_train(X)
    error_output = z_o - Y.reshape(-1,1)
    error_hidden = np.matmul(error_output, output_weights.T) * a_h * (1 - a_h)
```

where a_h * (1 – a_h) is the derivative of the sigmoid function.

We will first attempt to tailor this algorithm to the Franke function data by fine-tuning the learning rate and the regularization parameters. L2 regularization updates the general cost function by adding another term known as the regularization term. Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent. The following equation shows this mathematically

$$Cost\ function = Loss + \frac{\lambda}{2m} * \sum \|w\| \tag{5}$$

where λ is the regularization parameter [4]. The result from this is that the weights will decay towards zero, but will never actually reach zero. The weights will be initialized randomly with NumPy's "random.randn" and the bias will be set to 0.01, and our code will be compared to sklearn's "MLPRegressor", which is a multi-layer perceptron regressor that optimizes the squared-loss using stochastic gradient descent.

2.3 Testing different activation functions

In addition to the already mentioned Sigmoid activation function, we will test the RELU and the leaky RELU functions for the hidden layers. RELU stands for Rectified Linear Unit, and is a type of activation function. Mathematically, it is defined as y = max(0,x). Visually it looks like the following:

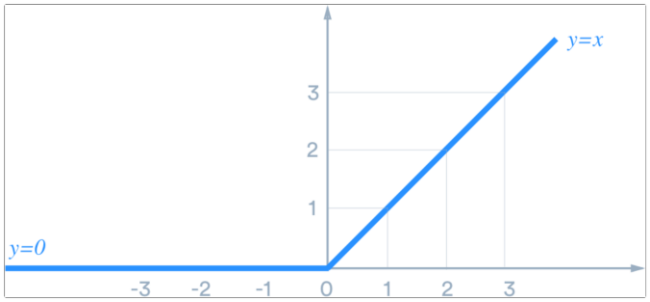

Figure 2: Visual representation of the RELU activation function

RELU is linear for all positive values, and zero for all negative values. This means that it's cheap to compute as there is no complicated math. Linearity means that it converges fast when the slope doesn't plateau or "saturate" when x gets large. It doesn't have the vanishing gradient problem suffered by Sigmoid [5].

Leaky RELU has a small slope for negative values, instead of altogether zero. For example, leaky RELU may have y = 0.01x when x < 0, which is what we'll use in this project. Leaky RELU attempts to fix the "dying RELU" problem where the function always gives the same output (usually zero). Leaky RELU fixes this problem as it doesn't have zero slope parts. However, it is known that leaky RELU isn't always superior to plain RELU.

2.4 Classification analysis using neural networks

Here we will change the cost function for our neural network code in order to perform a classification analysis. Mean Squared Error is a bad choice for classification problems because by using MSE we assume that the underlying data has been generated from a normal distribution. A dataset that can be classified into categories does not necessarily follow a normal distribution. Instead of MSE, we will now measure the performance of our classification problem with the so-called "accuracy score". The accuracy is here just the number of correctly guessed targets t_i divided by the total number of targets, that is

$$Accuracy = \frac{\sum_{i=1}^{n} I(t_i=y_i)}{n} \tag{6}$$

where I is the indicator function, 1 if $t_i = y_i$ and 0 otherwise if we have a binary classification problem. Here t_i represents the target and y_i the outputs of our FFNN code and n is simply the number of targets t_i.

The output activation function will be changed to the Softmax function because we're now switching to the multi-class classification problem of the handwritten numbers from 0-9 of the MNIST dataset. The Sigmoid function only works for binary systems, but the Softmax activation function assigns decimal probabilities to each class in a multi-class problem. Those decimal probabilities must add up to 1. Hence they form a probability distribution which is shown in the following equation:

$$s(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{7}$$

where we apply the standard exponential function to each element x_i of the input vector x and normalize these values by dividing by the sum of all these exponentials; this normalization ensures that the sum of the components of the outputs vector $s(x)$ is 1 [6].

When these changes have been made to the code, we will start optimizing the hyper-parameters like the learning rate and the regularization parameter like we did previously. The effect of the number of hidden layers and nodes will also be tested. The code will then finally be compared to sklearn's "MLPClassifier".

2.5 The Logistic Regression code for classification

Finally, we want to compare the FFNN code we have developed with a Logistic regression code that includes SGD. These two methods will be compared on the classification problem using the same cost function as in the previous case, accuracy score. When we're transitioning from linear regression to logistic regression we need to introduce the logistic function (equation 9)

$$ln\left(\frac{P}{1-P}\right) = w_0 + w_1 x \tag{9}$$

where P is the probability that an event will occur, x is the predictor variable and the b values are the weights (linear parameters). Logistic regression makes use of the Sigmoid function, giving outputs between 0 and 1, but by adding additional expressions $w_2 x + w_3 x + \cdots + w_n x$, the algorithm can handle multi-class problems as well [7]. Below is an illustration of the differences between linear and logistic regression which shows the logarithmic nature of the logistic function.

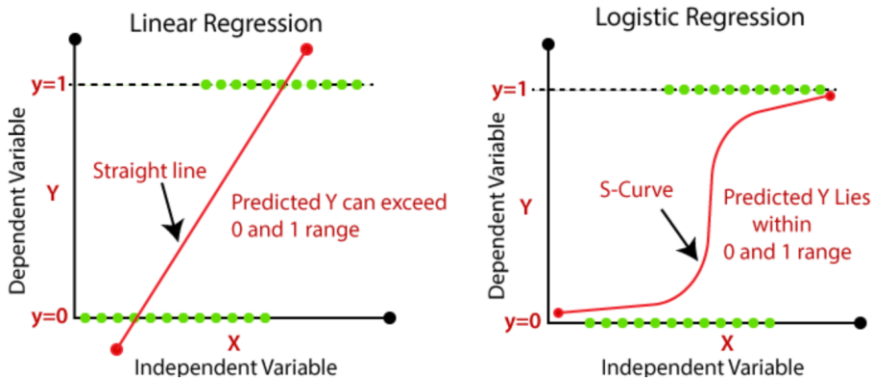

Figure 3: Graphs of linear and logistic regression to illustrate the differences. Notice how in logistic regression the Y values are constricted to being between 0 and 1.

For comparison, we will test our code against sklearn's "LogisticRegression".

3 Results and discussion

3.1 The Stochastic Gradient Descent implementation

The optimal parameters and results with the OLS method with mini-batch SGD are shown in table 1 below. The results from the Ridge regression method with mini-batch SGD are shown in table 2 and figure 4. We used a learning rate schedule that performed best while starting at 0.5/50i, and therefore starting at 0.01 and decreasing as iterations increase.

OLS	
Optimal train batch-size	6
Optimal number of train epochs	85
Optimal test batch-size	9
Optimal number of test epochs	94
Train MSE	0.012
Test MSE	0.013

Table 1: The optimal parameters with the OLS method using mini-batch SGD and the resulting MSE for the train and test data with these parameters.

Ridge regression	
Optimal train batch-size	9
Optimal number of train epochs	55
Optimal test batch-size	9
Optimal number of test epochs	55
Optimal lambda	2.33e-09
Train MSE	112
Test MSE	21.5

Table 2: The optimal parameters with the Ridge regression method using mini-batch SGD and the resulting MSE for the train and test data with these parameters.

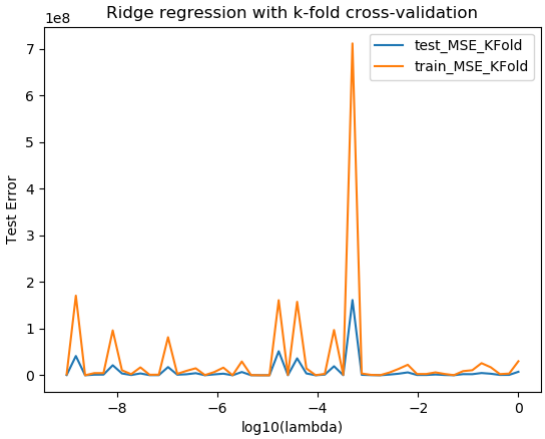

Figure 4: The lambda parameter plotted against MSE with the Ridge regression method.

On the Franke function data, it is very clear that the results point in the direction of OLS being superior, with a test MSE that is more than 1500 times better. The reason for this may very well be the added complexity and number of parameters because the dataset seems to work better for a simpler algorithm like OLS. Ridge regression seems to be overcomplicating the solution to the problem, either by being harder to fine-tune/optimize or simply because it isn't that good for this type of data. Figure 4 gives us an idea of how the error fluctuates to the extremes with just a tiny change in the lambda value.

3.2 The Feed Forward Neural Network code on Franke function data

In table 3 below, we can find the results from our FFNN backpropagation algorithm for the Franke function data. The code had overflow error for higher learning rates, and this optimal learning rate is the highest value that the program could run. The number of hidden neurons was set to 100, and the number of epochs was set to 10 because the algorithm converged at around 7-8 epochs every run. In table 4, we see the results from sklearn's "MLPRegressor" which serves as a benchmark for our own FFNN code.

Feed Forward Neural Network (Sigmoid)	
Feedforward train MSE	2915
Feedforward train R2	-30310
Optimal learning rate	0.00063
Optimal lambda	0.042
Backpropagation train MSE	34
Backpropagation test R2	-48117116

Table 3: The optimal parameters and resulting MSE and R2 score for the FFNN algorithm with backpropagation. The activation function used was Sigmoid.

MLPRegressor (Sigmoid)	
Optimal learning rate	0.0545
Optimal lambda	1e-10
Optimal accuracy score	0.967

Table 4: The optimal parameters and accuracy score for sklearn's MLPRegressor on the Franke function data. The activation function used was Sigmoid.

From these results, we can conclude that our own FFNN code has some bugs and/or mistakes that need to be corrected, as we are getting much better results with the benchmark code. Because of this, it's not possible to fairly compare FFNN against OLS or Ridge on the regression problem yet. However, the run with MLPRegressor points to the algorithm working pretty well for this type of problem.

3.3 Testing different activation functions

Feed Forward Neural Network (RELU)	
Feedforward train MSE	7655
Feedforward train R2	-79585
Optimal learning rate	0.00047
Optimal lambda	0.00079
Backpropagation train MSE	31
Back propagation test R2	-3.4e+54

Table 5: The optimal parameters and resulting MSE and R2 score for the FFNN algorithm with backpropagation. The activation function used was Sigmoid.

Feed Forward Neural Network (leaky RELU)	
Feedforward train MSE	7846
Feedforward train R2	-81576
Optimal learning rate	1e-07
Optimal lambda	1e-10
Backpropagation train MSE	71
Backpropagation test R2	-681177486

Table 6: The optimal parameters and resulting MSE and R2 score for the FFNN algorithm with backpropagation. The activation function used was Sigmoid.

MLPRegressor (RELU)	
Optimal learning rate	0.0545
Optimal lambda	1e-10
Optimal accuracy score	0.993

Table 7: The optimal parameters and accuracy score for sklearn's MLPRegressor on the Franke function data. The activation function used was RELU.

It's premature to draw any concrete conclusions about RELU and leaky RELU compared to Sigmoid with our own FFNN code, but we see a clear tendency of RELU being the better activation function compared to Sigmoid when using MLPRegressor. However, leaky RELU wasn't an option with sklearn's solver.

When changing the weights initialization to 0 we get about the same results or worse with Sigmoid and RELU, but significantly better results for leaky RELU's R2 score of -1000. Changing the biases to 0 did not have any noticeable effect.

3.4 Classification analysis using neural networks

Feed Forward Neural Network	
Feedforward training score	0.144
Optimal learning rate	1e-07
Optimal lambda	1e-10
Backpropagation training score	0.997
Accuracy score on test data	0.919

Table 8: The optimal parameters and accuracy score for our FFNN algorithm for the classification problem. Sigmoid was used for the hidden layers, and Softmax for the output layer.

As seen in table 8, we got an accuracy score of 0.919 on the test data with our FFNN backpropagation algorithm. This means that this model labels the handwritten numbers correctly about 92% of the time. When increasing the number of hidden neurons from 50 to 200 this accuracy increases to 94%, and 94.4% with 400 hidden neurons. By comparison, a benchmark with sklearn's MLPClassifier got an accuracy score of 0.986 with optimal learning rate and lambda at 0.0298 and 1e-10, respectively. This reveals that our own code has the potential to have its parameters optimized better, and to get better results.

3.5 The Logistic Regression code for classification

Logistic regression	
Optimal learning rate	0.00026
Optimal lambda	1e-10
Optimal accuracy score	0.972

Table 8: The optimal parameters and accuracy score with the logistic regression algorithm for the classification problem.

The logistic regression algorithm performs almost as good as sklearn's MLPClassifier score of 0.986 while predicting the handwritten numbers. By comparison, sklearn's LogisticRegression got an accuracy score of 0.95, which also isn't as good as the best performing algorithm for the classification problem. The reason for this is that a neural network is more complex than logistic regression, and for more complicated classification problems like this one, the more complicated model comes out on top. We can simulate a logistic regression model using a neural network with one hidden node with the identity activation function, and one output node with zero bias and logistic sigmoid activation. In principle, anything we can do with logistic regression, we can also do with a neural network. Therefore, theoretically, a neural network is always better than logistic regression, or at least just as precise [8].

The corresponding confusion matrix for our logistic regression algorithm can be seen in figure 5. We can see that our model is particularly good at predicting the handwritten numbers 1, 2, 4, and 6, but also very good overall. Figure 6 shows a few examples of when the predicted numbers didn't correspond to the actual number. Some of them can be a bit challenging even for human eyes.

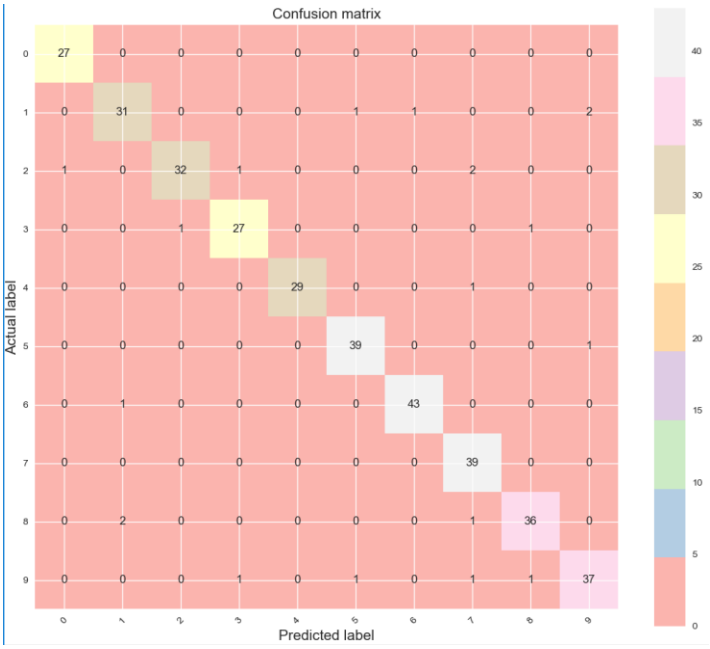

Figure 5: The corresponing confusion matrix for the logistic regression method.

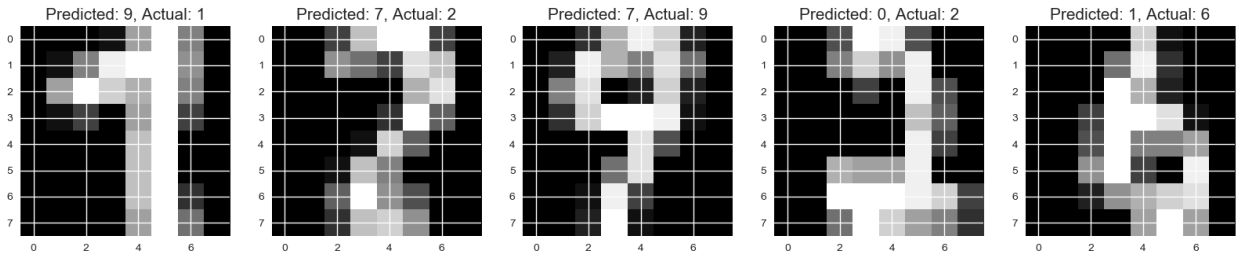

Figure 6: A visualization of some of the predictions that the logistic regression algorithm got wrong.

4 Conclusion

Looking first at the regression problem of the Franke function data, we can first take a look at the Ordinary Least Squares method. With implemented Stochastic Gradient Descent we managed to get the test MSE down to 0.012, which is slightly better than the plain OLS from project 1 [9] with a test MSE of 0.014. However, when implementing SGD into our Ridge regression code from project 1, the MSE increases from 0.007 to 112. The simplicity of OLS comes out on top after adding SGD and comparing these two methods. Next, we can take a look at the OLS method compared to our neural network method. The FFNN algorithm produced a test MSE of 34, which is better than our results with Ridge, but far from the accuracy of OLS. The reasons why OLS is so superior to the other two methods is likely because it's more difficult to tune the parameters correctly in the more complicated algorithms, and for this dataset, it's also sufficient with a simple model. Comparing the different activation functions for the FFNN algorithm on the regression problem, we found that RELU's test MSE of 31 performed better than Sigmoid's 34 and leaky RELU's 71. The RELU activation function's strength could also be found while using sklearn's MLPRegressor, achieving an accuracy score of 0.993. The MLPRegressor method reveals

that a properly implemented neural network can perform very well on regression problems. However, when looking at our own testing after implementing SGD, we have to conclude that the OLS method performs best on the Franke data.

Moving on to the classification problem of the handwritten numbers, we found that after a bit of parameter tuning, the FFNN code got an accuracy score of 0.944, and the logistic regression code got an accuracy score of 0.972. Sklearn's "MLPClassifier" and "LogisticRegression" got accuracy scores of 0.986 and 0.950, respectively. It is safe to say that from these results the neural network algorithm comes out on top as long as it is optimally tuned/implemented, because we reached a training score of 0.997, so the potential for further improvement is there.

Future work:

Improve the Ridge regression code and the FFNN code for the regression problem to discover if the suspicions of a mistake in the code are correct. Optimize more parameters in the FFNN code. Test the codes on other regression and classification problems.

5 Appendix

Here you can find the python codes used in addition to a test of input and output for each code:

https://github.com/gery2/FYS-STK-4155---Project-2

6 References

1. Aishwarya V Srinivasan, *Stochastic Gradient Descent – Clearly Explained* 2019, accessed 9 November 2020: https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31

2. djmw, *What is a feedforward neural network?* 2004, accessed 9 November 2020: https://www.fon.hum.uva.nl/praat/manual/Feedforward_neural_networks_1__What_is_a_feedforward_ne.html

3. Daniel Kobran and David Banys, *Weights and biases* 2019, accessed 9 November 2020: https://docs.paperspace.com/machine-learning/wiki/weights-and-biases

4. Shubham Jain, *An Overview of Regularization Teqhniques in Deep Learning* 2018, accessed 9 November 2020:

https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/.

5. Danqing Liu*, A Practical Guide to ReLU* 2017, accessed 9 November 2020: https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7

6. Victor Zhou, *A Simple Explanation of the Softmax Function* 2019, accessed 9 November 2020: https://victorzhou.com/blog/softmax/

7. Clare Liu, *Linear to Logistic Regression, Explained Step by Step* 2020, accessed 9 November 2020: https://www.kdnuggets.com/2020/03/linear-logistic-regression-explained.html

8. James D. McCaffrey, *Why a Neural Network is Always Better than Logistic Regression* 2018, accessed 9 November 2020: https://jamesmccaffrey.wordpress.com/2018/07/07/why-a-neural-network-is-always-better-than-logistic-regression

9. Wetteland, V. *Regression analysis and resampling methods*, University of Oslo, Norway, October 10, 2020: https://github.com/gery2/FYS-STK4155---Project-1/blob/main/Report/1602331789273_FYS-STK4155%20-%20Project%201.pdf

10. Project 2, *Classification and Regression, from linear and logistic regression to neural networks,* Department of Physics, University of Oslo, Norway, Fall semester 2020: https://compphysics.github.io/MachineLearning/doc/Projects/2020/Project2/html/Project2.html