Classification and Regression
From linear and logistic regression to neural networks

Written by:

Jens Bratten Due
Oliver Lerstøl Hebnes
Mohamed Ismail

Department of Physics UiO

November 8, 2019

Abstract

In this project, both a logistic regression and a neural network is implemented with respect to a classification problem, where the data in question is the Wisconsin breast cancer [1]. Later on, the neural network is also used to perform a regression on a data set generated from the Franke function. Throughout the report, all implemented methods are compared with alternatives provided in the Scikit-learn library. In addition, the regression case is also compared with standard regression methods such as ordinary least squares, ridge regression and lasso regression, all provided from a previous project [2]. For analyzing the quality of the models in classification, F1-scores and cumulative gain curves are utilized to determine the best performers. Our neural network proves to be the most accurate classifier, with an F1-score of 0.98, barely outperforming Scikit-learn with a score of 0.97.

In comparison, the regression results paint a different picture with respect to the R^2-score, where Scikit-learn's neural network comes out on top with a score of 0.96. Our neural network produces a score of 0.90, which is lower than the specialized regression algorithms, which all have scores above 0.90. In the end it is concluded that our own implementation of the neural network is the best model predicting the Wisconsin breast cancer data, and Scikit-learn's neural network is the better choice for the Franke function, as well as being the better all-round performer, with high scores for all tests.

Contents

I Introduction

The idea of assigning outcomes in categories has been part of human history, since the dawn of time. This is how humans and alike has evolved, by appraising items or assessing people and situations. Such as food being edible or poisonous, a situation being life threatening or safe or a partner being eligible or ineligible.

The challenges people face nowadays are not so straightforward, and the problems we face now are not so visible to the naked eye. In some cases there are too many variables that can affect the outcome of something, and the human brain may not have the capacity or the ability to assess a situation accurately and precisely. Examples of such trials, could be predicting if a person had a probability of developing a disease or conclude if a person is eligible for a bank loan or as simple as predicting the weather.

Therefore, we are dependent on machines which can tackle these hurdles. By taking advantage of these machines we have available in this century and statistical theory, we can possibly find solutions to problems we thought to be unachievable. This sets a precedent for this project, where the main goal is to develop our own multilayer perceptron (MLP) neural network for studying both regression and classification problems.

In the classification case, we will work with the well known Wisconsin breast cancer data, and attempt to assign the datapoints to their rightful category. In addition, we will also compare our implementation of MLP with other classifiers, such as our own implementation of logistic regression as well as modules provided from the scikit-learn library that perform similar tasks. While in the regression scenario, we will have a return of the infamous Franke function, on which we will try to fit our MLP, and compare the results with other similar algorithms, that was developed in a previous project [2] and codes provided from scikit-learn.

II Theoretical Methods and Technicalities

i Logistic Regression

Assume we have a dataset with $\boldsymbol{x}_i = x_{i1}, x_{i2}, .., x_{ip}$ input data, where we have p predictors for each corresponding output data y_i. The outcomes y_i are discrete and can only take certain values or classes. Logistic regression is a method used to classify data as described above to an assigned category. Logistic regression is most commonly used in binary cases, which is the simplest example of a classification case where there are only two outcomes or categories, for example yes or no, republican or democrat and 0 or 1. In this section we will focus on the binary classification case.

In addition, logistic regression is also thought of as a soft classification model, which means that it outputs the probability for a given datapoint to be assigned a given category rather than a single value. In our case we have two classes with y_i either being equal to 0 or 1. Therefore, the probability that a datapoint belongs to either class can be given by the so-called Sigmoid function.

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

Furthermore, we have the parameters $\boldsymbol{\beta} = \beta_1, \beta_2, ..., \beta_p$ of our fitting of the Sigmoid function, where the probabilities are defined as

$$p(y_i|\boldsymbol{x}_i\boldsymbol{\beta}) = \frac{\exp{(\boldsymbol{x}_i\boldsymbol{\beta})}}{1+\exp{(\boldsymbol{x}_i\boldsymbol{\beta})}}$$

The goal of logistic regression is then to correctly predict the category of a given dataset, which has different outcomes, by using an optimal parameter $\boldsymbol{\beta}$ that maximises the probability of seeing the observed data. How we find the parameters $\boldsymbol{\beta} = \beta_1, \beta_2, ..., \beta_p$ of the model, is to use the principle of maximum likelihood estimation (MLE)

$$P(\boldsymbol{\beta}) = \prod_{i=1}^{n} \left[p(y_i = 1|\boldsymbol{x}_i\boldsymbol{\beta})\right]^{y_i} \left[1 - p(y_i = 1|\boldsymbol{x}_i\boldsymbol{\beta})\right]^{1-y_i}$$

where we obtain the log-likelihood function, which is easier to work with, since the log-likelihood turns the exponentials into summations [3].

$$C(\boldsymbol{\beta}) = \sum_{i=1}^{n} \big(y_i(\boldsymbol{x}_i\boldsymbol{\beta}) - \log{(1+\exp{(\boldsymbol{x}_i\boldsymbol{\beta})})}\big)$$

Finally, we take our cost function to be the so called cross entropy which is defined as the negative log-likelihood.

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \big(y_i(\boldsymbol{x}_i\boldsymbol{\beta}) - \log{(1+\exp{(\boldsymbol{x}_i\boldsymbol{\beta})})}\big)$$

So to maximize the accuracy and precision of the logistic regression model, we need to find the optimal parameters $\boldsymbol{\beta}$ by minimising the cross entropy. The method for doing so is discussed further below.

ii Neural Network

A multi layer neural network is a computational system that consists of an input layer, hidden layers and an output layer, where every layer again consists of a given number of neurons. These neurons connect with each other through mathematical functions, and the connections are loaded with weights as variables. There are several types of neural networks, and in this project we will focus on the multi layer perceptron with a backpropagation algorithm.

The idea behind a neural network is to mimic the interactions in the brain. The output y from the output layer is given by the activation function f and the activation z,

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

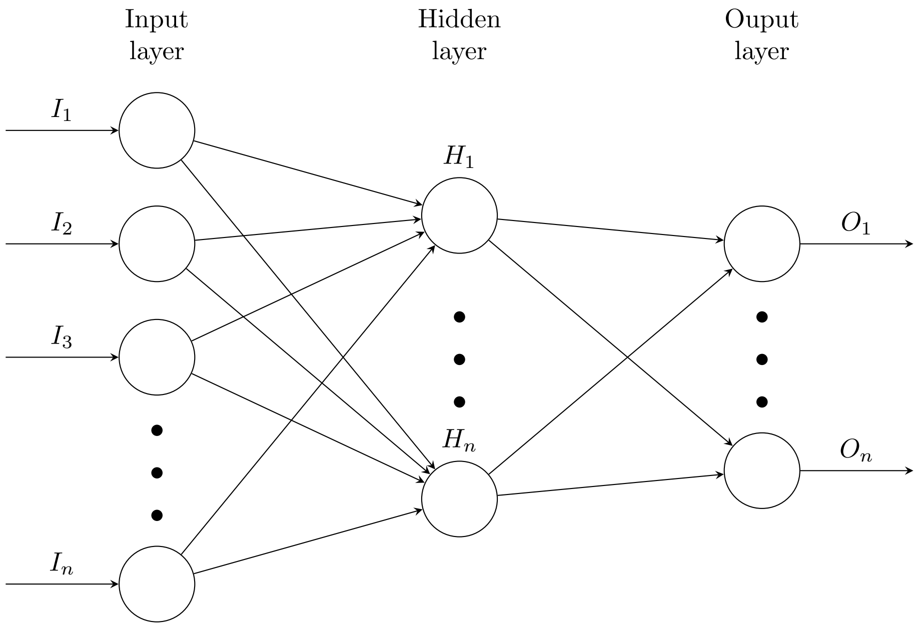


Figure 1: A schematic drawing of a neural network with one input layer, one hidden layer and one output layer. The amount of neurons in each layer does not need to be the same.

where x_i are the inputs, w_i are the weights and b_i are the bias for each neuron i. For more layers, x_i will be outputs for the preceding layer in a feed forward neural network. This can be generalized for l layers.

$$y_i^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^l f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^{l-1}\left(\dots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\dots\right) + b_k^{l-1}\right) + b_1^l\right]$$

Here, x_n is the only input needed to generate an output from the last layer $l+1$, which is also known as the output layer. This algorithm activates neuron i in the l-th layer, giving rise to the name "feed forward neural network". This alone, however, does not allow the neural network to improve and learn.

Thus, we will define a cost function to maximise the quality of the model.

$$\begin{aligned}\mathcal{C}(\hat{W}) &= \frac{1}{2}\sum_{i=1}^{n}(y_i - t_i)^2 \\ &= \frac{1}{2}\sum_{i=1}^{n}\left(a^l - t_i\right)^2,\end{aligned}$$

For every layer there will be outputs a^l from the previous layer, and we can find the change in both weights and biases by the gradient of the cost function to obtain the following equations. This is also known as the backpropagation algorithm.

$$\begin{aligned}
\delta_j^L &= f'(z_j^L)\frac{\partial \mathcal{C}}{\partial (a_j^L)} \\
\delta_j^l &= \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l) \\
w_{jk}^l &\leftarrow w_{jk}^l - \eta \delta_j^l a_k^{l-1} \\
b_j^l &\leftarrow b_j^l - \eta \frac{\partial \mathcal{C}}{\partial b_j^l} = b_j^l - \eta \delta_j^l
\end{aligned}$$

Here we define the error of neuron j in layer l as δ_j^L in the purpose of scaling the weights and the biases to improve our cost function, because we want it as small as possible. The derivation of the algorithm can be found in the course's GitHub repository [3].

i Activation function

There are several options for choosing an activation function, but in this project we have chosen to use the sigmoid function

$$S(x) = \frac{1}{1 + e^{-x}}$$

This function maps every $x \in (-\infty, \infty)$ to $S \in (0, 1)$.

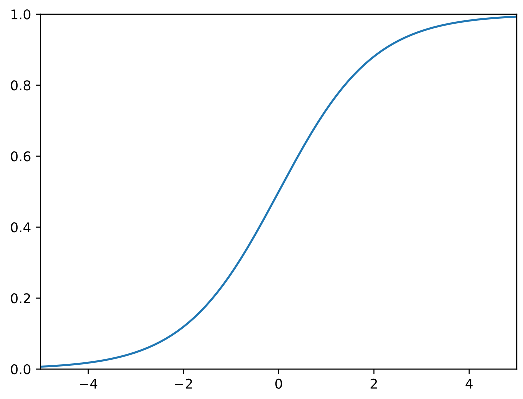


Figure 2: The sigmoid function.

iii Stochastic Gradient Descent

As discussed earlier, for finding the optimal models, we need to minimize our cost functions for both the neural network and the logistic regression. One common numerical method for finding the minimum of a function is stochastic gradient descent, often shortened down to SGD.

The fundamental idea of SGD comes from the observation that the cost function can be written as a sum over n data points $\{\boldsymbol{x}_i\}_{i=1}^{n}$.

$$C(\boldsymbol{\beta}) = \sum_{i=1}^{n} c_i(\boldsymbol{x}_i, \beta)$$

We can then compute the gradient as shown below

$$\nabla_\beta C(\boldsymbol{\beta}) = \sum_{i=1}^{n} \nabla_\beta c_i(\boldsymbol{x}_i, \beta)$$

We can introduce a randomness by only taking the gradient on a small interval of the data, called a minibatch. With n total data points, and M datapoints per minibatch, the number of minibatches is then $\frac{n}{M}$.

The idea is now to approximate the gradient by replacing the sum over all data points with a sum over the data points in one of the minibatches picked at random in each gradient descent step.

$$\nabla_\beta C(\boldsymbol{\beta}) = \sum_{i=1}^{n} \nabla_\beta c_i(\boldsymbol{x}_i, \beta) \rightarrow \sum_{i \in B_k}^{n} \nabla_\beta c_i(\boldsymbol{x}_i, \beta)$$

Where B_k is the set of all minibatches, with $k = 1, .., \frac{n}{M}$. One step of gradient descent is then defined by

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(\boldsymbol{x}_i, \beta)$$

where k is picked at random with equal probability from $[1, \frac{n}{M}]$. An iteration over the number of minibathces ($\frac{n}{M}$) is commonly referred to as an epoch. Thus it is typical to choose a number of epochs and for each epoch iterate over the number of minibatches.

However, a problem arises. When does one stop searching for a new minimum? A well known approach is to let the step length γ_j depend on the number of epochs in such a way that it becomes very small after a reasonable time such that we do not move at all.

The way we have implemented it is by defining the epochs as $e = 0, 1, 2, ..$ and $t_0, t_1 > 0$, be two fixed numbers. Furthermore, we defined $t = e \cdot m + i$, where m is the number of minibatches and $i = 0, ..., m - 1$. Then the step length γ_j is defined as

$$\gamma_j(t; t_0 t_1) = \frac{t_0}{t + t_1}$$

and goes to zero as the number of epochs gets larger.

In this way we can fix the number of epochs, compute β and evaluate the cost function at the end. Repeating the computation will give a different result since the scheme is random by design. The $\boldsymbol{\beta}$ that results in the lowest value of the cost function is then the most optimal parameter [3].

iv Performance Measure

i Confusion Matrix & F-score

A confusion matrix is a method for measuring the performance of classifiers. It is set up as a table with 4 different categories, where two of the categories are the predicted outcomes of the model/classifier and the two final categories are the true outcomes, which one wants to fit.

An example of a confusion matrix for a binary classifier predicting the presence of a pregnancy is shown below.

Total number of datapoints/people (N) N = 200	Predicted: NO	Predicted: YES	Sum
Actual: NO	TN 60	FP 5	65
Actual: YES	FN 10	TP 125	135
Sum	70	130	200

Table 1: Illustration of the confusion matrix

The information one can extrapolate from this table is:

- There are two possible predicted outcomes: positive and negative.
- The classifier made a total of 200 predictions of 200 patients being pregnant or not.
- Out of the 200 cases, the classifier predicted positive 130 times, and negative 70 times.
- In truth, 135 patients in the trial are pregnant, and 65 patients are not.

This gives rise to some terminology

- True Positive (TP): The classifier predicted that a person is pregnant and it is true.
- True Negative (TN): The classifier predicted that a person is not pregnant and it is true.
- False Positive (FP): The classifier predicted that a person is pregnant and it is false.

- False Negative (FN): The classifier predicted that a person is not pregnant and it is false.

From the confusion matrix one can then start estimating the performance of the model, by calculating different factors, such as

- **Recall** is how much the classifier predicted correct divided by all the true positive outcomes.

$$Recall = \frac{TP}{TP + FN}$$

- **Precision** is if the classifier predicts positive, how often is it correct.

$$Precision = \frac{TP}{TP + FP}$$

Sometimes a classifier can have drastically different values for the precision and recall. This leads to another estimator for the performance of a classifier, which is known as the F-score.

The F-score is defined as

$$F - score = \frac{2 \cdot Recall \cdot Precision}{Recall + Precision}$$

and is a measure of classifiers accuracy on the data, by using the average of the recall and precision [4].

ii Cumulative gain curve

A cumulative gain curve evaluates the model as function of correct predictions and percentage of sample [5]. As seen in figure 3, it consists of three dashed lines that are equal for all the figures. The black dashed line shows how a random classifier performs, while the orange dashed line shows the ideal case where a classifier predicts all true target for the class benign, while the blue dashed line shows the same for the malignant class. The closer the orange and blue lines are to the striped ones, the better is the model. The orange and blue dashed lines maximises at 0.38 and 0.62, respectively, because this is the percentage of benign and malignant cases. Thus, for a perfect fit, the dashed lines and their corresponding lines should totally overlap.

III Data Sets

i Breast cancer

For our classification, we will use the Wisconcin breast cancer dataset from the UCI repository [1]. It is a binary classification data set included in the scikit-learn library and consists of 569 samples where 357 are benign cases, and 212 are malignant. There are 30 features in the set, corresponding to various physical properties of the cell nuclei in a breast mass.

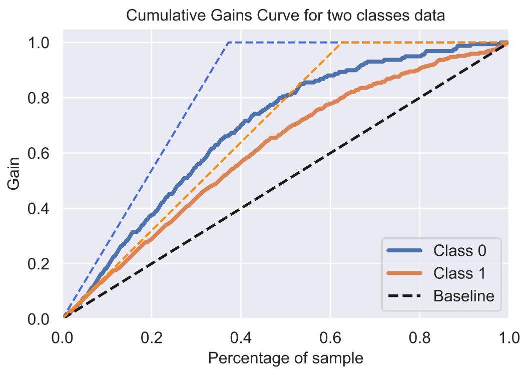


Figure 3: An example of a cumulative gain curve

ii Franke's Function

For performing our regression we will use a data set generated from Franke's function, as we did in project 1 [2]. It is defined as

$$f(x,y) = \frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) + \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right)$$
$$+ \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) - \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right).$$

In addition, stochastic noise with zero mean and standard deviation set to one is added to the function before fitting. $N(0,1)$.

A plot of the Franke function without the noise can be seen in figure 4.

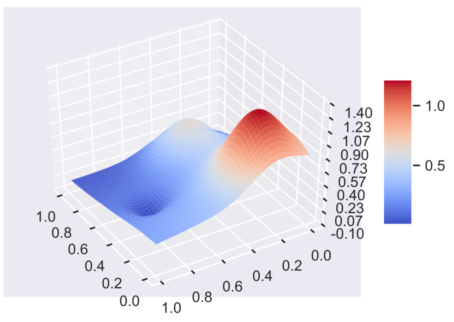


Figure 4: The Franke function plotted for $0 \leq x, y \leq 1$

IV Implementation

Programs used in this project can be found on our github repository [6]. These programs are inspired by the codes found in the course's github repository [3].

i Neural Network

Our implementation of neural network can only take in values that ranges in between $[0, 1]$. In the classification case this has no effect, but in the regression case, where the values in theory ranges from $[-\infty, \infty]$, this has enormous effect. To avoid this, we are mapping the data from the regression case, into values between $[0, 1]$ by using the Sigmoid function

$$S(x) = \frac{1}{1 + e^{-x}}$$

and then mapping the output values from our neural network, into new values that resemble the true data by using the inverted Sigmoid function.

$$x(S) = \ln\big(\frac{S}{1 - S}\big)$$

We discovered that there was no significant loss in numerical precision when scaling the data with the Sigmoid function and then scaling them back with the inverted function. The absolute mean difference in the processed data and the exact data was proportional to 10^{-17}.

ii Finding hyperparameters for neural network

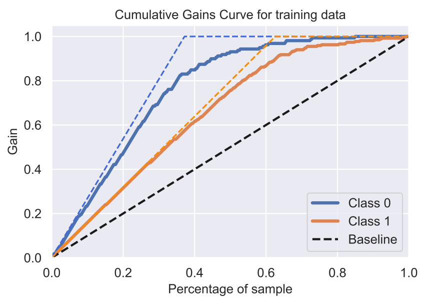
There can easily arise some challenges in training a neural network. One of the difficulties of finding the best model is to find the best parameters. A possibility is to search the entire space of solutions with all the parameters and their range, but unfortunately, this space requires a heavy machinery which we do not possess.

Instead of searching for the entire space of solutions, we have started off with a initial 'guess' of parameters, and then tuning every parameter afterwards. It is known that the learning rate η and the penalty parameter λ is of high importance, thus we do not choose these parameters until the very end of the tuning. This development can be observed in the jupyter notebook on our Github [6].

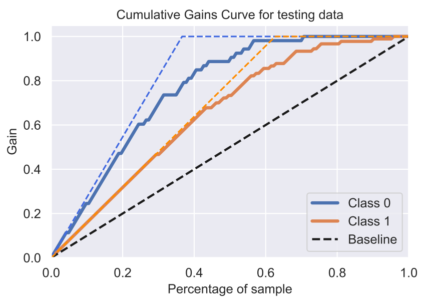
V Results

i Classification

i Logistic regression

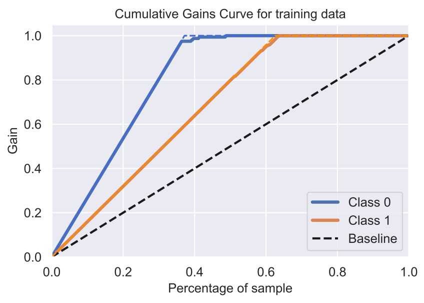


(a)

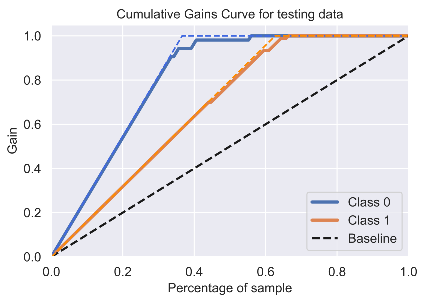


(b)

Figure 5: Cumulative gains curve for (a) training data and (b) testing data for the classification data when using our own logistic regression with a stochastic gradient method.



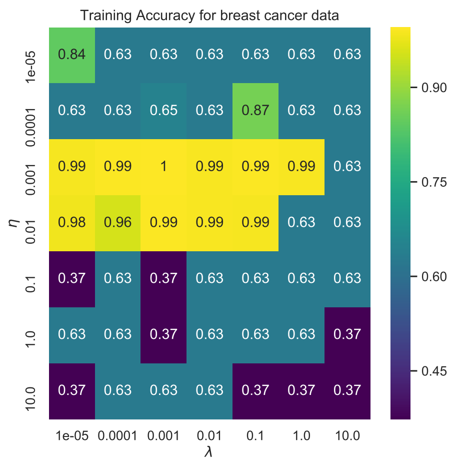
(a)



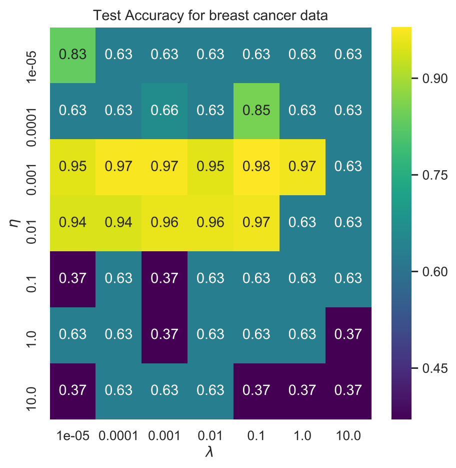
(b)

Figure 6: Cumulative gains curve for (a) training data and (b) testing data for the classification data when using **scikit-learn's** logistic regression with a stochastic gradient method.

ii Neural network

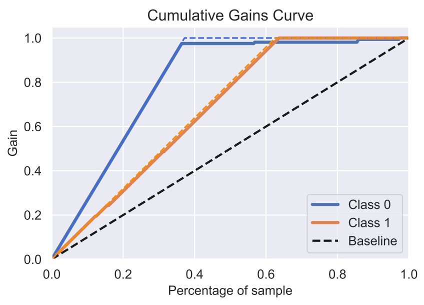


(a)

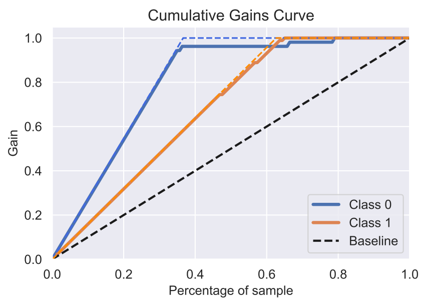


(b)

Figure 7: Heatmap showing accuracy score for (a) training data and (b) testing data for the classification data when using feed forward neural network.

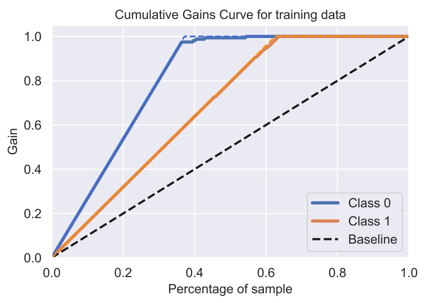


(a)

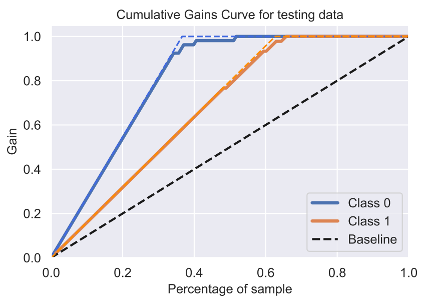


(b)

Figure 8: Cumulative gains curve for (a) training data and (b) testing data for the classification data when using our own neural network with $\lambda = 10^{-1}$ and $\eta = 10^{-3}$.



(a)



(b)

Figure 9: Cumulative gain curve for (a) training data and (b) testing data for the classification data when using **scikit-learn's** neural network.

Table 2: The parameters used for the final model of neural network in both classification and regression.

Method	Epochs	Batch size	Layers	Neurons	η	λ
NN Scikitlearn classification	-	-	1	100	10^{-3}	10^{-4}
NN classification	140	160	6	30	10^{-3}	10^{-1}
NN regression	150	10	5	100	10^{-2}	10^{-7}
NN scikitlearn regression	-	-	1	100	10^{-3}	10^{-4}

Table 3: Highest $F1$-score calculated on the testing data from breast cancer.

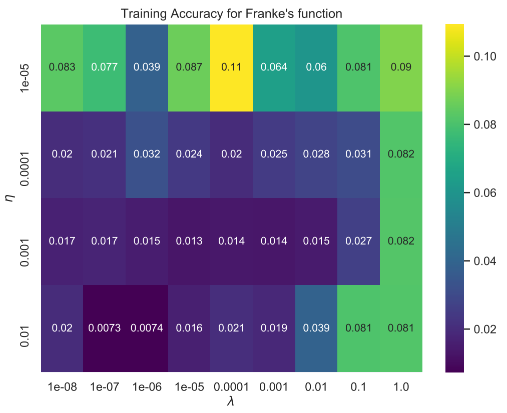
Classification method	F1-score
Logistic regression	0.92
Sklearn logistic regression	0.97
NN	0.98
Sklearn's NN	0.97

ii Regression

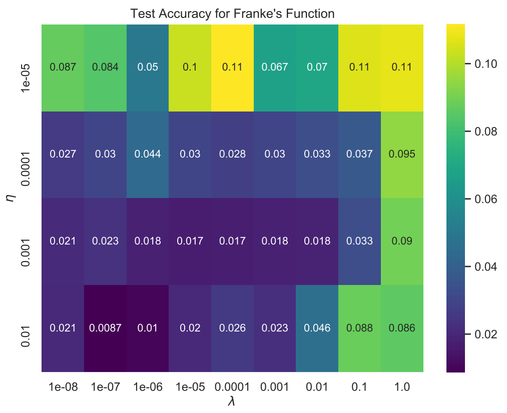
All calculations on Franke's function has been with a total amount of 400 data points with the parameters given in table

Regression method	R^2-score	MSE
OLS	0.92	0.010
Ridge	0.91	0.008
LASSO	0.91	0.007
NN	0.90	0.009
Sklearn's NN	0.96	0.041

Table 4: Lowest mean squared error and R^2 calculated on the testing data of Franke's function.



(a)



(b)

Figure 10: Heatmaps showing the MSE for different parameters λ and η for (a) training data and (b) testing data for Franke's function using our own neural network with $\lambda = 10^{-2}$ and $\eta = 10^{-7}$.

VI Discussion

i Classification

Figure 5 shows the cumulative gains curve for our own implementation of the logistic regression for both training data and testing data. To avoid the danger of overfitting, we include both figures to show how a model that performs good on the training data should also perform similarly good on the testing data. If a model is overfitting, we would expect a worse curve for the testing data, but this is not the case. In table 3 we can also see that the corresponding F1-score is 0.92, which is not a bad case. However, this score seems not to be stable. For every run, the F1-score fluctuates from 0.60 to 0.94. One reason why this might be the cause of is because the breast cancer data is not a big data set. Thus, the testing data will only consist of 25% of the total data, which gives only 142 datapoints. From a big data point of view, this does not qualify as a big data set.

Scikit-learn's logistic regression outperforms our own logistic regression with an F1-score of 0.97, and an overall good fit to the ideal case in figure 6. The figure shows that the model fits almost perfectly to the training data, while there is a bigger error in the testing data. Thus, it might indicate an overfit. However, the F1-score shows that the model also performs very good on the testing data which counter the indication of the overfit-argument.

In the case of the neural network, figure 7 clearly illustrates optimal choices of the learning rate η and penalty rate λ, with respect to both the training and test data. In addition, the figure shows that the best accuracy score is highly dependent on η, and less dependent on λ. Furthermore, by extrapolating from the results in figure 7 it was found that multiple choices gave satisfactory results for the training data, while choosing $\eta = 10^{-2}$ and $\lambda = 10^{-1}$ led to the best result on the test data. These parameters were used in the neural network to produce both cumulative gains curves in figure 9 which shows that we get a tight fit for both the training and testing data. Our own implementation of the neural network yielded an F1-score of 0.98, which is the highest score of the four tested classification methods.

The corresponding neural network from scikit-learn's library yields an F1-score of 0.97 in table 3, which means that our neural network code slightly outperforms scikit-learn's alternative. This neural network is a very stable method, and yields the same results every run. The corresponding cumulative gains show a minor error which translates into a 3% error in the F1-score. It is worth to mention that both our and scikit-learn's neural network use the same learning rate η, and if our neural network had used the same penalty parameter λ as scikit-learn, the F1-score would be approximately the same. Thus, this may indicate that our neural network has proven to be a better model than the alternative from the scikit-learn library. Nonetheless, this is not without sacrifice. We have built a model using time consuming and heavy machinery with six hidden layers and huge batch sizes and epochs, while scikitlearn's package gets almost a equivalent $F1$-score as our own neural network by only using one hidden layer.

ii Regression

For evaluating the neural network's regression performance, similar analysis as for the classification has been performed. In table 4, a stable mean squared error of 0.009 and corresponding R^2-score of 0.90 is found, using a learning rate η of 10^{-2} and penalty parameter λ of 10^{-7}. However, in contrast to the classification case, the penalty parameter plays a much bigger role. It is seen in figure 10 that fine tuning of the penalty parameter is required to achieve an optimal model, as the mean squared error fluctuates from 0.009 to 0.086 within the same learning rate. On the other hand, increasing the range of learning rates to under 10^{-5} or higher than 10^{-2} resulted in inaccurate models with very high mean squared errors, indicating that η has a higher priority than the penalty parameter.

Based on a high accuracy in both training and testing data, the neural network does not seem to overfit the data. It scores similar R^2-scores and mean squared errors as the regression types used in the previous project [2], but overall it does not perform better than the mentioned regression types with the values given in table 4.

Scikit-learn's neural network scores a mean squared error of 0.041 and a R^2 score of 0.96. From the discussion of how to evaluate regression methods in Project 1 on our GitHub repository [2], one can find out that mean squared error is not the correct way to evaluate a noise-based function, and thus, the R^2-score is a better evaluation score. This leads to the conclusion that the neural network from scikit-learn is performing better for regression tasks than our own implementation. This is an interesting contrast to the classification case, where our neural network presumably comes out on top. In addition, scikit-learn's neural network finds a great model using only one hidden layer which indicates the superiority of the package in regression when it is only using a small fraction of the computational time and power needed.

VII Conclusion

In the first part of this project, we have implemented the models logistic regression and neural network to predict malignant or benign cases of the Wisconsin breast cancer data, where we have found that our own implemented neural network gives the best $F1$-score. However, it is inevitable to observe the conveniency of the data set. It is a clean data set which is easy to analyse, without needing to do an extensive preprocessing before the analysis. This might suggest that our neural network may not give the same prediction accuracy for complex and bigger data sets.

In the second part, we have used the implemented neural network to model Franke's function while comparing it to other regression methods like ordinary least squares, ridge regression and lasso regression. This, however, did not result in as good R^2-scores as the specialized methods, leading to an inferior model. The best score was obtained by scikit-learn's neural network which ended up triumphing over all the other regression types, while also being extremely time efficient. All in all, we see that Scikit-learn's neural network consistently delivers high scores across all tested applications, indicating a robust toolset.

For an evaluation of our work, we have seen how easy the implementation of scikit-learn's neural network is, while also producing stable and accurate results. More thorough analysis could also be performed by comparing our network with other machine learning libraries, such as Tensorflow or Keras. This unique experience will be of great value when we further dive into new and undiscovered data sets.

References

[1] UCI Machine Learning Repository: Breast Cancer Wisconsin (Diagnostic) Data Set. Irvine, CA: University of California, School of Information and Computer Science https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)

[2] GitHub repository for project 1 https://github.com/ohebbi/FYS-STK4155/tree/master/project1

[3] Morten Hjorth-Jensen, Github Repository
https://github.com/CompPhysics/MachineLearning/tree/master/doc

[4] Markham, K. Simple guide to confusion matrix terminology. Data School. https://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/

[5] Reiichiro S. Nakano. Metrics Module of scikit-plot. 2017. https://scikit-plot.readthedocs.io/en/stable/metrics.html

[6] GitHub repository containing all code and plots, as well as a Jupyter notebook for easy reproducibiliy https://github.com/ohebbi/FYS-STK4155/tree/master/project2

[7] Trevor Hastie, Robert Tibshirani, Jerome H. Friedman, The Elements of Statistical Learning, Springer https://www.springer.com/gp/book/9780387848570.