FYS-STK 3155 Project 2 Erlend Abrahamsen

November 11, 2019

Abstract

For binary classification on default payment from bank customers (credit data 1.1) we found the best average accuracy at 80%, but with high false-negative rate (24%). This was with a 4 hidden layer neural network, beating logistic regression, and the conclusion was that the model is only good enough as an indication.

For regression on noise generated franke data we found that a 2 hidden layer neural network beats the best linear regression method, Ridge regression, from project 1. The mean squared errors were respectively $1.01 \cdot 10^{-2}$ and $1.08 \cdot 10^{-2}$ which is a good regression.

(Project1 :`https://github.com/ErlendAbrahamsen/FYS-STK3155/blob/master/Project1/Report.pdf`).

1 Introduction

For the source code (folder Source_code) and the results (folder Results) go to my Github: `https://github.com/ErlendAbrahamsen/FYS-STK3155/tree/master/Project2`

Our aim is to analyze logistic regression and an feed forward neural network for binary classification on the credit data presented in section 1.1. I.e. predict if costumer is credible or not.
We also want to try out continuous regression trained on noise generated

franke data for reproducing the real franke function using the neural network. Does it beat linear regression methods. Franke function `https://www.sfu.ca/~ssurjano/franke2d.html`.
Numerical tools used include Newtons method, backpropagation and stochastic gradient descent.

Outline [Introduction, Methods, Results and discussion, Summary]
(I will reference to source code/project part where relevant!)

1.1 Credit data format

This data is taken from an important bank in Taiwan from 2005. The original sample size is 30.000 id's/persons.

y: Default payment: (Yes=1, No=0)

X1: Amount of the given credit (NT dollar): it includes both the individual consumer credit and his/her family (supplementary) credit.

X2: Gender
(1 = male; 2 = female).

X3: Education
(1 = graduate school; 2 = university; 3 = high school; 4 = others).

X4: Marital status
(1 = married; 2 = single; 3 = others).

X5: Age (year).

X6–X11: History of past payment.
We tracked the past monthly payment records (from April to September, 2005) as follows: X6 = the repayment status in September, 2005; X7 = the repayment status in August, 2005;...;X11 = the repayment status in April, 2005. The measurement scale for the repayment status is: 1 = pay duly; 1

= payment delay for one month; 2 = payment delay for two months; ...; 8 = payment delay for eight months; 9 = payment delay for nine months and above.

X12–X17: Amount of bill statement (NT dollar).
X12 = amount of bill statement in September, 2005;
X13 = amount of bill statement in August,2005;
...; X17 = amount of bill statement in April, 2005.

X18–X23: Amount of previous payment (NT dollar).
X18 = amount paid in September, 2005;
X19 = amount paid in August, 2005;
...;X23 = amount paid in April, 2005.

Cited from section 3.1 in
`https://bradzzz.gitbooks.io/ga-seattle-dsi/content/dsi/dsi_05_classification_databases/2.1-lesson/assets/datasets/DefaultCreditCardClients_yeh_2009.pdf`

We look for erros in the data set by checking the following:

y is in range (Only 0's or 1's)
X2, X3, X4, X6-X11 is in range
X1, X5, X12-X17, X18-X23 ≥ 0 (Assuming bank is not in debt to costumer)

The data set is reduced to 3792 data points / costumers before we begin analysis. *(See creditDataReduction() in Part_a.py)*

Basic properties:
y shape: (3792, 1)
X shape: (3792, 24) (Including 1 as constant predictor)

Credible 63.3 %

Non-credible: 36.7 %

2 Methods

2.1 Binary Logistic regression

Let $\mathbf{x} = (1, x_1, ..., x_p)$, $\beta = (\beta_0, ..., \beta_p)^T$.
Our goal is to calculate prediction probabilities on our binary response variable $y_i \in \{0, 1\}$ depending on the linear combination $\mathbf{x}\beta$.

For setting up an probability mass function, the logistic function $p(t) = \frac{exp(t)}{1+exp(t)}$ is used.

We finally get the two probabilities $P(y_i = 1|x_i, \beta) = p(\mathbf{x}\beta)$ and $P(y_i = 0|x_i, \beta) = 1 - p(\mathbf{x}\beta)$.

For optimizing β weights we use the maximum likelihood estimation (MLE), which is maximizing probability prediction of observed data.

I.e. maximazing $P(D|\beta) = \prod_{i=1}^{n} [p(\mathbf{x}\beta)]^{y_i} [1 - p(\mathbf{x}\beta)]^{1-y_i}$, where D is any possible event.

We then get the cost function by computing partial derivatives of P, since we want to maximize P:

$C(\mathbf{x}, \beta) = -\frac{\partial P}{\partial \beta}$

$= \sum_{i=1}^{n} \Big(y_i \log[p(\mathbf{x}\beta)] + (1 - y_i) \log[1 - p(\mathbf{x}\beta)] \Big)$

Now we want to find $\frac{\partial C(\beta)}{\partial \beta}$ and $\frac{\partial^2 C(\beta)}{\partial \beta \partial \beta^T}$ for minimizing cost:

Define the designmatrix (x_{ij} is the i'th sample value of j'th predictor)

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1p} \\ 1 & x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix}$$

and the vector $\mathbf{p} = (p(X_1\beta), ..., p(X_n\beta))^T$ (X_i is the i'th row of X)

The derivatives for n samples can be expressed as:

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

$\frac{\partial^2 C(\beta)}{\partial \beta \partial \beta^T} = X^T W X$, where $W = diag(\mathbf{p}(\mathbf{1} - \mathbf{p}))$. (element wise product)

(*See dlogcost() in Part_b.py*)
We cannot minimize the cost analytically, but we can attempt numerically with e.g. Newton's method.

2.1.1 Newton's method

The derivation of the 1d case $f(x)$ is derived by an 2. order taylor expansion of $f(x)$.

We get $x_{n+1} = x_n - f'(x_n)/f''(x_n)$, this is extended to higher dimensions as

$\mathbf{x_{n+1}} = \mathbf{x_n} - [\mathbf{H}f(\mathbf{x_n})]^{-1} \nabla f(\mathbf{x_n})$, with the hessian and the gradient.

Setting up our case we also add additional step size η:

$\beta^{i+1} = \beta^i - \eta \left[\frac{\partial^2 C(\beta)}{\partial \beta \partial \beta^T} \right]^{-1} \left[\frac{\partial C(\beta)}{\partial \beta} \right]$ (Hessian, gradient computed in section 2.2)

$\beta^{i+1} = \beta^i - \eta [X^T W X]^{-1} X^T(\mathbf{p} - \mathbf{y})$ **

Now we choose some initial guess for $\beta^0 = (\beta_0^0, ..., \beta_p^0)$ and iterate **, until convergence or maximum number of iterations get exceeded. We can also try different η values to ensure that we dont end up in local minima.
(*For implementation see newtonsMethod in part_b.py.*)

2.2 Feed forward neural network using back propagation with stochastic gradient descent

2.2.1 Basic setup

Our aim is to predict probabilities on our binary response variable $y_i \in \{0, 1\}$ depending on activations of weighted linear combinations with biases/constants $W\mathbf{x} + \mathbf{b}$.
This is almost the same as in logistic regression except that we can have multiple/iterative functions called activation functions.
We would also like to implement continuous output for regression on Franke data with noise.

Setting up our feed forward neural network we have multiple layers such as an input layer, hidden layers and an output layer.
Note that feed forward means no loops in the network.
Figure of neural network with 2 hidden layers:

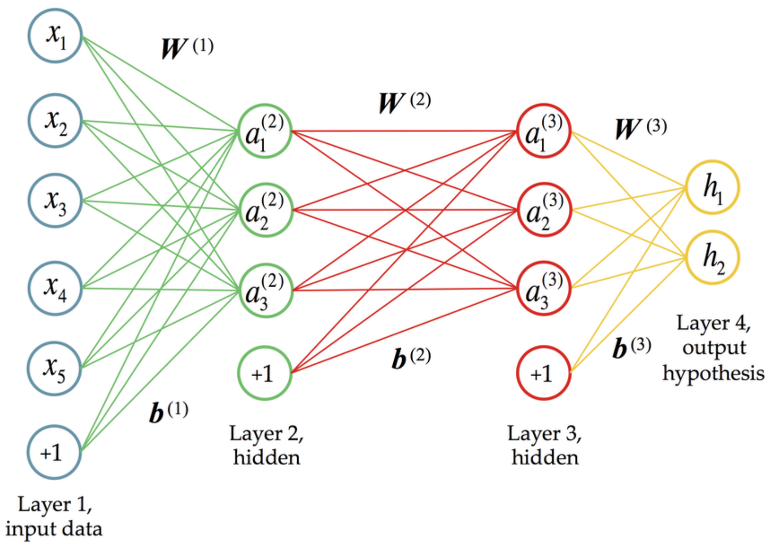


Figure source: https://www.researchgate.net/publication/299474560

We will attempt to setup an feed forward neural network (FFNN) capable of multiple hidden layers. Before we start, note that the python implementation uses slightly different notation than explained here. (Wrote the setup before implementing it)
Setting up matrices and vectors:

Given the activation function σ we have the l'th layer activation as $\mathbf{a}^l = \sigma(W^l\mathbf{a}^{l-1} + \mathbf{b}^l)$, $l = 1, ..., L$ with $\mathbf{a}^1 = \mathbf{x} \in R^{23}$.

For optimizing W and $\mathbf{b}$ we need to assign a cost function $C(\mathbf{y}, \mathbf{a}^L)$ and minimize it using the gradient.
Calculating the gradients is done with a process called back propagation.

2.2.2 Back propagation

For notation purposes let $\mathbf{z}^l = W^l\mathbf{a}^{l-1} + \mathbf{b}^l$ and $\delta^l = \frac{\partial C}{\partial \mathbf{z}^l}$.

By the chain rule we have from the last layer

$\delta^L = \frac{\partial C}{\partial \mathbf{a}^L} \cdot \frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L} = \frac{\partial C}{\partial \mathbf{a}^L} \cdot \sigma'(\mathbf{z}^L)$

By chain rule and vector summations it can also be shown that

$\delta^l = [(W^{l+1})^T \delta^{l+1}]\sigma'(\mathbf{z}^l).$

Finally we get the scheme:

(*feed_forward() in NeuralNetwork class*)

Let $\mathbf{a}^1 = \mathbf{x}$. Compute $\mathbf{a}^l = \sigma(\mathbf{z}^l)$ for $l = 2, ..., L$.

(*back_propagation() in NeuralNetwork class*)

Let $\delta^L = \frac{\partial C}{\partial \mathbf{a}^L} \cdot \sigma'(\mathbf{z}^L)$ and compute

$\delta^l = [(W^{l+1})^T \delta^{l+1}]\sigma'(\mathbf{z}^l)$ for $l = L - 1, ..., 2$.

$\frac{\partial C}{\partial b_i^l} = \delta_i^l$

$\frac{\partial C}{\partial W_{ij}^l} = a_j^{l-1}\delta_i^l$

Since we have the gradients/partials we can use gradient descent to find minima. Also, see Cost- and activation derivatives for derivation of used formulas in the NeuralNetwork class.

2.2.3 Stochastic gradient descent

This is an approximation of gradient descent using less data.
We still have the standard gradient descent equations of weighs and biases with an given learning rate η:

$W^l = W^l - \eta \nabla C(W^l, X)$

$\mathbf{b}^l = \mathbf{b}^l - \eta \nabla C(\mathbf{b}^l, X)$

The difference is that we choose so called randomly selected mini-batches of X, instead of using the whole data set, which converge faster for bigger data sets.
Process goes as follows:

Let n, S be number of datapoints and batch size.
Iterate $n//s$ times (whole division) with an randomly selected mini-batch of X with size S.

2.2.4 Cost- and activation functions with derivatives

We implement cross-entropy (log cost) and square error (mse cost) as possible cost functions. cross-entropy cost at final layer L:

$C(W) = -\sum_{i=1}^{n} \Big(y_i \log a_i^L + (1 - y_i) \log(1 - a_i^L) \Big)$

We get the partials $\frac{\partial C}{\partial a_i^L} = \frac{a_i^L - y_i}{a_i^L (1 - a_i^L)}$.

For the square error cost $C = \frac{1}{2} \sum_{i=1}^{n} (y_i - a_i^L)^2$ *

we get the partials $\frac{\partial C}{\partial a_i^L} = a_i^L - y_i$. **
(** and ** are used in backpropgation() method in NeuralNetwork class Part_c.py*)

For choices of activation functions we use sigmoid, tanh and relu. These derivatives can easily be calculated.
(*activ() method in NeuralNetowork class Part_c.py contains these derivatives*)

$\sigma'(z^L) = \frac{exp(-z^L)}{(1+exp(-z^L))^2} = \frac{exp(-z^L)}{1+exp(-z^L)}(1 - \frac{exp(-z^L)}{1+exp(-z^L)}) = a^L(1-a^L)$

$tanh'(z^L) = 1 - tanh^2(z^L) = 1 - (a^L)^2$ (element wise product)

$relu'(z_i^L) = 1$ for $z_i^L > 0$ and 0 for $z_i^L \leq 0$.

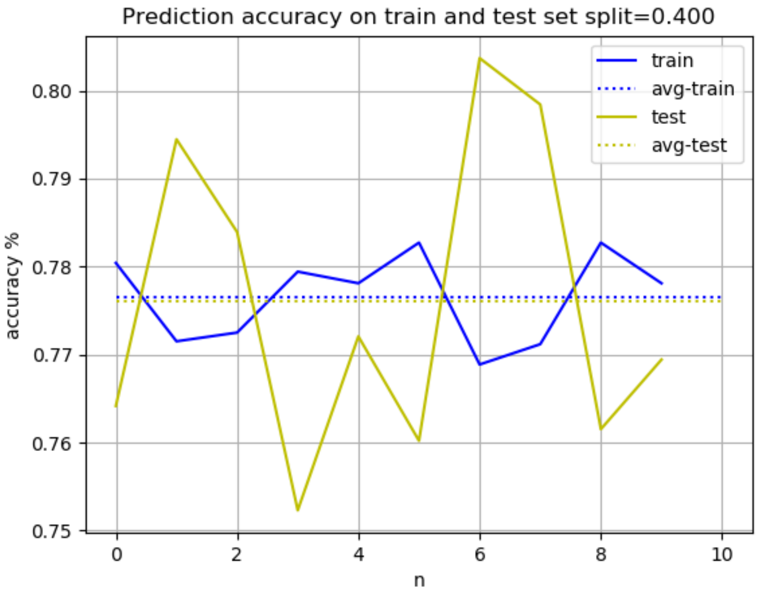
In the output layer we use softmax function for classification probabilities, and the linear transform $T(A) = \frac{1}{max(A_{ij})}A$ for continuous regression output.

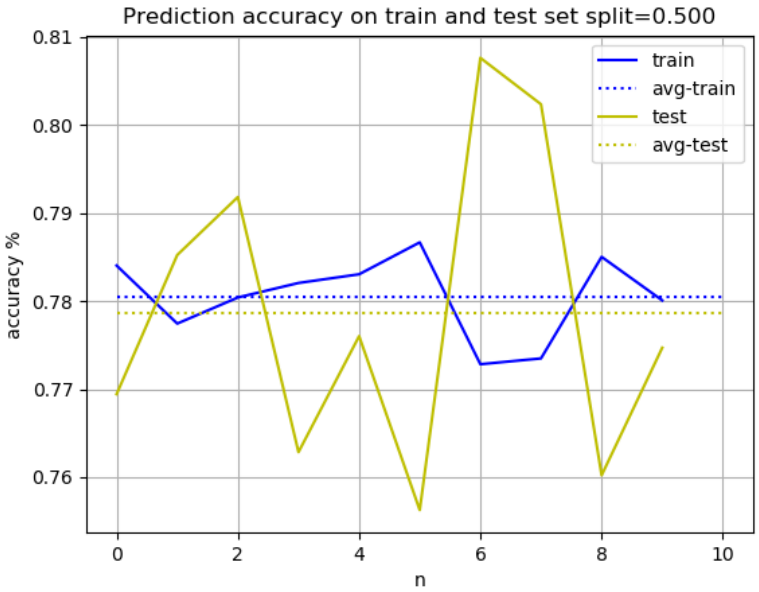
3 Results and discussion

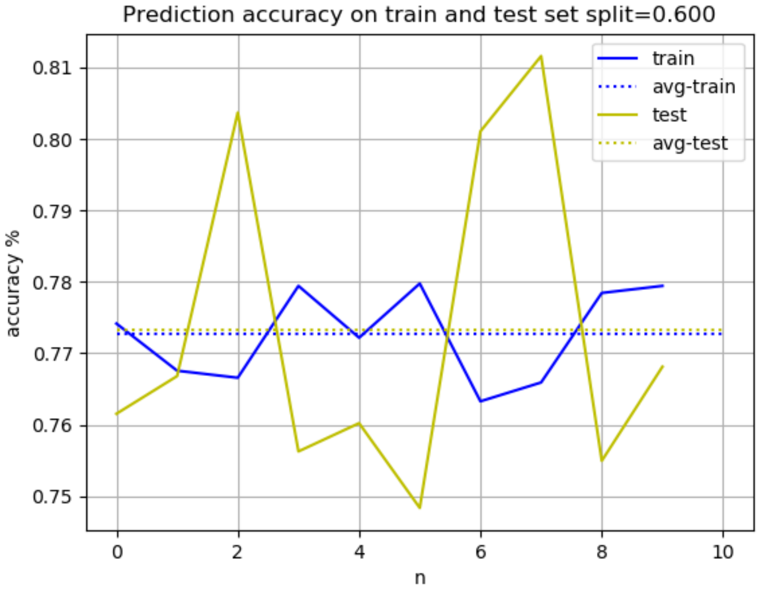
3.1 logistic regression for credit predictions (Part B)

We split the data $X, \mathbf{y}$ into training and testing data using 20% as testing data, and perform binary logistic regression.
Using the training data to find β coefficients we do predictions on both the training set and testing set.
The output is a vector containing probabilities of $y_i = 1$.

We compute the accuracy of our predictions (Correctly guessed / total) using some different splitting threshold.
Splitting threshold = 0.5 predicts $\tilde{y_i} = 1$ if $prob(y_i = 1) \geq 0.5$.
Plotting accuracy's over $n = 10$ runs with random picked training/testing data:



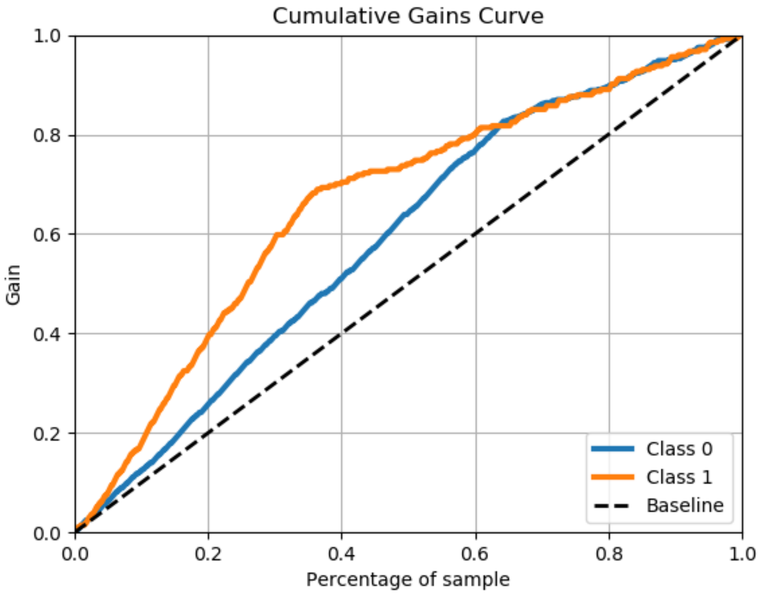




We see that the accuracy variance is very low on training predictions and a bit high for testing predictions.
We get the best average test prediction at 77.9% with split = 0.5. I.e. choosing the prediction with the highest probability.

Note that we should introduce some other accuracy measures since 63.3 % of the people are credible. This means that Predicting all 0's gives an accuracy score of 63.3 % which is terrible.

Cumulative gains on test prediction:



Both classes are above the baseline, so we are atleast beating a random classifier.
Lets calculate false-positive and false-negative rates at the test data.

($\#\tilde{y}_i = 1$ when $y_i = 0$ divided by $\#y_i = 0$)
false-positive rate 17.3%:

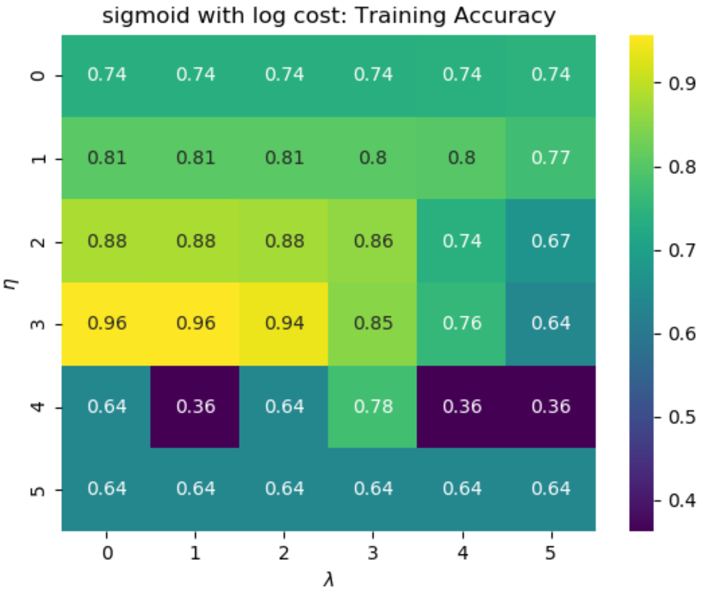
($\#\ \tilde{y}_i = 0$ when $y_i = 1$ divided by $\#y_i = 1$)
false-negative rate: 31.8%

This means that 31.8% of the non credible customers gets predicted as credible, which is pretty bad, so we stop the scoring computations here.

3.2 FFNN classification for credit predictions(Part c)

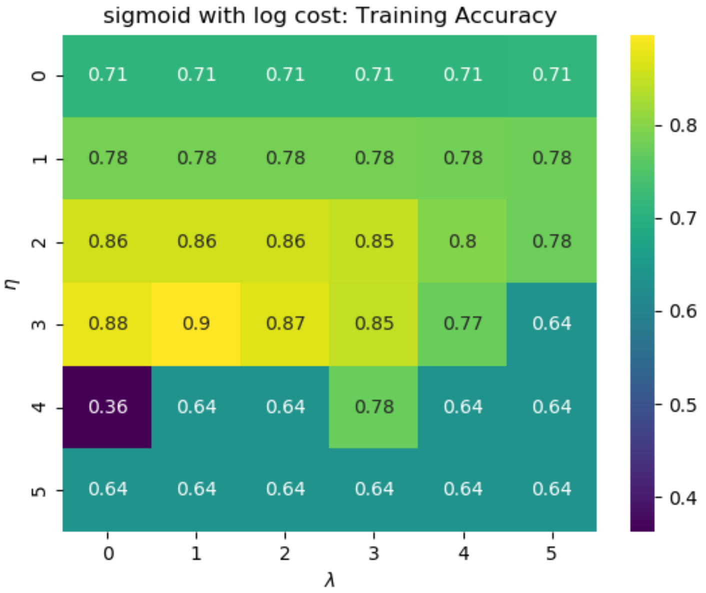
With fixed # hidden layers and neurons we run a grid search over learning rates $\eta = \{10^{-5}, ..., 10^{0}\}$ and regularization parameters $\lambda = \{10^{-5}, ..., 10^{0}\}$. Note that running tanh or relu with log cost is highly unstable as division by zero occurs frequently because of denominator in log-cost derivative. We combine tanh and relu with square cost and sigmoid with log cost, even though log cost is much preferable for probability based predictions.

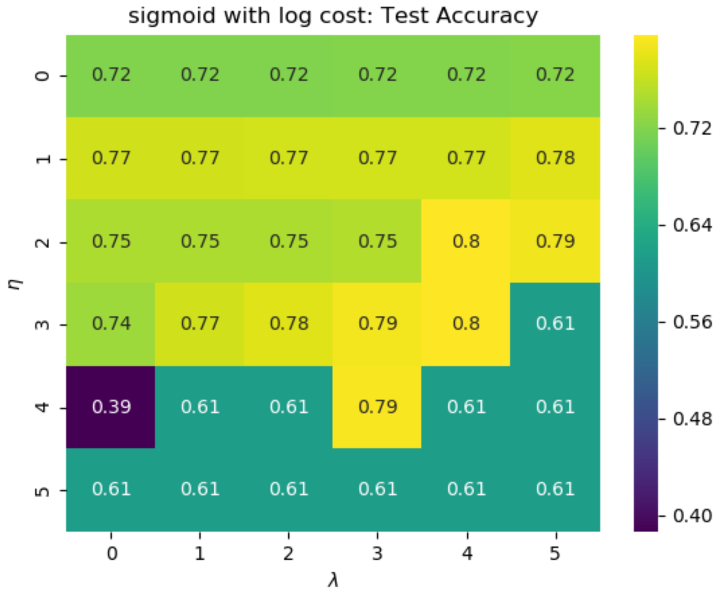
Using 5 hidden layers with 100 hidden neurons over 200 epochs leads to an overfit:
(Yellow training-scores at cost of bad testing-scores)



We try some different combinations of layers and neurons over 100 epochs and pick out the best results.

(4 hidden /w 100 neurons)



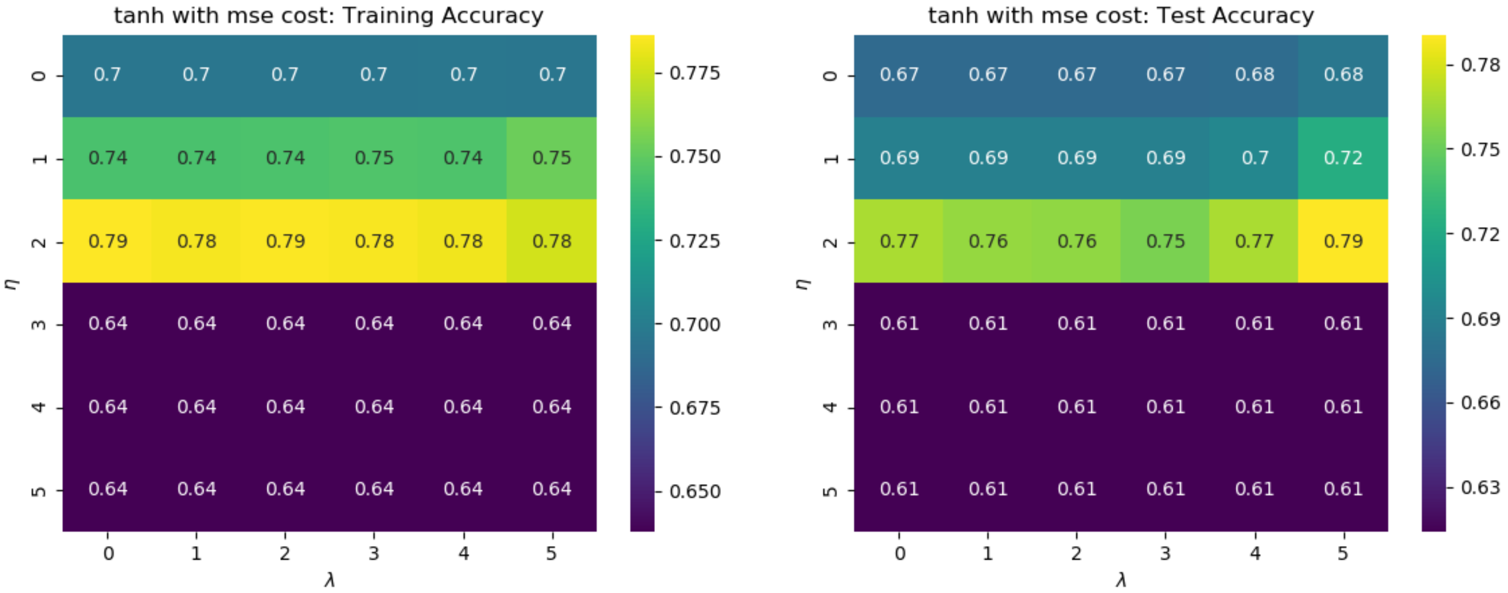


In both plots we see only one bad purple square and constant blue-ish values along λ axis in the bottom. We can interpret this as high numerical instability at (0, 4) and local minima for $\lambda \geq 10^{-1}$. $(4, 2)$ seems to be the best with $\eta = 10^{-3}$, $\lambda = 10^{-1}$.
Both train- and test-accuracy are 80%. We also calculate false-negative rate at 0.239 and false-positive rate at 0.182.

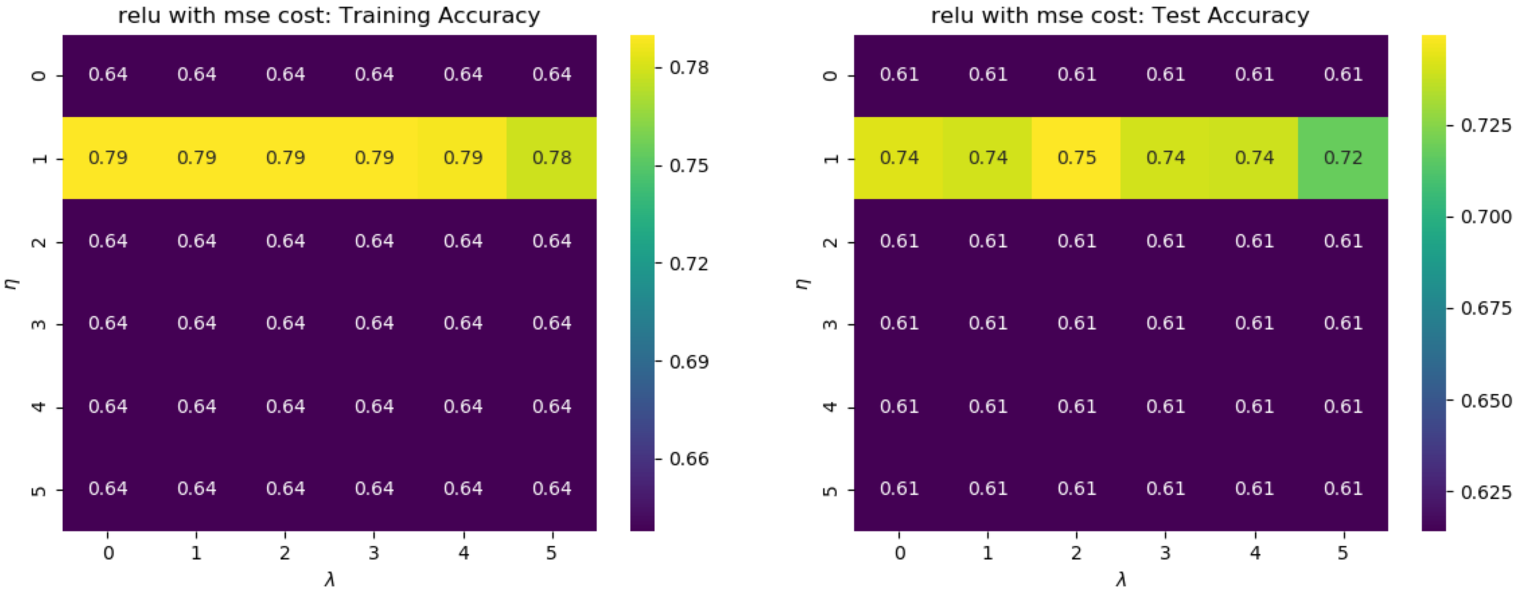
23.9% of non-credible customer gets predicted as credible, which is still bad, but better than the logistic regression (prev. section). Seeing as the error rate is that high we can already say the model is not good.

Doing the same for tanh and relu activation (figures below), we find worse training and testing accuracy's. Their false-negative rates are respectively 0.276 and 0.440. Tanh performed close to sigmoid, but relu results were a bit disappointing as it is common these days. I suspect that a bounded version of it would do better, but keep in mind that using square error is usually a bad idea for probabilistic classification.

(2 hidden /w 50 neurons)



(2 hidden /w 50 neurons)



In these plots the majority of learning rates makes the cost stuck in a local minima and for the other grid points, they are worse than using sigmoid with cross-entropy.

3.3 Continuous regression on Franke function using FFNN (Part d)

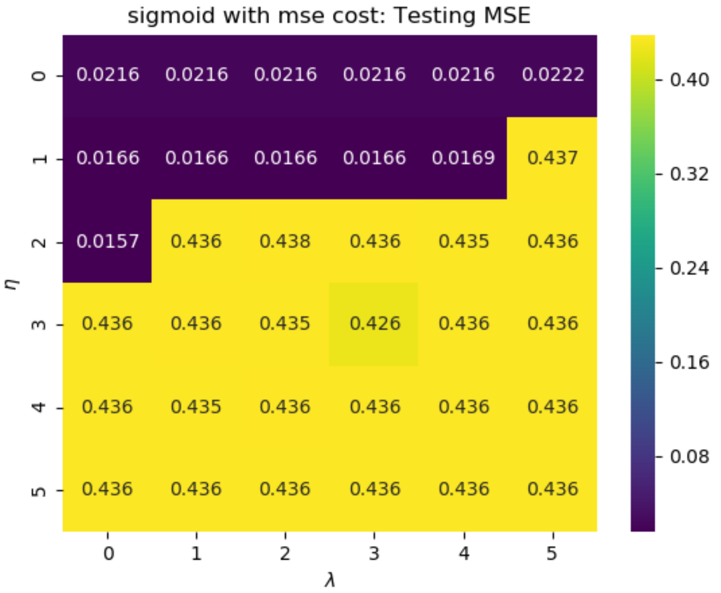
We train on sub-sets of [50x50] noised Franke data attempting to reproduce the true [50x50] Franke data using prediction on outside test data.

To recap from Project 1 (Report link in references) we got the best true test data prediction at $MSE = 1.081 \cdot 10^{-2}$ with ridge regression. Let's try to beat it.

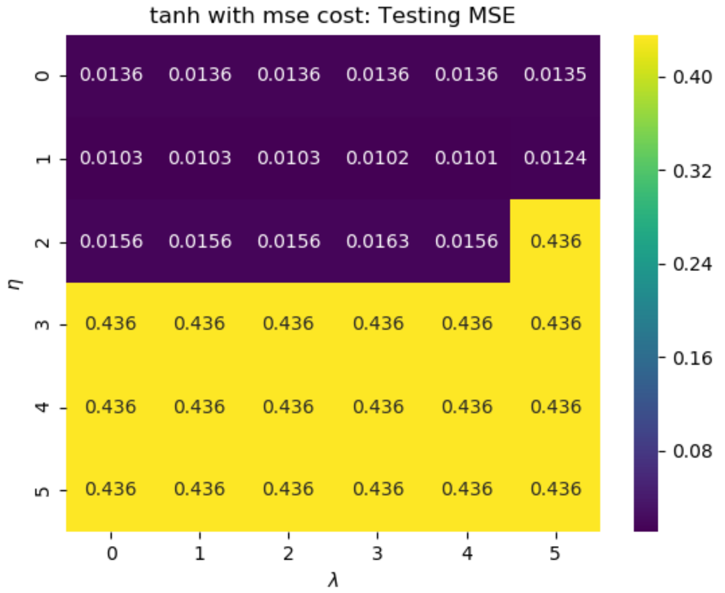
With different fixed # hidden layers and neurons, we run a grid search over learning rates $\eta = \{10^{-5}, ..., 10^{0}\}$ and regularization parameters $\lambda = \{10^{-5}, ..., 10^{0}\}$ (*See Part_d.py*).

Best grids seems to be these two:

(4 hidden /w 100 neurons, 200 epochs) (2 hidden /w 50 neurons, 200 epochs)



η	0	1	2	3	4	5
0	0.0216	0.0216	0.0216	0.0216	0.0216	0.0222
1	0.0166	0.0166	0.0166	0.0166	0.0169	0.437
2	0.0157	0.436	0.438	0.436	0.435	0.436
3	0.436	0.436	0.435	0.426	0.436	0.436
4	0.436	0.435	0.436	0.436	0.436	0.436
5	0.436	0.436	0.436	0.436	0.436	0.436



η	0	1	2	3	4	5
0	0.0136	0.0136	0.0136	0.0136	0.0136	0.0135
1	0.0103	0.0103	0.0103	0.0102	0.0101	0.0124
2	0.0156	0.0156	0.0156	0.0163	0.0156	0.436
3	0.436	0.436	0.436	0.436	0.436	0.436
4	0.436	0.436	0.436	0.436	0.436	0.436
5	0.436	0.436	0.436	0.436	0.436	0.436

Notice that we have terrible all yellow results for $\eta \geq 10^{-3}$ (sigmoid) and $\eta \geq 10^{-2}$ (tanh) meaning that we have reached a local minima. From (4,1) in tanh we find the best $MSE = 1.010 \cdot 10^{-2}$ with $(\lambda, \eta) = (10^{-1}, 10^{-4})$, which beats all our previously linear regression methods, Lasso, OLS and Ridge from project 1:

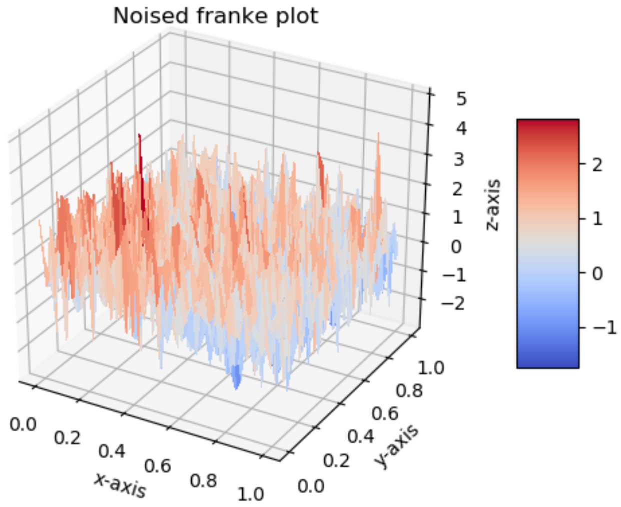
Lasso:
$\lambda = 10^{-3}$
$MSE(f_{test}, \tilde{z_{test}}) = 1.464 \cdot 10^{-2}$

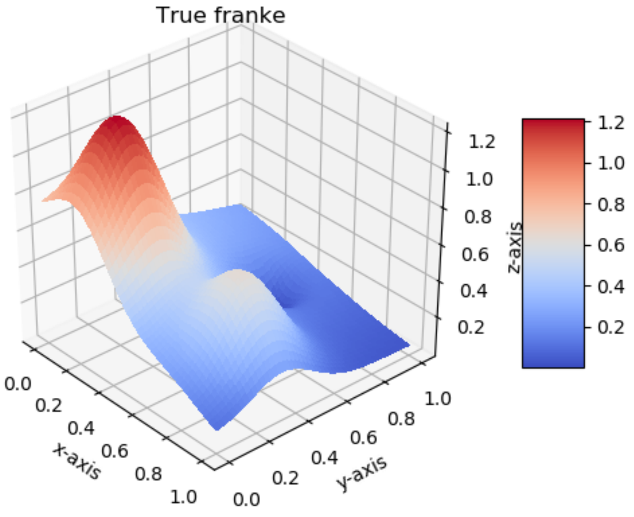
OLS:
$MSE(f_{test}, \tilde{z_{test}}) = 1.095 \cdot 10^{-2}$

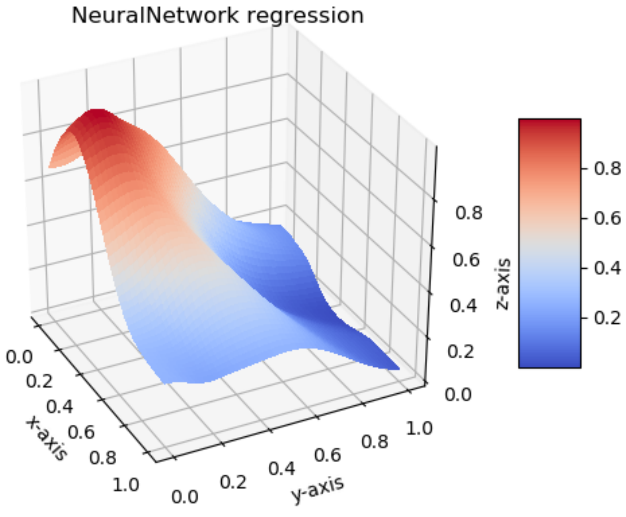
Ridge:
$\lambda = 10^{-3}$
$MSE(f_{test}, \tilde{z_{test}}) = 1.081 \cdot 10^{-2}$

NeuralNetwork:
$MSE(f_{test}, \tilde{z_{test}}) = 1.010 \cdot 10^{-2}$

Plot of franke /w $\mathcal{N}(0,1)$ noise used for training, true franke used for scoring, and our best regression:


Noised franke plot


True franke

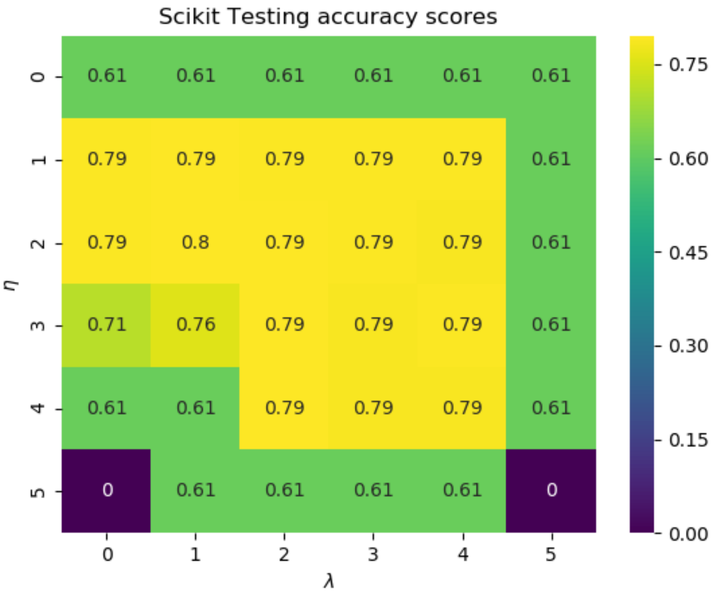

NeuralNetwork regression

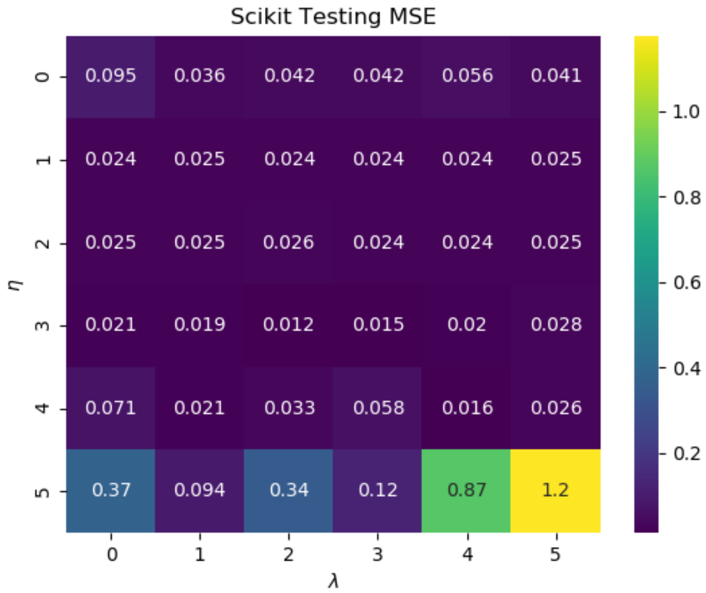
3.4 Testing of developed code

(See testing.py)
We develop a simple unit test for our self written NeuralNetwork class which passes with 100% accuracy. (NeuralNetwork_unit_test())

We also compare our best neural network results against the scikit library with similar network architecture. (scikit_classifier_test() and scikit_regressor_test)

Classification and regression with almost same network architecture:





We see that the grid is a bit different but the top scores acc=80% and MSE=0.012 are close to ours (80% and 0.0101), which is a good sign.

4 Summary and conclusions (Part e)

4.1 Binary logistic regression

For classification on our credit data we found the best average accuracy score, on test data over 10 runs, at 77.9%. This was with predicting positive or negative according to the greatest probability.

We also calculated the false-negative error rate at 31.8% on a test data sample which is to high to call our model a good classifier. We conclude that it is an descent indicator at best.

4.2 FFNN classification of credit data

We got the highest testing accuracy at 80% with an false-negative rate at 23.9%. This was with sigmoid as activation, 4 hidden layers with 100 neurons each, trained for 100 epochs.

This was better than logistic regression (78% with 32% rate), but still only good enough as an indication.

4.3 FFNN regression on Franke function

The best model we got on noised Franke regression was with regularization and learning rate $(\lambda, \eta) = (10^{-1}, 10^{-4})$ using tanh as activation, 2 hidden layers, and scaled $f(x) = x$ as output function. Comparison against previously used methods in project 1, from worst to best:
Lasso:
$\lambda = 10^{-3}$
$MSE(f_{test}, \tilde{z_{test}}) = 1.464 \cdot 10^{-2}$

OLS:
$MSE(f_{test}, \tilde{z_{test}}) = 1.095 \cdot 10^{-2}$

Ridge:
$\lambda = 10^{-3}$
$MSE(f_{test}, \tilde{z_{test}}) = 1.081 \cdot 10^{-2}$

NeuralNetwork:
$MSE(f_{test}, \tilde{z_{test}}) = 1.010 \cdot 10^{-2}$

4.4 Final remarks

Logistic regression is a simpler model, and such must faster, than multi-layer neural networks. It can be a powerful tool, especially for the binary case, but it's simplicity have major drawbacks for more complex/non-linear problems.

Neural networks can be a very powerful tool if tuned correctly, solving all sorts of linear or non-linear problems. E.g. self driving cars, image recognition, statistical research. It is super flexible and the choices of layers, activation functions, cost function and output function are endless.
Some cons of nn's are that they are very data hungry and needs big training sets, requires alot of hardware power and often needs alot of hyper-parameter tuning.

In the logistic regression we opted for Newtons method which has much better convergence rate than gradient descent but have the drawback of calculating the inverse hessian, which is costly and might be singular.
For back-propagation in the neural network we used stochastic gradient descent with mini-batches which is usually more efficient than regular gradient descent.
Both optimization tools can diverge, or converge only to local minima, or simply require to many steps for convergence.

5 References

1. Credit data research: `https://bradzzz.gitbooks.io/ga-seattle-dsi/content/dsi/dsi_05_classification_databases/1-lesson/assets/datasets/DefaultCreditCardClients_yeh_2009.pdf`

2. Credit data: `https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients`

3. Logistic regression lecture notes: `https://compphysics.github.io/MachineLearning/doc/pub/LogReg/html/LogReg.html`

4. Neural network lecture notes: `https://compphysics.github.io/MachineLearning/doc/pub/NeuralNet/html/._NeuralNet-bs000.html`

5. Additional nn notes: `http://neuralnetworksanddeeplearning.com/chap2.html`

6. Project 1: `https://github.com/ErlendAbrahamsen/FYS-STK3155/blob/master/Project1`