UiO : Department of Physics
University of Oslo

PROJECT 2
Neural Networks and Logistic Regression

FYS-STK 4155

Sagal Mohammed and Akuzike Banda

13th November 2019

1 Abstract

We studied two important machine learning algorithms: Neural network and Logistic Regression. The models were applied to a regression problem: Franke's function and a classification problem: Breast Cancer Data set. We studied the performance of the algorithms on the two data sets by looking at train , test accuracy scores, R2-score and MSE. Parameters of the algorithms were varied in different cases and combinations quest to ftnd the optimal parameter values and ideal algorithm for each dataset. The performance of the algorithms we built was compared to Scikit learn's neural network and logistic regression. Neural network proved to be a better model to ftt our classiftcation data set as compared to logistic regression. For the regression dataset, for the activation functions used, fttting a simple linear regression would be the best option rather than a neural network or logistic regression model.

Contents

1 Introduction

Machine learning algorithms are very important for our modern society. They are used by huge companies like Google and Facebook, in the medical field, education and e.t.c. In this project we have studied to important machine learning algorithms;Neural Network and logistic Regression. We have used the methods to study a cancer data, and a Franke's function. We studied their performance by looking at accuracy score, R2-score and MSE. At last we compared it to how our own built in code performed with respect to a professional package like Scikit learn. The project consists of an introduction, Result and implementation, discussion and a conclusion part. We shall continue with the introduction by going on with useful theory.

2 Theory

2.1 Logistic Regression

In the world of statistics, we may be interested to find out whether an individual will default on their credit card payment based annual income, monthly credit card balance and sex of the individual. Only one of two outcomes can be realised at the end: The individual defaults on the credit card payment or he does not. This kind of problem which is concerned with the response variable taking a discrete form is regarded as a classification problem. Logistic regression is a statistical model used to solve classification problems which tend to have qualitative outputs. Linear regression, on the other hand, is used to make a continuous outcome. The most common situation where logistic regression is applied are those with two possible outcomes, referred to as binary outcomes. From the credit card data example, the values balance, sex and income are known as predictors X_i's. The outcome: whether the credit card defaults or not, is Y, with values y_i=0 for no and y_i=1 for yes. This goal is to use this set of observations$(x_1,y_1),\ldots(x_n,y_n)$ to build a classification model that predicts Y from the design matrix $\tilde{X} \in \mathbb{R}^{n*p}$ with n samples and p predictors. The prediction is modelled using a threshold function that gives a binary output either 0 and 1. A Sigmoid function represents the probability for for a given event,

$$p(t) = \frac{exp(t)}{1 + exp(t)} \tag{1}$$

The model in equation 1 is fit by a method called maximum likelihood using conditional probability. The logistic function always produces an *S-shaped* curve and saturates when its argument is very positive or very negative, ensuring that we always get a sensible prediction.

As in Linear regression, the accuracy of the coefficient estimates can be measured by computing their standard errors. To define all the predictors and likelihood of all outcomes in Y from the training dataset $D = (y_i, x_i$ Maximum Likelihood estimation (MLE) principle is used. MLE is a set or parameters that maximize the log likelihood function by choosing β values that will maximize the function. The aim is to maximize the probability of seeing the observed data. The likelihood function can be written as

$$P(D|\beta) = \Pi_{i=1}^{n}[p(y_i = 1|x_i, \beta)]^{y_i}[1 - p(y_i = 1|x_i, \beta)]^{1-y_i} \tag{2}$$

From equation 2 we obtain the log likelihood and our cost/ loss function. To maximize the log likelihood function is equal to minimize the cost function. After reordering logarithms, the cost/ loss function can be written as

$$C(\beta) = \sum_{i-1}^{n}(y_i(\beta_0 + \beta_1 x_i) - log(1 + exp(\beta_0 + \beta_1 x_i))). \tag{3}$$

This equation is known as cross entropy.If the cross entropy is a convex function of the weights β, then any local minimizer is a global minimizer. Minimizing the function in 3 with respect to β_0 and β_1 we obtain

$$\frac{\partial C(\beta)}{\partial \beta_0} = -\sum_{i=1}^{n}(y_i - \frac{exp(\beta_0 + \beta_1 x_i)}{1 + exp(\beta_0 + \beta_1 x_i)}) \tag{4}$$

$$\frac{\partial C(\beta)}{\partial \beta_1} = -\sum_{i=1}^{n}(y_i x_i - x_i \frac{exp(\beta_0 + \beta_1 x_i)}{1 + exp(\beta_0 + \beta_1 x_i)}) \tag{5}$$

From which we can obtain the following

$$\frac{\partial C(\beta)}{\partial \beta} = -X^T(y - p) \tag{6}$$

and the second derivative

$$\frac{\partial^2 C(\beta)}{\partial \beta \partial \beta^T} = X^T W X \tag{7}$$

2.2 Gradient Descent

Gradient Descent (GD) is an iterative optimization algorithm to find the best fit line for a given training set. This best line has values of parameters that minimize the cost function. The cost function measures how good our hypothesis is doing on a single pair. x_i, y_i. GD is based on the observation that a function $F(x)$ decreases fastest

if one goes from x in the direction of the negative gradient $-\Delta F(x)$. We start with an initial guess x_0 for a minimum of F and compute new approximations according to

$$x_{k+1} = x_k - \gamma_k \nabla F(x_k), k \geq 0. \tag{8}$$

The parameter γ_k is the learning rate. This parameter is a very small flexible number which controls how the function changes as we go down the curve. Larger learning rates lead to the algorithm taking huge steps down the slope while smaller learning rates lead to taking smaller steps. The step size for each feature is calculated $\gamma_k \nabla F(x_k)$. As we get closer to the minimum, the step size decreases to avoid missing the minimum.

In situations where F is a convex function,all local minima are also global minima. When F is non-convex, high dimensional cost functions with many local minima are encountered. With GD, even though using the entire training set to calculate GD is useful to obtain the minima in a less noisy manner, computing the derivative term gets more expensive as the training set gets larger. GD is in addition very sensitive to initial conditions, for instance, the choice of the learning rate. Stochastic Gradient Descent (SGD) alleviates some of the problems encountered in GD by introducing randomness.

2.2.1 Stochastic Gradient Descent

As the name suggests, Stochastic gradient descent (SGD) involves selecting a few samples at random instead of using the entire training data. SGD comes from the observation that the cost function can be written as a sum over n data points $x_{i i=1}^{n}$,

$$C(\beta) = \sum_{i=1}^{n} c_i(x_i, \beta). \tag{9}$$

This in turn means that the gradient can be computed as a sum over i-gradients

$$\nabla_\beta C(\beta) = \sum_{i}^{n} \nabla_\beta c_i(xi, \beta) \tag{10}$$

The gradient of a subset of the data, known as minibatch, is taken thereby introducing randomness. If there are n data points and the size of each minibatch is M, there will be $n \div M$ minibatches. A gradient descent step now looks like

$$\beta_{j+1} = \beta_j - \gamma_j \sum^{n} -i \in \beta_k \nabla_\beta c_i(x_i, \beta) \tag{11}$$

Where k is picked at random with equal probability from $[1, n/M]$. An iteration over the number of minibatches (n/M) is commonly referred to as an epoch. Typically, a number of epochs is chosen and for each epoch iterate over a number of minibatches. The first step when carrying out SGD involves randomly shuffling the dataset. This is followed by picking mini batches of a particular size and calculating the stochastic gradient of each batch. Every iteration is faster.Collectively, SGD moves into direction of global minimum but every iteration may not go in this direction. It does not converge to the global minimum: As we approach the minimum, it gets very close to the global minimum.

There are two possibilities to know where to stop when searching for a minimum. One approach involves computing the full gradient after a given number of epochs and checking if the norm of the gradient is smaller than some threshold and stop id true. A second possibility lets the step length γ_i depend on number of epochs to make the smaller length smaller with time. If $e = 0, 1, 2, 3, ...$ denote the current epoch, $t_0, t_1 > 0$ are two fixed numbers and $t = e\dot{m} + i$ where m is number of minibatches and $i = 0, 1, 2, ..., m - 1$. Then the function

$$\gamma_i(t; t_0, t_1) = \frac{t_0}{t + t_1} \tag{12}$$

goes to zero as the number of epochs gets large. The number of epochs an be fixed to compute β and evaluate the cost function at the end. The final β that gives the lowest value of the cost function is picked

2.3 Neural Networks

Before diving into Neural networks, we first describe a perceptron. Briefly, this is a type of an artificial neuron. An artificial neuron is a connection point in a neural network. A perceptron accumulates incoming inputs which must exceed an activation threshold to yield an output. The behaviour of a neuron is explained by the model below.

$$y = f(\sum_{i=1}^{n} w_i x_i) = f(u) \tag{13}$$

The output y is the value of its activation function, which have as input a weighted sum of signals $x_i, ...x_n$ received by n other neurons. Each input signal has a corresponding weight $w_i, ...w_n$ which expresses the significance of the input to the output and a small change in the weight $b_1, ..., b_n$ known as the bias. A binary output, for instance, is

determined by whether the weighted sum $\sum_j w_j x_j + b_j$ is less than or greater than some threshold value.

$$output = \begin{cases} 0, & \text{if } \sum_j w_j x_j + b_J \leq \text{threshold.} \\ 1, & \text{if } \sum_j w_j x_j + b_j > \text{threshold.} \end{cases} \tag{14}$$

An artificial neural network (ANN) is a computational that mimics the basic idea of a human brain where billions of connected neurons in multiple layers communicate by sending electronic signals. Neural networks are neural-inspired nonlinear models for supervised learning. The simplest type of ANN is a feed-forward neural network (FFNN) where information only moves in a single direction, forward, through the layers.

The basic architecture of a neural network contains three layers: Input, hidden and output layers. The network can have a single hidden layer or several. [width=3cm, height=4cm]ANN.png

2.3.1 Multi Layer Perceptron

Fully-connected FFNN three or more layers consisting of neurons with non-linear activation functions are known as multilayer perceptrons (MLPs). MLP is a series of logistic regression models stacked, with the final layer being another logistic or linear regression model.It is basically ana analytical function mapping real-valued vectors $\vec{x} \in \mathbb{R}^n \rightarrow \vec{y} \in \mathbb{R}^m$ An activation function or learning algorithm is applied to neurons in an MLP to fine tune it. One such activation function is the sigmoid function. OUr MLPs make use of sigmoid neurons. Sigmoid neurons are similar to perceptrons but have the sigmoid function applied to the output on each neuron to aid in learning. The sigmoid function defined by

$$\sigma(z) = \frac{1}{1 + \exp^{-z}} \tag{15}$$

is applied to the output of a sigmoid neuron, $z = \sum_{j=1}^n w_j x_j + b_j$ that was defined earlier. By using the σ function, we get a smoothed out perceptron. The smoothness of σ means that small changes in the weights w_j and small changes in the bias b_j will produce a small change to the output of the neuron.

First, for each neuron i in the first hidden layer we calculate a weighted sum z_i^1 of the input coordinates x_j

$$z_i^1 = \sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1 \tag{16}$$

Here b_i is the bias, z_i^1 is the argument for our sigmoid activation function f_i of each node i and M stands for all possible inputs to a given node i in the first layer. For each layer, assuming that the activation function is the same, for the l-th layer,

$$y_i^l = f^l(u_i^l) = f^l(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^l + b_i^l) \tag{17}$$

where N_l is the number of nodes in layer l. When the output of all the nodes in the first hidden layer are computed, the values of the subsequent layer can be calculated and so forth until the output is obtained. This illustrates that MLP is essentially a nested sum of scaled activation functions of the form $f(x) = c_1 f(c_2 x + c_3) + c_4$ where c_i's are weights and biases.

The biases and activation functions can be represented as layer-wise column vectors $\vec{b_l}$ and $\vec{y_l}$ so that the i-th element of each vector is the bias b_i^l and the activation function y_i^l of node i in layer l

2.3.2 Back Propagation

Back propagation is the practice of fine-tuning the weights w_i of a NN based on the error rate obtained from the previous epoch.The error rate from the previous epoch was obtained with the help of gradient descent as our optimization function. At the core of back propagation is an expression for the partial derivatives of the cost function C with respect to weight w and bias b $\partial C/\partial w$ and $\partial C/\partial b$ which tells us how quickly the cost changes when we change the weights and biases. Back propagation provides a way of computing the gradient of the cost function.

From the discussion in the previous subsection, we had 17, the activation function of l-th layer given node i in the first layer. To rewrite the equation in matrix form, we define a weight matrix w^l for each layer, l. For each layer l we define a bias vector b^l. The components of the bias vector are just b_j^l values for each neuron in the l-th layer. We define an activation function a^l whose components are the activations a_j^l. The equation can be written in a compact vectorized form as:

$$a^l = \sigma(w^l a^{l-1} + b^l) \tag{18}$$

Which is sometimes written in terms of the weighted input as $a^l = \sigma(z^l)$ with $z^l = w^l a^{l-1} + b^l$. We first define our cost function for a single training example, assuming that x is fixed, as $C_x = \frac{1}{2}||y - a^L||^2$ where a^L is the vector of activations when x is input and x is an

individual training example and $y = y(x)$ is the desired output.We make an assumption that

$$C = \frac{1}{2}||y - a^L||^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2, \tag{19}$$

Is a function of the output activations. To understand how the weights and biases change with respect to the cost function we compute partial derivatives $\partial C/\partial w_{jk}^l$ and $\partial C/\partial b_j^i$. We then introduce an intermediate quantity, δ_j^l, which is the error rate in the j-th neuron in the l-th hidden layer. The error rate adds a little amount Δz_j^l to the neuron's input causing the nueron to output $\sigma(z_j^l + \Delta z_j^l)$. This change propagates through later layers in the network and finally causes the overall cost to change by an amount $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$. The algorithm tries to improve the cost by finding a Δz_j^l which makes the cost smaller, for instance, by choosing Δz_j^l to have the opposite sign to $\frac{\partial C}{\partial z_j^l}$. It is with this regard that the error δ_j^l for neuron j in layer l is defined by

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l} \tag{20}$$

The error rate for the output layer δ_j^L for a particular output neuron j is given by

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \tag{21}$$

The first term on the right measures how fast the cost is changing as a function of the j-th output activation. The second term on the right measures how fast the activation function σ is changing at z_j^L. 21 can be expressed in a matrix based form as $\delta^L = \Delta^a C \odot \sigma'(z^L)$. For the hidden layers, error rate is expressed in regards to the next layer since we back propagate the error. For each hidden layer $l = L-1, L-2, ..., 2$ the error rate δ^l is calculated as

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f\prime(z_j^l) \tag{22}$$

This moves the error backward through the activation function in layer l to give the error δ^l in the weighted input to layer l. By combining 21 and 22 the error δ^l for any layer in the network can be computed. The error rate for the output layer is applied first, then applying the error for the hidden layers to back propagate from hidden layer $L-2$ to layer 2 of the network. An equation for the rate of

change of the cost with respect to any bias in the network is expressed as:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{23}$$

The rate of change of the cost function with respect to any bias is equal to the error rate at the same neuron as the bias. With the definition of δ_j^l, a more compact definition of the derivative of the cost function in terms of the weights can be derived. The equation of the rate of change of the cost with respect to any weight in the network is expressed as:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{24}$$

The rate of change can be understood as a product of the activation of the neuron input to the weight w_i and error of the neuron output from the weight w. This is represented as $\frac{\partial C}{\partial} = a_{in}\delta_{out}$

The four equations output error: 21, back propagate error for hidden layers:22, rate of change of cost with respect to bias:23 and rate of change of cost with respect to weight: 24 provide us with a way of computing the gradient of the cost function. They are useful when setting up the back propagation algorithm.

2.4 Activation Functions

Activation functions are equations used to determine the output of hidden layers of a classification problem: logistic regression and neural network. They are used in neural networks to calculate the back propagated error that An activation function is considered as a step function that assigns a value to the the outgoing neuron in the next layer based on the threshold that is applied or a transformation that maps input signals to output signals. They can be linear or non-linear. Nonlinear activation functions are the most used. The derivatives of the nonlinear activation functions are convenient for efficiently calculating gradients in neural networks.These use the derivative of the function to represent the slope or change in y-axis with respect to change in the x-axis. The derivative is used to know in which direction and how much to change or update the curve depending on the slope. Below are some of the commonly used activation functions.

2.4.1 Identity Activation Function

This is a linear function that maps the pre-activation to itself and outputs are not confined in any range but vary from$(-\infty, \infty)$. It is not possible to use gradient descent to train a model that uses this function. This due to the problem that the derivative of the function is a constant and therefore has no relation to the input X. This is a limitation when trying to improve weights in a neural network through back propagation.

2.4.2 Sigmoid Activation Function

This nonlinear function is bounded: It returns values between 0 and 1.The logistic /sigmoid function has the form:

$$sigmoid(z) = \frac{1}{1 + \exp^{-z}} \tag{25}$$

2.4.3 Tanh

The hyperbolic tangent (tanh) function has output values in the range $(-1, 1)$. Hence strongly negative inputs to the tanh will map to negative outputs. Tanh has been said to work better in practice because it is not limited to only positive outputs in the hidden layers. Tanh function takes the form:

$$tanh(z) = \frac{sinh(z)}{cosh(z)} \tag{26}$$

2.4.4 ReLU Activation Function

Rectified Linear Unit (ReLU) is a nonlinear function whose outputs range from $[0, \infty$ meaning it can blow up the activation.With ReLU, fewer neurons are fired and the network is lighter for big neural networks with a lot of neurons.ReLU takes the form:

$$R(z) = max(0, z) \tag{27}$$

2.5 Dimensionality Reduction

Many machine learning problems often contain a great number of columns of features for training, which makes training slow and harder to find a good solution. This problem is referred to as dimensionality. Data is considered to be high-dimensional if it has thousands or millions of variables or features lying near or on a low dimensional subspace. Although high-power computing can handle high-dimensional data, it is still necessary to compress or reduce the dimensionality of the original data while preserving the richness of the data. It is essential to reduce the dimensionality so as to discover hidden correlations; remove redundant, noisy features; interpretation and easier processing of the data. Several techniques for dimensionality reduction are available, for instance: the principal component analysis (PCA), Kernel PCA, and Locally Linear Embedding (LLE).

2.5.1 Data Preprocessing

Preprocessing can be done Feature scaling or mean normalization.For mean normalization, we replace each x_j^i with $x_j^i - \mu^j$. If different features on different scales, scale features to have comparable range of values, that is $x_j^i \leftarrow \frac{x_j^i - \mu^j}{s_j}$ where s_j is the standard deviation for feature j. This is feature scaling.

2.5.2 Principal Component Analysis (PCA)

PCA algorithm tries to find intrinsic variables that still approximately describe the data from the extrisic (measurement) variables. PCA low dimensional subspace on which to project the data to minimize the squared projection errors. We need to find a vector $\vec{u}$ on which to project the surface of our data. To reduce dimensionality of the data from $n-$ dimensions to $k-$ dimensions using PCA we compute the covariance (covariance) matrix Σ . Then compute eigenvectors of the covariance matrix using single value decomposition (SVD) of the matrix $[U, S, V] = svd(\Sigma)$. Out of the outputs of SVD, what is needed is U where $U \in \mathbb{R}^{n\times n}$ with $n =$ all the dimensions of the data.

The $explained\_variance\_ratio\_$ tells how much variance can be attributed to each of the principal components.

3 Data

3.1 Franke Function or Terrain Data

The Franke's function was chosen as the set for regression analysis. Franke function is a 2-dimensional function which has been widely use when testing various interpolation and fitting algorithms. The function is a weighted sum of four exponentials read as follows.

$$f(x, y) = \frac{3}{4}exp(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}) + \frac{3}{4}exp(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}) + \frac{1}{2}exp-(\frac{(9x7)^2}{4} - \frac{(9y-3)^2}{4}) - \frac{1}{5}ex$$

(28)

The function was defined for $x, y \in [0, 1]$. The function was defined for polynomial degree 5.

3.2 Breast Cancer Data Set

The Breast Cancer data set is a classic and easy binary classification data set available from Scikit learn library. The set has 569 total samples and a dimensionality of 30. Features describe characteristics of the cell nuclei present in the digitized image of the breast mass. The dimensionality stems from the relevant features that were selected through an exhaustive search. The binary outputs, targets, of the set include: Benign, if the tumor identified in the breast is not harmful and malignant if the tumor identified is cancerous. The data set was created at University of Wisconsin, Clinical Sciences Center.

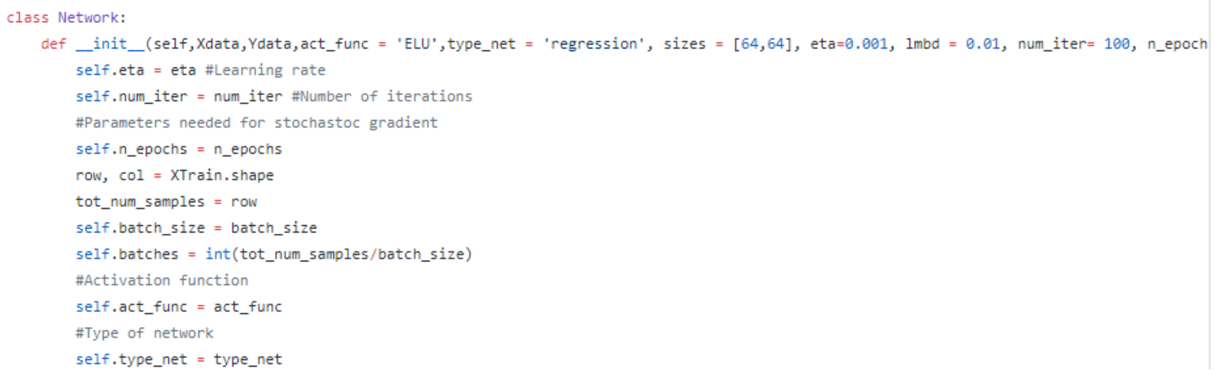
5 Implementation and Results

The report consists of two parts; a regression part, and a classification part. Since the measurement methods are not correlated we will present each part separately.

Emplementation of own code

Neural Network

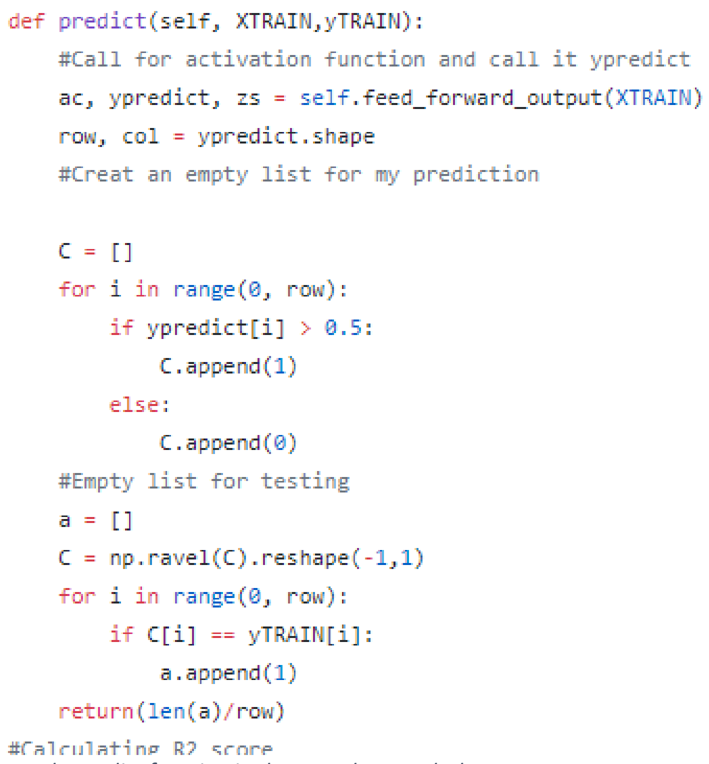
A class was created where the Neural Network was defined with its functions and variables. Some default arguments were made available to be used when constructing an object of the class without specifying all arguments as shown in figure a. The class was put in a separate file *NN_class.py* and was imported into all files that needed to make use of the network class.



```
class Network:
    def __init__(self,Xdata,Ydata,act_func = 'ELU',type_net = 'regression', sizes = [64,64], eta=0.001, lmbd = 0.01, num_iter= 100, n_epoch
        self.eta = eta #Learning rate
        self.num_iter = num_iter #Number of iterations
        #Parameters needed for stochastoc gradient
        self.n_epochs = n_epochs
        row, col = XTrain.shape
        tot_num_samples = row
        self.batch_size = batch_size
        self.batches = int(tot_num_samples/batch_size)
        #Activation function
        self.act_func = act_func
        #Type of network
        self.type_net = type_net
```

Figure a: Part of the Definition of the Network Class showing some arguments, parameters and the init function

Some of the functions that were defined in the class are: different activation functions and their derivatives, feed forward training, back propagation using gradient descent or stochastic gradient descent, feed foward outputs and predictions. The default activation function was set as sigmoid through an *esleif statement.* Figure b below shows how the prediction was calculated using the backpropagated values obtained through feed forward training.



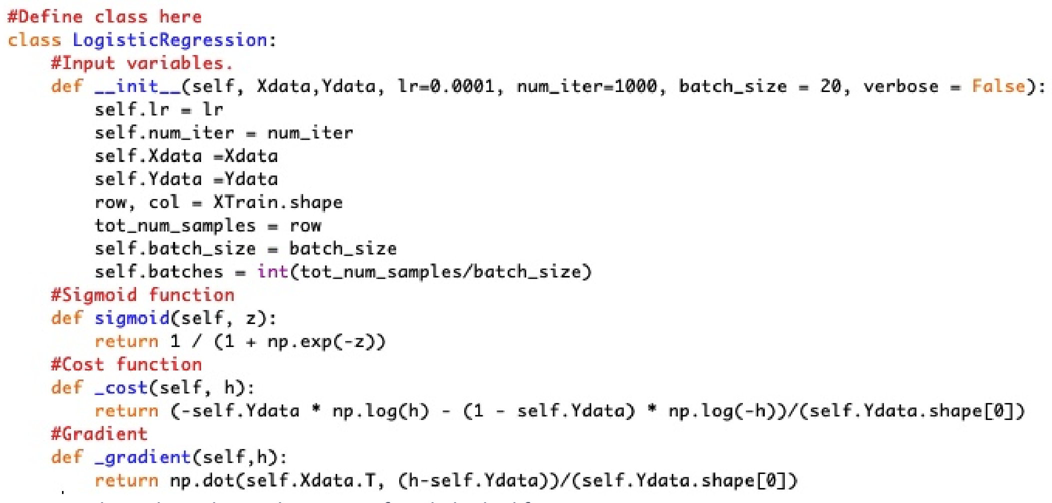
```
def predict(self, XTRAIN,yTRAIN):
    #Call for activation function and call it ypredict
    ac, ypredict, zs = self.feed_forward_output(XTRAIN)
    row, col = ypredict.shape
    #Creat an empty list for my prediction

    C = []
    for i in range(0, row):
        if ypredict[i] > 0.5:
            C.append(1)
        else:
            C.append(0)
    #Empty list for testing
    a = []
    C = np.ravel(C).reshape(-1,1)
    for i in range(0, row):
        if C[i] == yTRAIN[i]:
            a.append(1)
    return(len(a)/row)
#Calculating R2 score
```

Figure b: Predict function in the neural network class

Logistic Regression

We started by making a class that contained a code that could do a Logistic Regression. The code can be found at the map called logistic regression in code called logRegClass.py. We defined inputs variables in the class as follows; *Xdata*, *Ydata*, *lr* referring to learning rate, *num_iter* representing number of iterations and *batch size*. Since the inputs could change it was important to make them passable and flexible when we wanted to switch them. We then went on to define a sigmoid, cost and gradient functions as expressed respectively in equations (????),(????)and (????). In the class we added a *self* parameter to the functions so that they could be callable when we needed them. Figure below illustrates the view of the code up to this point:-



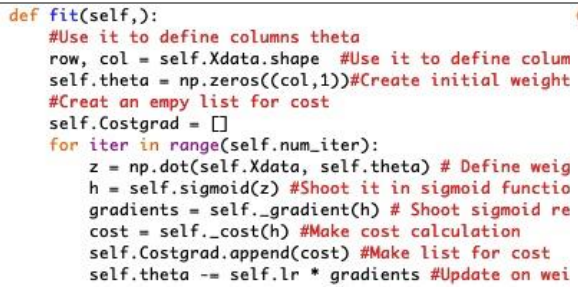
```
#Define class here
class LogisticRegression:
    #Input variables.
    def __init__(self, Xdata,Ydata, lr=0.0001, num_iter=1000, batch_size = 20, verbose = False):
        self.lr = lr
        self.num_iter = num_iter
        self.Xdata =Xdata
        self.Ydata =Ydata
        row, col = XTrain.shape
        tot_num_samples = row
        self.batch_size = batch_size
        self.batches = int(tot_num_samples/batch_size)
    #Sigmoid function
    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))
    #Cost function
    def _cost(self, h):
        return (-self.Ydata * np.log(h) - (1 - self.Ydata) * np.log(-h))/(self.Ydata.shape[0])
    #Gradient
    def _gradient(self,h):
        return np.dot(self.Xdata.T, (h-self.Ydata))/(self.Ydata.shape[0])
```

Figure c:Shows how the implentation of code looked for part 1.

The next thing was to implement the gradient descent, and stochastic gradient descent methods. As explained in the theory, these are optimization parameters that minimize the cost function with respect to different parameters, in our case β. For the gradient descent we started by defining an empty array for β as shown in figure (2) below. We then made a loop that ran through the number of iterations. Under the loop a weighted sum expressed by equation(???) was definied. Also a sigmoid as function of the weighted sum, and an update on the weight (self.theta -= self.lr * gradients) included. Figure(2) below gives a view of how the code looked.

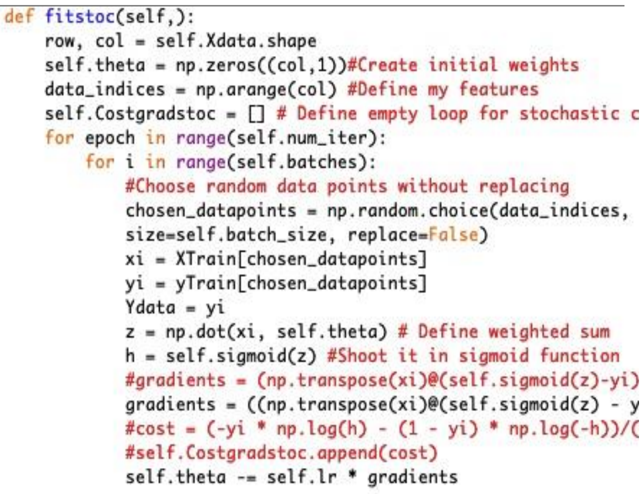
For the stochastic gradient descent, the same applications in gradient descent was applied. The difference now was that we defined a loop that went through different batches. We also defined a line in the code(Chosen_datapoints) that left out a batch every time it was used. Also, only a certain batch was used to calculate the gradient each time. The full code is illustrated in picture (3) below.

Gradient descent code implementation	**Stochastic gradient descent code implementation**



```python
def fit(self,):
    #Use it to define columns theta
    row, col = self.Xdata.shape  #Use it to define colum
    self.theta = np.zeros((col,1))#Create initial weight
    #Creat an empy list for cost
    self.Costgrad = []
    for iter in range(self.num_iter):
        z = np.dot(self.Xdata, self.theta) # Define weig
        h = self.sigmoid(z) #Shoot it in sigmoid functio
        gradients = self._gradient(h) # Shoot sigmoid re
        cost = self._cost(h) #Make cost calculation
        self.Costgrad.append(cost) #Make list for cost
        self.theta -= self.lr * gradients #Update on wei
```

Figure d: Shows the code for gradient descent for logistic regression.



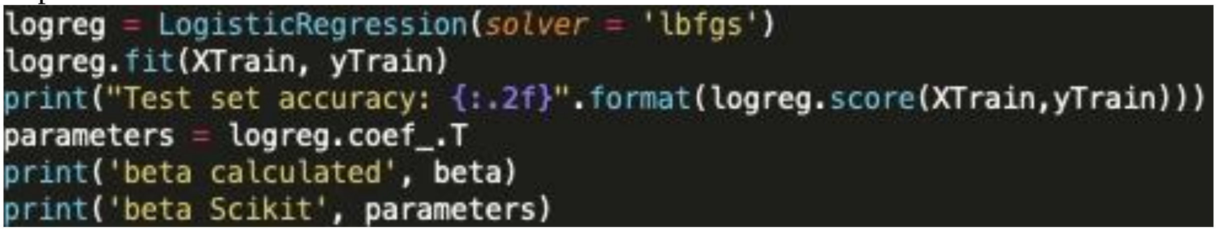
```python
def fitstoc(self,):
    row, col = self.Xdata.shape
    self.theta = np.zeros((col,1))#Create initial weights
    data_indices = np.arange(col) #Define my features
    self.Costgradstoc = [] # Define empty loop for stochastic c
    for epoch in range(self.num_iter):
        for i in range(self.batches):
            #Choose random data points without replacing
            chosen_datapoints = np.random.choice(data_indices,
            size=self.batch_size, replace=False)
            xi = XTrain[chosen_datapoints]
            yi = yTrain[chosen_datapoints]
            Ydata = yi
            z = np.dot(xi, self.theta) # Define weighted sum
            h = self.sigmoid(z) #Shoot it in sigmoid function
            #gradients = (np.transpose(xi)@(self.sigmoid(z)-yi)
            gradients = ((np.transpose(xi)@(self.sigmoid(z) - y
            #cost = (-yi * np.log(h) - (1 - yi) * np.log(-h))/(
            #self.Costgradstoc.append(cost)
            self.theta -= self.lr * gradients
```

Figure e: Shows code for stochastic gradient descent for logistic regression.

The last part of the class contained a function that had a threshold included. The most recent updated theta value that was inserted into the sigmoid function to compute the output, had not zeros and ones as results. So we made a threshold of 0.5 to fix that. All above 0.5 became ones, and all below became ones. So we compared the predicted outcome with the true values from the data, and calculated accuracy as explained in equation in the theory section .

Emplementation of Scikit Learn

To use the Scikit Learn package we had to import the following; `from sklearn.linear_model import LogisticRegression.` We could chose different solvers, and in our case we chose 'lbfgs' solver. Figure(3) below shows a simple implementation of that.



```python
logreg = LogisticRegression(solver = 'lbfgs')
logreg.fit(XTrain, yTrain)
print("Test set accuracy: {:.2f}".format(logreg.score(XTrain,yTrain)))
parameters = logreg.coef_.T
print('beta calculated', beta)
print('beta Scikit', parameters)
```

Figure f: Simple overview of how Scikit Learn package is used.

Results

5.0.1 Regression

Logistic Regression

In project we have studied and seen how the learning rate variated with different number of iteration. We calculated the accuracy score, and presented the results in a seaborn graph. The result can be seen in figure (6), and (7). As can be seen from figures, we get our best accuracy score for the test data learning rates equal 1.5, and 1 between iterations 40, and a 1000. If we look at figure (6) we see similar values for training accuracy in that region is 0.99 which is good. But, we also see that we get an accuracy equal 1 for higher values iterations around 3000, and 10000 for learning rates equal. But the test accuracy in that region drops to 0.96 and 0.95 indicating an overfitting.

We did also similar calculations for stochastic gradient descent but we opted for gradient descent since training accuracy for gradient descent showed it pipped the stochastic gradient descent by 1 percent. But, we would say using both methods is okay.

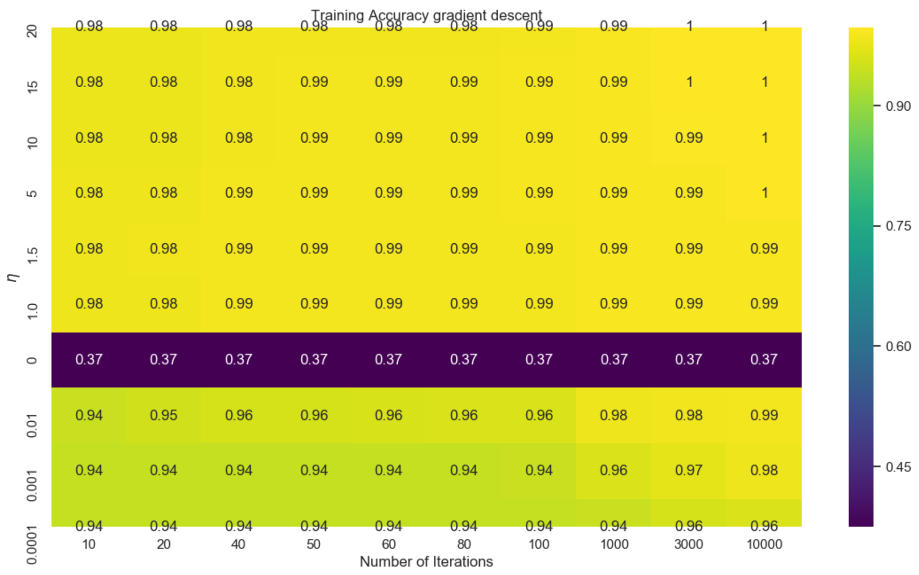


Figure g: Figure showing a seaborn plot of training accuracy varying with the number of iterations in x-axis, and the learning rate eta in y-axis. Optimization method used is gradient descent.

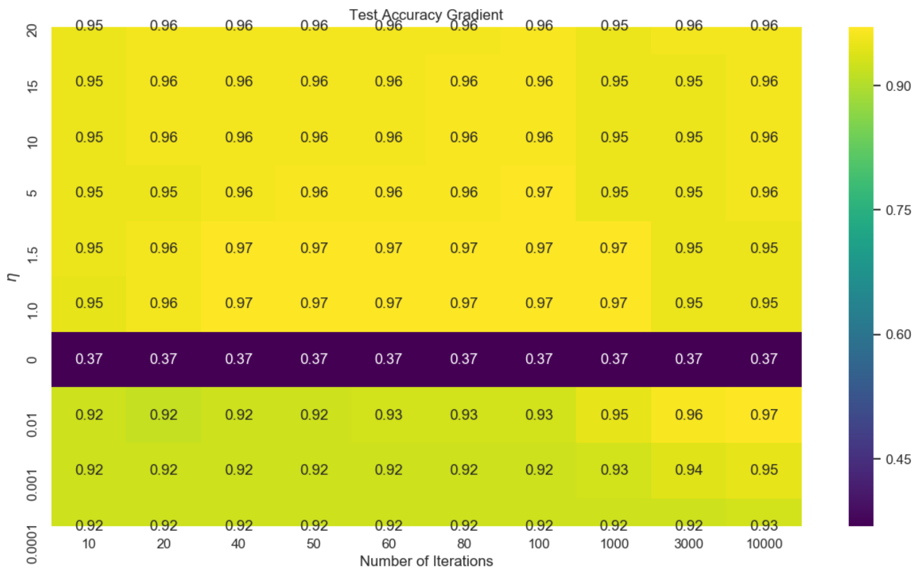


Figure h: Figure showing a seaborn plot of test accuracy varying with the number of iterations in x-axis, and the learning rate eta in y-axis. The optimization method used is gradient descent.

Then we looked at what logistic regression in scikit Learn perfomed. We used a L-BFGS solver, and got accuracy of 0.97 on test data and 0.99 on the train. The only difference we observed while doing our analysis was that our code used longer time in computing the different accuracies.
Sometimes the accuracy is not a good measurement of the accuracy of the model as we explained in the theory part. We wanted to se if this was the case in our data, and table(1) and (2) below show the confusion matrices gotten for the test gradient descent and Scikit Learn.

As shown the result show accuracy score is a good prediction model since false positive, and false negative almost are the same number.

Confusion matrix own

61	**2**
3	105

Table a: Confusion matrix for own logistic regression code.

Confusion matrix own

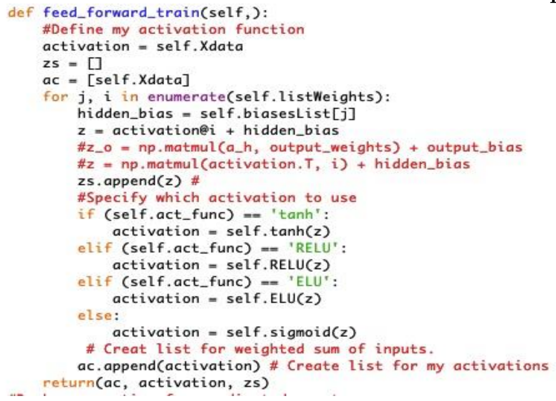
61	**2**
3	105

Table b: Confusion matrix for Scikit Learn logistic regression code.

Neural Network Classification

As we did for Logistic Regression we developed a class that could take in a number of input variables for the MLP. The variables are as follows; *Xdata,Ydata,act_func,,type_net, sizes, eta, lmbd, num_iter,, n_epochs , batch_size = 20. Xdata,* and *Ydata* represented the design matrix, and the output respectively. *act_func* represents the type of activation function we used, while *type_net* expresses what kind of measurment we do. If we do classification we plug in that, if we do regression likeways is done. *lmbd* represents the regularization paramter, *eta the learning rate*, *n_epoch* number of epochs if stochastic gradient is used to fit the model.

The Neural Network code we made consists of a feed forward, and a backpropagation part. In the feed forward we feed in starting with our input, calculate a weighted sum using equation (16), and pass it through an activation. As seen in our code in figure (8) below we have different activation functions, and calculate for the activation function we chose. Then the same activation that is calculated with (16) is passed through (16) again and a new weighted sum is calculated. This continues til we end up with the final output.



```
def feed_forward_train(self,):
    #Define my activation function
    activation = self.Xdata
    zs = []
    ac = [self.Xdata]
    for j, i in enumerate(self.listWeights):
        hidden_bias = self.biasesList[j]
        z = activation@i + hidden_bias
        #z_o = np.matmul(a_h, output_weights) + output_bias
        #z = np.matmul(activation.T, i) + hidden_bias
        zs.append(z) #
        #Specify which activation to use
        if (self.act_func) == 'tanh':
            activation = self.tanh(z)
        elif (self.act_func) == 'RELU':
            activation = self.RELU(z)
        elif (self.act_func) == 'ELU':
            activation = self.ELU(z)
        else:
            activation = self.sigmoid(z)
         # Creat list for weighted sum of inputs.
        ac.append(activation) # Create list for my activations
    return(ac, activation, zs)
```

Figure i:Shows how the feed forward for the Neural Network looked like.

The next part is the back propagation. Here errors for the bias and the weights are calculated, and by updating them, their gradients are produced by the function. Since we are new to programming we could not make our code flexible enough to both handle a back propagation both for stochastic and gradient. So, we created separate functions for stochastic, and gradient descent. This is commented in the code. Finally the methods are optimized by stochastic gradient, and gradient descent. The idea we used with the logistic regression is the same, but here the weights and biases are updated for each layer.
To measure the performance of the method we use the same prediction function we used for the Logistic Regression.

5.0.2 Regression

Neural Network

In the regression part, we analysed the Franke function by using the linear regression methods; OLS, Ridge and Lasso. The methods are thoroughly described in project 1, and we wont repeat theory behind nor the discussion. We will just present the results. Table (3) below shows result gotten for Ridge Regression when fitted to a Franke function of degree 5. In project 1 we found out that a polynomial degree of 11 gave the highest R2-score and MSE values Ridge Regression. But due to comparison we used a polynomial of degree 5. This is also the degree of polynomial we have used for analysis in our Neural Network.
Table(4) is the result gotten for Lasso Regression. For OLS by reading table (1) in project 1 we got that the R2 score, and MSE for polynomial degree 5 is 0.85 and 0.012.

$\lambda - values$	R^2-score	MSE
1,00E-7	0.834	0.013
1,00E-06	0.832	0.013
1,00E-05	0.78	0.016
1,00E-04	0.63	0.024
1,00E-03	-0.12	0.049
1,00E-02	-12.944	0.12
1,00E-01	-84.72	0.2
1,00E+00	0	0.25
1,00E+01	0	0.25

Table c: Shows the R2 score, and MSE for different values of λ . n = 40, and thus number of data points is 1600. Added noise is 0.1.

$\lambda - values$	R^2-score	MSE
1,00E-7	0.834	0.013
1,00E-06	0.832	0.013
1,00E-05	0.78	0.016
1,00E-04	0.63	0.024
1,00E-03	-0.12	0.049
1,00E-02	-12.944	0.12
1,00E-01	-84.72	0.2
1,00E+00	0	0.25
1,00E+00	0	0.25

Table d: Shows the R2 score, and MSE for different values of λ . n = 40, and thus number of data points is 1600. Added noise is 0.1. This is for Lasso

Lastly we did our analysis using MLP. The implementation of the network class is described under implementations, and we used the R2 score function to calculate R2 score, and MSE. This was done while changing the number of hidden nodes and layers, learning rate and the regularization parameter. It was not simple to get the optimal number of layers, and we had to run through numerous number of layers and nodes combinations to get a descent result. But, we ended up with what is shown in figure (8) below.

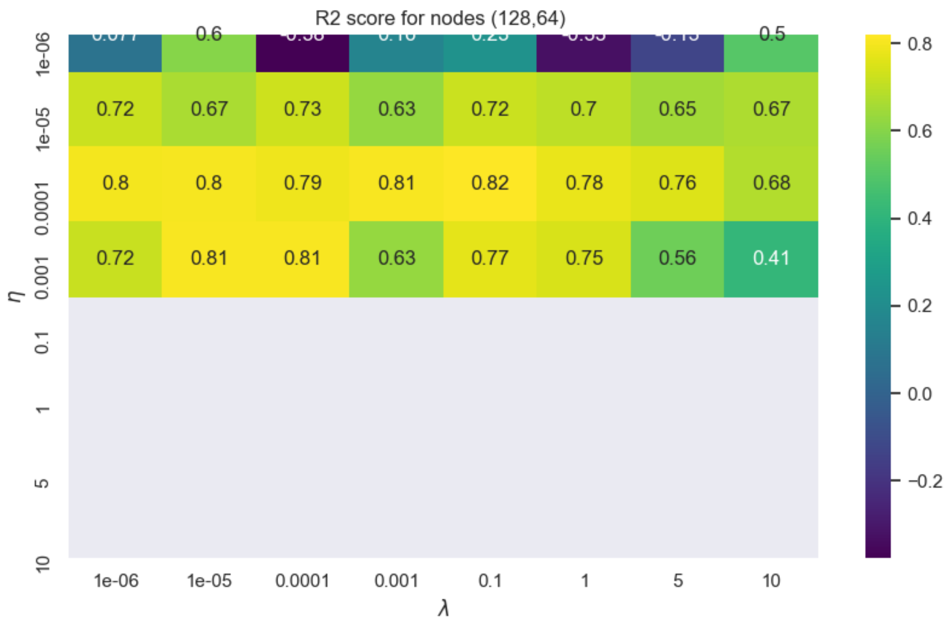


Figure j: Shows MLP performing a regression problem using the sigmoid function as an activation function. Number of hidden layers and nodes is written at the title. We have learning rate in y-axis, and regularization parameter in the

The MSE seaborn plot for the corresponding plot in figure (8) can be seen in figure (9) below.

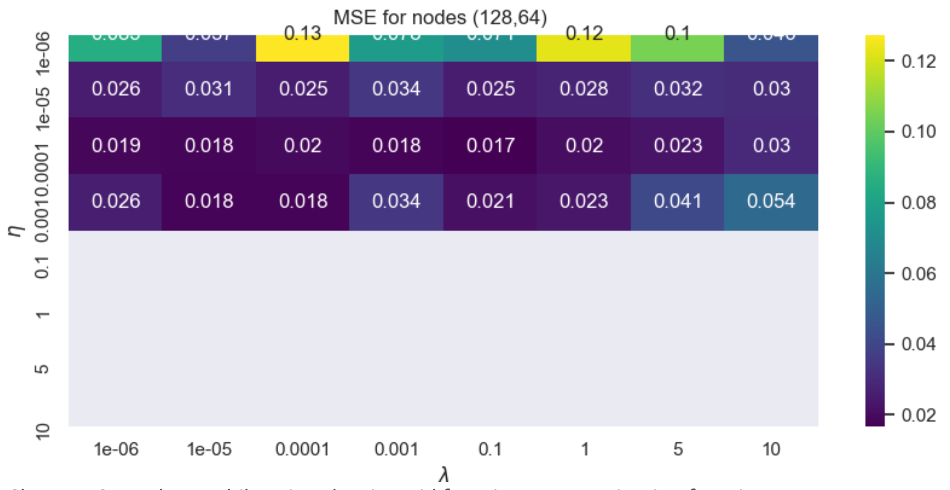


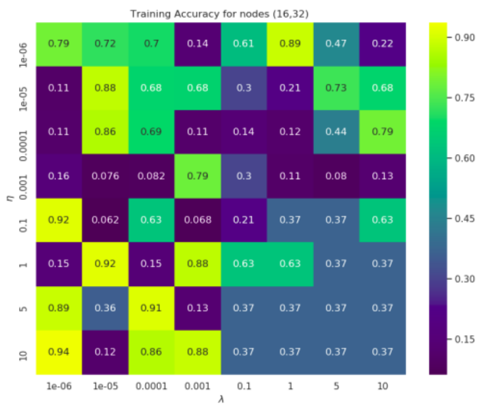
Figure k: Shows MSE seaborn while using the sigmoid function as an activation function.

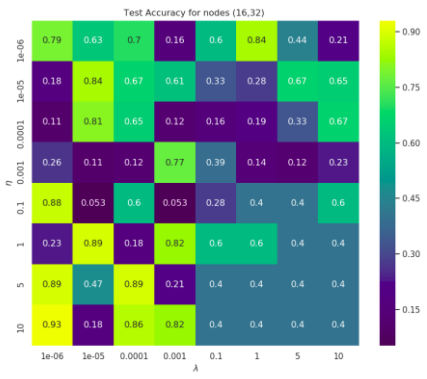
Looking at the plots above we see that optimal value of learning rate and regularization parameter equals 0.0001, and 0.1. The MSE and R2-score read from the figures above then become 0.82 and 0.017.
We did also similar analysis for RELU. They were not good at all, and seaborn graphs for different values are presented in the map called RELU in the figures folder in git hub.

5.0.2 Classification

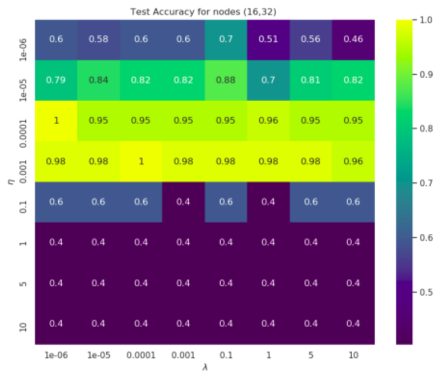
As explained in the introduction we studied the breast cancer data using Logistic Regression, and multi layer neural network. We did that by developing our own code, and also using a specialized package called Scikit Learn. In the project we have analysed how our simply produced code performed against the more professional Scikit Learn.

Neural Network
From the various tests that were run using neural network, we gathered that the best results in terms of train, test accuracy and time efficiency were obtained when the model was trained with 500 maximum iterations. In all the runs included in this section, 500 maximum iterations were used. We used different learning rates, η and different lambda values, λ for a range of hidden nodes in one and two hidden layers to find the optimal hidden nodes and layers for our neural network. For each pair of nodes, the test and train accuracy were calculated. We noticed that the optimal η and λ value varied depending on the number of nodes included in the hidden layers and the number of hidden layers included: Either one or two with gradient descent . For the neural network with one layer, using the ReLU activation function, The best training and test accuracies were observed for one of the following pairs of parameters: η = *0.1, 0.001,0.0001* and λ= *1e-06, 1e-05,-0.0001*. With an increase in the number of nodes for one layer, higher λs gave the best training and test accuracies. The one-layered hidden nodes *[1,8,16,32,64,128]* where used in this analysis. For two-layered hidden nodes, the following pairs were used in the calculations:
[[1,1],[1,8],[8,1],[8,64],[16,32],[32,16],[64,1],[64,32],[128,32]]. We discovered that the range of optimal η and λ *got broader.*





A similar analysis was carried out using sigmoid as the activation function. For most of the pairs of hidden nodes that were used, the optimal η was 0.001. The regularization rate however did not show much impact on the training and test scores that were observed. Figure 10 below are two plots indicating the training and test accuracy for different hidden nodes using the sigmoid function.



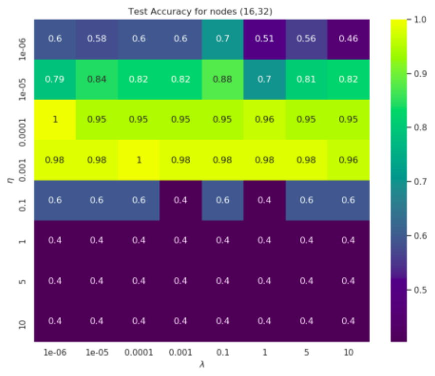


Figure l: train and test accuracy for different lambda and eta values for [16,32] hidden nodes

We compared the test and train accuracy scores of our Neural Network with and without PCA with the scikit learn neural network. The optimal parameters obtained earlier were used:

hidden nodes= [16,32] ,η = 0.001, λ= 0.0001 and *sigmoid* as the activation function. The results that were obtained are presented in the table 5 below. The dimension of the data was reduced to 2 components.

Table e: GD vs SGD for Neural network models with and without dimensionality reduction using PCA

Method		**Gradient Descent**		**SGD**	
		Train Accuracy	Test Accuracy	Train Accuracy	Test Accuracy
Customised Neural Network	Without PCA	0.9863	1.0	0.8613	0.8947
	With PCA	0.9627	0.8947		
		Adam		**SGD**	
Scikit learn Neural Network	Without PCA	0.9912	0.9649	0.6374	0.5877
	With PCA	0.9626	0.9122	0.94285	0.9122

For our customised neural network, Gradient Descent performed better than Stochastic gradient descent in the model. Gradient descent provided higher train and test accuracy. For Scikit learn Neural network, the MLPClassifier was used to train the model. Since this classifier does not provide gradient descent as an option for the solver, the option *'adam'* was used. The model performed extremely better for SGD with PCA using 2 components included in the preprocessing of the data. The *adam* solver performed better than SGD in both cases where dimensionality reduction using PCA was done or not. However, the test accuracy obtained by using these two solvers was the same for with dimensionality reduction.

Logistic Regression

We trained the logistic regression model using varying values of *η = [1e-05, 0.0001,0.001,0.01,0.1,1]* to obtain the train and test accuracies for gradient descent and stochastic gradient descent. The values are presented in the table 6 below.

Table f: Train Test Accuracy for different eta values for GD and SGD

η	**Gradient Descent**		**SGD**	
	Train Accuracy	**Test Accuracy**	**Train Accuracy**	**Test Accuracy**
1e-05	0.9422	0.92397	0.9598	0.9356
0.0001	0.9447	0.92397	0.9598	0.9532
0.001	0.9623	0.9298	0.9623	0.9649

0.01	0.9824	0.9474	0.9598	0.9649
0.1	0.9874	0.9707	0.9648	0.9532
1	0.9925	0.9707	0.9673	0.9532

Gradient descent and stochastic gradient descent are observed to give good train and test accuracies. Apart from the test accuracy of SGD, $\eta=1$ had the highest accuracies. Using logistic regression, with all factors being equal to this case, gradient descent and SGD would perform fairly the same.

We trained the Cancer data using our logistic regression model with varying training shares, when splitting the data into train and test sets, against varying regularization rate. The figure below presents the test accuracy that was obtained from this training. The results show that using 0.75 of the data set for training led to the highest test accuracy. The pattern of the colors in the seaborn plot show that the choice of the regularization rate had no effect on the accuracy of the model for different training shares.

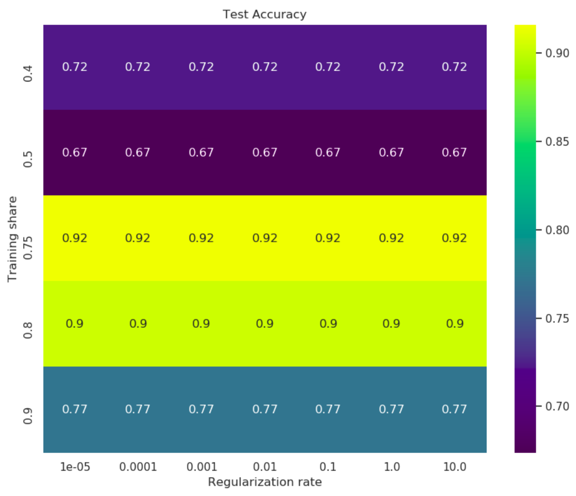


Figure m: Test Accuracy for different Regularization rates vs different Training shares when splitting data into train and test sets

We preprocessed our data with and without dimensionality reduction using PCA using scikit Learn's logistic regression to find the effect of the reduction on out scores. We tried using different dimensions ranging from *[2,5,10]* to see their effects. The results are presented in the table below.

Table g: Train and Test Accuracy for Logistic regression models with or without PCA dimensionality reduction.

Model		**Train Accuracy**	**Test Accuracy**
Data scaled but without PCA		0.99	0.96
Data scaled and PCA	n_components=2	0.9626	0.9122

used	n_components=5	0.9736	0.9298
	n_components=10	0.9846	0.9210

The dataset whose dimensionality was not reduced and used all the 30 features as presented and scaled had the highest train and test accuracy. For the dataset PCA dimensionality reduction performed, the order of accuracy for the different components was as follows: 10, 5 and lastly 2 with the lowest values. However, all the accuracy scores were higher than 0.92 demonstrating that all the models would be good enough to use.

5.1 Discussion

5.1.2 Regression

For the Cancer data, when PCA dimensionality reduction was included in the preprocessing, the accuracy got lower for both the neural network and logistic regression. With fewer PCA components being used, the accuracies were even lower. For logistic regression, all the accuracies were higher than 0.9 meaning that it would be recommended to reduce the dimensionality of our data if fitting a logistic regression model since the scores would still be considerably high. For the neural network, reducing the dimentionality when using stochastic gradient descent in an MLPClassifier predefined in scikit learn would make the model perform badly. Dimensionality reduction performed better for the logistic regression fit than the neural network.

For our tailor-made neural network, gradient descent was a better optimization algorithm as compared to stochastic gradient descent although in theory SGD was expected to give better accuracy scores. We observed that the size of our data set must have played a part in making gradient descent perform well. 569 samples were included with 30 features: This may be considerably small in relation to the size of most datasets that be trained using a neural network. We noticed that different parameters perform best in different sets of conditions. For the scikit learn, the conditions that were used when fitting showed a bit of bias towards the solver '*adam*' as compared to gradient descent. Even though '*adam*' *is a better solver than 'SGD',* SGD would have performed better than it did with a different combination of parameters.

Out of the activation functions that were used: Tanh, ReLU, Sigmoid and Identity: Sigmoid was found to be optimal. As in the seaborn plots for neural network, sigmoid had smaller ranges of η *and* λ *values* but for the optimal values highest accuracies were consistent. Different number of nodes and hidden layers would show the same optimal parameter values. To obtain the best accuracy, the ideal hidden layer had 2 layers with nodes *[16,32]*. Higher number of nodes performed good as well but this was ideal because the time to compute was low as compared to other nodes combinations with good performance. Using the ReLU activation function, there were a range of η *and* λ *values* which qualified to be considered as the optimal values but there was no consistency. Using ReLU, different combinations of hidden nodes in two layers had different optimal values which appeared in a manner that made it hard to locate a single optimal value for each parameter.

With 500 iterations, the neural network had the best results and less time used for computation. The ideal neural network in our case would have the following parameters:

η=0.001, λ =0.0001, max_iterations = 500, hidden_nodes = [16,32], activation function = sigmoid, solver = gradient descent with preprocessed data.

A neural network with a single layer with one node performs similar to logistic regression. Apart from the ability to have many nodes and hidden layers, neural network has many additional parameters that make it a better model for a classification problem as compared to logistic regression. How these two perform with the parameters is dependent on the type of data that is being used.

5.1.2 Regression

In the regression we got descent values for all the methods we implemented. The one that performed the worst was MLP. First of all it was very difficult as named to find optimal number of layers and secondly it is very complex compared to the simple linear regression methods. The paradox here is that the simples method works best and gives the highest R2 score and MSE at respectively as 0.85 and 0.012. Sometime easy is best.
But, on the other hand we have not tried out all activation functions, and not looked at every possible combination of layers. Also if we varied also the number of iteration we would have gotten higher R2 scores and lower MSE.

6. Conclusion

Neural network proved to be a better model to fit our classification data set as compared to logistic regression. With our Cancer data set, sigmoid performed best as our activation function with gradient descent as the activation function. PCA dimensionality reduction perfomed well for all neural network and logistic regression models except for scikit learn's MLP Classifer using SGD. For the regression dataset, for the activation functions used, fitting a simple linear regression would be the best option rather than a neural network or logistic regression model.

7. References

1. Casella, G., Fienberg, S., & Olkin, I. (2013). An Introduction to Statistical Learning. In *Springer Texts in Statistics*. https://doi.org/10.1016/j.peva.2007.06.006
2. Hastie, T. et. all. (2009). Springer Series in Statistics The Elements of Statistical Learning. *The Mathematical Intelligencer*, *27*(2), 83–85. https://doi.org/10.1007/b94608
3. Geron. (2017). *Hands on ML*. https://doi.org/10.3389/fninf.2014.00014
4. McKinney, W. (2012). *OReilly.Python.For.Data.Analysis*. Retrieved from http://www.postget.net/dl/63e3000032220300c515f152/%5Cnpapers3://publication/uuid/36B1A94F-FBB5-4FE5-B055-A41967EB8FC0
5. https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)
6. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
7. https://theclevermachine.wordpress.com/2014/09/08/derivation-derivatives-for-common-neural-network-activation-functions/
8. https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a
9. https://machinelearningmastery.com/cross-entropy-for-machine-learning/