UNIVERSITY OF OSLO

Project 2

Data Analysis and Machine Learning

FYS-STK3155/4155

Isabella Rositi

November 15, 2021

Abstract

An essential part of analyzing data is the ability of making accurate predictions, which often requires specific models that best fit this purpose. In this project the aim is to find out which model can best predict our data. Firstly, I will work on a regression problem, and I will split the dataset in a training and a test set in order to make predictions, so that the model can be trained on specific observations and then tested on new data. By doing so, the results are going to be more relevant. I will take into consideration different methods: Ordinary Least Squares and Ridge Regression, Stochastic Gradient Descent and Neural Networks. Among these models the Neural Network with three hidden layers and the ReLU as activation function turned out to be the one showing the smallest error, suggesting it works better than linear regression when making predictions. Secondly, I will focus on a classification problem following the same approach based on training and test sets, and I will evaluate the results of both Logistic Regression and Neural Networks. Neural Network works better also for the classification of this dataset. Choosing the correct model is of pivotal importance when performing analysis and this decision-making process should not be taken lightly, since wrong choices lead by definition to misleading results.

1. Introduction

Data is an extremely important tool and nowadays it can offer precious insights on everything, but being able to use it correctly to gain useful information is not as intuitive as it might seem. Researchers in the statistical field have studied and developed numerous methods that can be implemented when performing an analysis, but how to choose which one? This is one of the main problems everybody has to face when dealing with data: there is not a ready-made answer.

In this project the aim is to define the best model for this specific study, namely for making predictions. In fact, models perform differently based on the purpose of the analysis, and the results shown here have to be interpreted keeping this important aspect in mind. I am going to work with two different datasets, one for regression and one for classification. The first set of data contains information about Cars CO_2 Emissions and I am going to focus on the response variable which is explained by two features. The analysis starts with OLS and Ridge Regression and then I add both a Stochastic Gradient Descent algorithm and a Momentum SGD. Lastly, I compute Neural Networks with different numbers of hidden layers, nodes, and different activation functions to answer the question previously asked: Which model is the best? Since I am interested in prediction making, I use the *Test Mean Square Error* as cost function and the goal is to minimize it. The second dataset is the so-called Wisconsin Breast Cancer Data, which has a dichotomous response variable explained by thirty features, from which I am going to select eight in order to perform the analysis. The analysis starts with Logistic Regression implemented by using a Stochastic Gradient Descent algorithm and then I compute the same structure of Neural Networks as in the previous problem. The aim stays unvaried, but for this classification problem I compute the *Accuracy Score*, with the aim of maximizing it.

I will describe all the methods I use in this project, then I will present the salient outputs and the respective analysis followed by the conclusion.

2. Methods

2.1 Data

2.1.1 Cars CO_2 Emissions

The first dataset I will be using to carry out the regression analysis contains information about Car CO_2 Emissions, and the focus is on the response variable *Co2 Emissions* calculated in g/km, which is explained by two features: *Engine Size* calculated in L and *Fuel Consumption*, both in the city and on the highway, calculated in L/100km. The two features are included in the model as a polynomial up to fifth order, which means that the design matrix has dimensions (7385x20).

	Engine Size(L)	Fuel Consumption Comb (L/100 km)	CO2 Emissions(g/km)
1	2.0	8.5	196
2	2.4	9.6	221
3	1.5	5.9	136
4	3.5	11.1	255
5	3.5	10.6	244

Table 1

For the purpose of the analysis, as shown in Project 1, it is best to scale the data by standardizing the variables. By doing so, the range of the variance decreases immensely and the predictions result more accurate.

2.1.2 Wisconsin Breast Cancer

The second dataset is going to be used for classification: it is a dataset on breast cancer, that focuses on the response variable *diagnosis* that is dichotomous and represents the nature of the tumor (Bening or Malign). The features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. There are 30 features, but I am going to perform an analysis based on correlations and collinearity in order to choose a subset of variables to include in the model.

The features that have very low correlation in absolute value with the response variable are the first ones to be taken out of the model. Secondly, by taking a look at the correlations between features, the values that are too high in absolute value correspond to covariates highly correlated, which could cause problems of collinearity,. Therefore, I am going to choose just one of the two variables to include in the model. Repeating this process a sufficient amount of times, the features reaming are the following eight.

	symmetry_mean	radius_se	concave points_se	texture_worst	perimeter_worst	smoothness_worst	concave points_worst	symmetry_worst	dignosis
0	0.2419	1.0950	0.01587	17.33	184.60	0.16220	0.2654	0.4601	1
1	0.1812	0.5435	0.01340	23.41	158.80	0.12380	0.1860	0.2750	1
2	0.2069	0.7456	0.02058	25.53	152.50	0.14440	0.2430	0.3613	1
3	0.2597	0.4956	0.01867	26.50	98.87	0.20980	0.2575	0.6638	1
4	0.1809	0.7572	0.01885	16.67	152.20	0.13740	0.1625	0.2364	1
...	...	...	...	...	...	...	...	...	...
564	0.1726	1.1760	0.02454	26.40	166.10	0.14100	0.2216	0.2060	1
565	0.1752	0.7655	0.01678	38.25	155.00	0.11660	0.1628	0.2572	1
566	0.1590	0.4564	0.01557	34.12	126.70	0.11390	0.1418	0.2218	1
567	0.2397	0.7260	0.01664	39.42	184.60	0.16500	0.2650	0.4087	1
568	0.1587	0.3857	0.00000	30.37	59.16	0.08996	0.0000	0.2871	0

Table 2

The correlation matrix of the subset of chosen variables.

	symmetry_mean	radius_se	concave points_se	texture_worst	perimeter_worst	smoothness_worst	concave points_worst	symmetry_worst	dignosis
symmetry_mean	1.000000	0.303379	0.393298	0.090651	0.219169	0.426675	0.430297	0.699826	0.330499
radius_se	0.303379	1.000000	0.513346	0.194799	0.719684	0.141919	0.531062	0.094543	0.567134
concave points_se	0.393298	0.513346	1.000000	0.086741	0.394999	0.215351	0.602450	0.143116	0.408042
texture_worst	0.090651	0.194799	0.086741	1.000000	0.365098	0.225429	0.359755	0.233027	0.456903
perimeter_worst	0.219169	0.719684	0.394999	0.365098	1.000000	0.236775	0.816322	0.269493	0.782914
smoothness_worst	0.426675	0.141919	0.215351	0.225429	0.236775	1.000000	0.547691	0.493838	0.421465
concave points_worst	0.430297	0.531062	0.602450	0.359755	0.816322	0.547691	1.000000	0.502528	0.793566
symmetry_worst	0.699826	0.094543	0.143116	0.233027	0.269493	0.493838	0.502528	1.000000	0.416294
dignosis	0.330499	0.567134	0.408042	0.456903	0.782914	0.421465	0.793566	0.416294	1.000000

Table 3

Also for this dataset I am going to scale the data, applying a normalization to each of the features included.

2.2 Stochastic Gradient Descent

Stochastic gradient descent is an extension of the gradient descent algorithm, which is introduced in Machine Learning to find the minimum of a function, in this case specifically to find the values of the parameters that minimize the cost function. A gradient is a derivative of a function that has at least two input variables and it is also known as the slope of a function in mathematical terms. The higher the gradient, the steeper the slope. If the slope is zero, however, the model stops learning.

The algorithm starts by defining the initial values of the parameters, which are usually chosen randomly, and from there the gradient descent uses calculus to iteratively adjust the values so that they minimize the given cost function. At each step the new values are found by an equation including the gradient and a learning rate. The learning rate defines the size of the steps the gradient descent takes into the direction of the minimum. If the learning rate is too small it might take a lot of iterations to reach the minimum, if it is too big, it might miss it because overstepped.

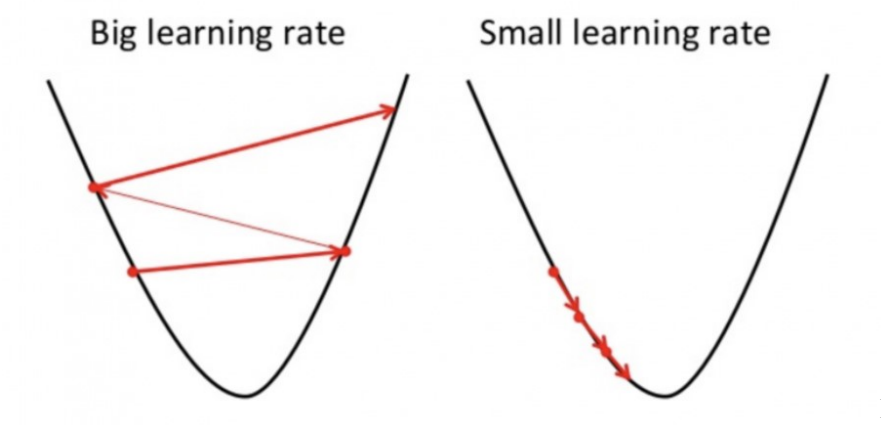

Figure 1

Moreover, the training set is divided into several mini-batches that are independent with one another in order to make it easier when dealing with high-dimensional problems to calculate the Hessian matrix (it coincides with the second order derivative and it is used to calculate the gradient). The gradient obtained on a mini-batch B_k is used to implement the gradient of the following mini-batch B_{k+1}. When the gradients have been calculated and the parameters have been updated for all the mini-batches, an epoch has been completed. The researcher can decide the number of epochs that fits best the study.

The main formula is:

$$\beta^{(n+1)} = \beta^{(n)} - \eta^{(n)} \nabla_{\beta} C(\beta^{(n)})$$

2.2.1 The Algorithm

1. Initialize the learning rate η

2. Initialize the parameters β

3. Initialize number of epochs and size of mini-batches

While stopping criterion not met, do:

4. Sample a mini-batch of observations from the training set (with M size of mini-batch)

$$B_0 = \{(x_0, y_0), \ldots, (x_{M-1}, y_{M-1})\}$$

in total we sample $\frac{n}{M}$ mini-batches

5. Compute the gradient: $g = \frac{1}{M} \sum \nabla_\beta C(\beta)$

6. Update: $\beta \leftarrow \beta - \eta g$

End do.

It is common to apply a decay to the learning rate, in order to speed up the convergence process. In this project two different methods are used:

2.2.2 Time-Based Learning Schedule

The time-based learning schedule decrements the learning rate η at every epoch, using a hyperparameter of decay d multiplied by the number of the epoch currently running.

$$\eta^{(k+1)} = \frac{\eta^{(k)}}{(1 + d * e^{(k+1)})}$$

2.2.2 Drop-Based Learning Schedule

The drop-based learning schedule decrements the initial learning rate $\eta^{(0)}$ every e_d epochs by using a hyperparameter of decay d.

$$\eta^{(k+1)} = \eta^{(0)} \, d^{\frac{e^{(k+1)}}{e_d}}$$

Where $\frac{e^{(k+1)}}{e_d}$ has to be rounded down to the nearest integer.

When gradient descent cannot decrease the cost function anymore and remains constant, then it has converged. The insight of SGD is that the gradient is an expectation. It is not guaranteed that the optimization algorithm arrives at at a local minimum in a reasonable amount of time, but it often finds a very low value of the cost function quickly enough to be useful.

In Python, the code for implementing the stochastic gradient descent will look like this. Here is implemented a drop-based decay schedule.

```
min_mse = float("inf")
for lmb in lambdas:
    for eta in etas:
        eta_dec = eta
        np.random.seed(198)
        theta = np.random.randn(features, 1)
        for d in drop:
            for epoch in range(epochs):
                chunk = 0
                for i in range(m):
                    chunkrows = range(chunk, chunk+M)
                    xi = X5_train[chunkrows]
                    zi = z_train[chunkrows]
                    gradient = (2.0/M) * xi.T @ (xi @ (theta) - zi) + 2*lmb*theta
                    theta -= eta_dec*gradient
                    chunk += M
                eta_dec = eta*d**floor(epoch/epoch_drop)
                zpred_sgd = X5_test @ theta
                estimated_mse = MSE(z_test, zpred_sgd)
                if  estimated_mse < min_mse:
                    best_eta = eta
                    min_mse = estimated_mse
                    best_drop = d
                    best_lambda = lmb

print("Best MSE = ", min_mse)
print("Best eta = ", best_eta)
print("Best drop = ", best_drop)
print("Best lambda = ", best_lambda)
```

When dealing with a OLS model, the value of lmb is zero and there is no penalization of the parameters.

2.3 Momentum SGD

Many variants of gradient descent have been developed and are nowadays widely used. In this project it is implemented the Momentum Stochastic Gradient Descent. In order to demonstrate its algorithm, let's Taylor expand the cost function by first defining $\hat{\beta}$.

$$\hat{\beta} = \beta^{(n+1)} = \beta^{(n)} - \eta^{(n)} g^{(n)}$$

Now its cost-function can be written as:

$$C(\hat{\beta}) \simeq C(\beta^{(n)}) + (\hat{\beta} - \beta^{(n)}) g^{(n)} + \frac{1}{2} (\hat{\beta} - \beta^{(n)})^T H (\hat{\beta} - \beta^{(n)})$$

Only the first two derivatives are included because it is assumed that the derivatives of higher order have very little to none impact on the cost function. This method is particularly useful when the second derivative, which represents the curvature of the function, results to be greater than the gradient and therefore having a greater impact on the cost function. In fact, the momentum helps in regions with ravines and the updates of the β parameters depend also on a new set of parameters v and a hyperparameter α. Ravine is an area where the surface curves much more steeply in one dimension than in another and they are common near local minima. SGD has troubles navigating them and will tend to oscillate across the narrow ravine since the negative gradient will point down one of the steep sides rather than along the ravine towards the optimum. Momentum helps accelerate gradients in the right direction.

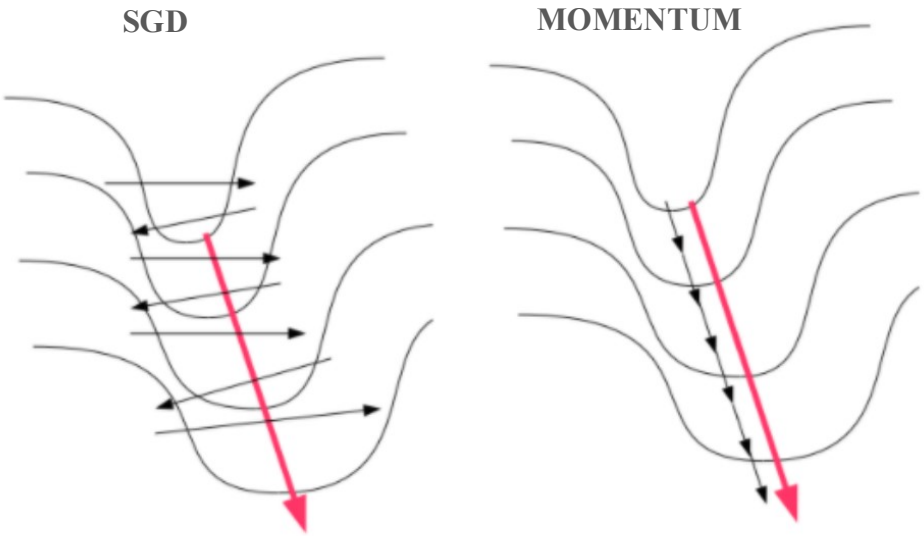

Figure 2

2.3.1 The Algorithm

1. Initialize the learning rate η and the momentum parameter α

2. Initialize the parameters β and v

3. Initialize number of epochs and size of mini-batches

While stopping criterion not met, do:

4. Sample a mini-batch of observations from the training set (with M size of mini-batch)

$$B_0 = \{(x_0, y_0), \ldots, (x_{M-1}, y_{M-1})\}$$

in total we sample $\frac{n}{M}$ mini-batches

5. Compute the gradient: $g = \frac{1}{M}\sum \nabla_\beta C(\beta)$

6. Compute: $v \leftarrow \alpha v - \eta g$

7. Update: $\beta \leftarrow \beta + v$

End do.

In Python, the code for implementing the momentum stochastic gradient descent will look like this.

```
min_mse = float("inf")
for alpha in alphas:
    for lmb in lambdas:
        for eta in etas:
            np.random.seed(198)
            theta = np.random.randn(features,1)
            v = np.random.randn(features,1)
            for epoch in range(epochs):
                chunk = 0
                for i in range(m):
                    chunkrows = range(chunk, chunk+M)
                    xi = X5_train[chunkrows]
                    zi = z_train[chunkrows]
                    gradient = (2.0/M) * xi.T @ ((xi @ theta) - zi) + 2*lmb*theta
                    v = alpha * v - eta * gradient
                    theta += v
                    chunk += M
                zpred_sgd = X5_test @ theta
                estimated_mse = MSE(z_test, zpred_sgd)
                if  estimated_mse < min_mse:
                    best_eta = eta
                    min_mse = estimated_mse
                    best_alpha = alpha
                    best_lambda = lmb

print("Best MSE = ", min_mse)
print("Best eta = ", best_eta)
print("Best alpha = ", best_alpha)
print("Best lambda = ", best_lambda)
```

Here is implemented a drop-based decay schedule. As for the SGD there is a lmb parameter that is equal to zero when dealing with OLS models.

2.4 Neural Networks

An Artificial Neural Network (ANN) is a computational non-linear model for supervised learning that consists of layers of connected neurons (or nodes). It is supposed to mimic a biological nervous system by letting each neuron interact with other neurons by sending signals in the form of mathematical functions between different layers. A wide variety of ANNs have been developed, but most of them consist of an input layer, eventual in-between layers (hidden layers) and an output layer. All layers can contain an arbitrary number of nodes, each connection between two nodes is associated with a weight variable and each time the information goes from a layer to another there is a threshold (bias) that needs to be reached in order for the signal to move forward. There are several types of Neural Networks, but the one implemented in this project is a feed-forward neural network. In this network the information moves in only one direction: forward through the layers, from the input to the output.

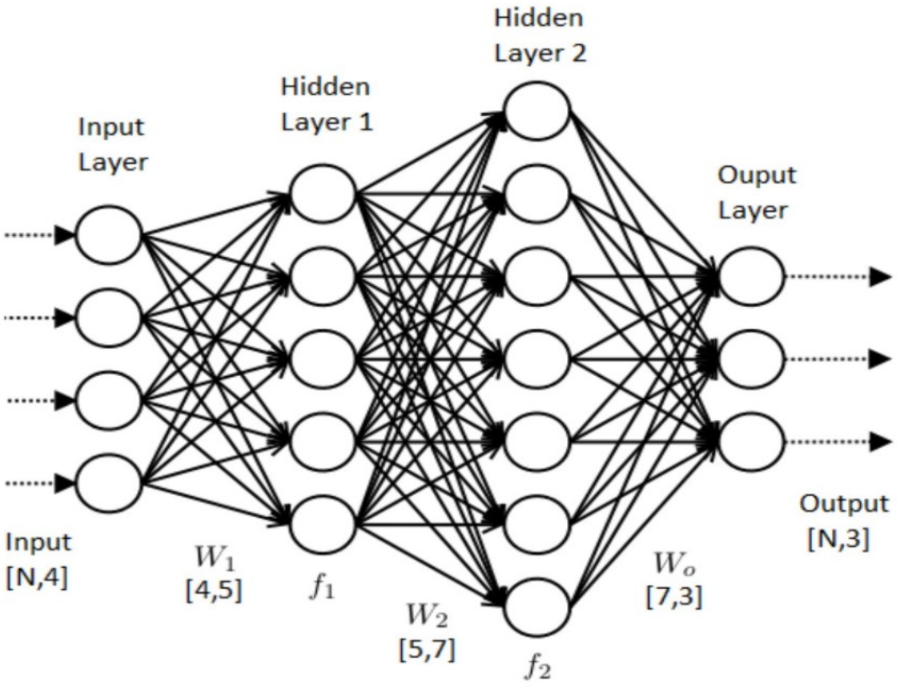

Figure 3

Nodes are represented by circles, while the arrows display the connections between the nodes, including the direction of the information flow. Additionally, each arrow corresponds to a weight variable. It is observable that each node in a layer is connected to all nodes in the subsequent layer, making this a so-called fully-connected FFNN. One uses often the fully-connected feed-forward neural networks with three or more layers (an input layer, one or more hidden layers and an output layer) consisting of neurons that have non-linear activation functions. Such networks are often called multilayer perceptrons (MLPs).

2.4.1 Mathematical Model

The output y is produced via the activation function f:

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

Where w_i is the weight for every neuron and b_i is the bias. Usually the bias is kept constant throughout the whole layer. This function receives x_i as inputs. In a FFNN the inputs x_i are the outputs from the previous layer.

Let's start considering the first hidden layer: for each node i is calculated a weighted sum z_i^1 of the input coordinates x_j:

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

Where M is the total number of possible inputs to the i-th node and the superscript 1 refers to the first hidden layer. The value of z_i^1 is the argument of the activation function f_i of each node i and the output is a_i^1:

$$a_i^1 = f(z_i^1) = f(\sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1)$$

where it is assumed that all nodes in the same layer have identical activation functions.

The output of neuron i in layer 2 is then:

$$a_i^2 = f^2(z_i^2) = f^2(\sum_{j=1}^{N} w_{ij}^2 \, y_j^1 + b_i^2) = f^2[\sum_{j=1}^{N} w_{ij}^2 \, f^1(\sum_{k=1}^{M} w_{jk} \, x_k + b_j^1) + b_i^2]$$

And so on for every layer.

The generalized expression for an MLP with l hidden layers is:

$$a_i^{l+i} = f^{l+1}[\sum_{j=1}^{N^l} w_{ij}^{l+1} \, f^l(\sum_{k=1}^{N^{l-1}} w_{jk}^l \, f^{l-1}(\dots f^1(\sum_{n=1}^{N^0} w_{mn}^1 \, x_n + b_m^1) \dots) + b_k^l) + b_j^{l+1}]$$

2.4.2 Activation Functions

Each layer has an activation function that links the inputs to the outputs and therefore one layer to the next. There are various activation functions that are used in different situations and work better with different types of data. The activation functions can also be different among the hidden layers, moreover, special attention has to be put on the activation function of the output layer, because it determines whether the problem is regression or classification.

The activation functions have to have certain proprieties:

- be non-constant
- be bounded
- be monotonically increasing
- be continuous

In this project the functions used for the hidden layers are:

1. Sigmoid
2. ReLU (Rectified Linear Unit)
3. Leaky-RELU

and the ones used for the output layer are:

4. Linear
5. Softmax

1. Sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

with derivative equal to:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

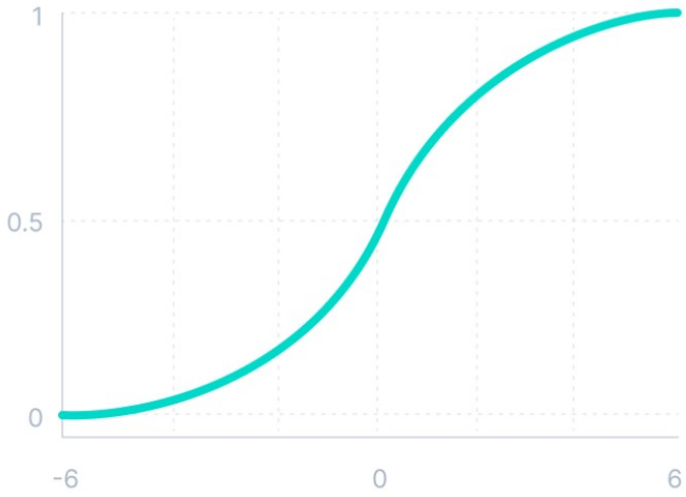

Figure 4

The main reason why the sigmoid function is frequently used, is because it exists between 0 and 1. Therefore, it is especially used for models where we have to predict the probability as an output. However, when $|z| >> 0$, then $\sigma'(z)$ is very small and it can lead to the "vanishing gradient" problem. This means that the gradient becomes zero, the training of the NN stops and the model is not able to find the right values for the weights. Therefore, other activation functions have been introduced to avoid this problem.

2. ReLU

$$ReLU(z) = max\{0, z\}$$

with derivative equal to:

$$ReLU'(z) = \begin{cases} 1 & if\ z \geq 0 \\ 0 & if\ z < 0 \end{cases}$$

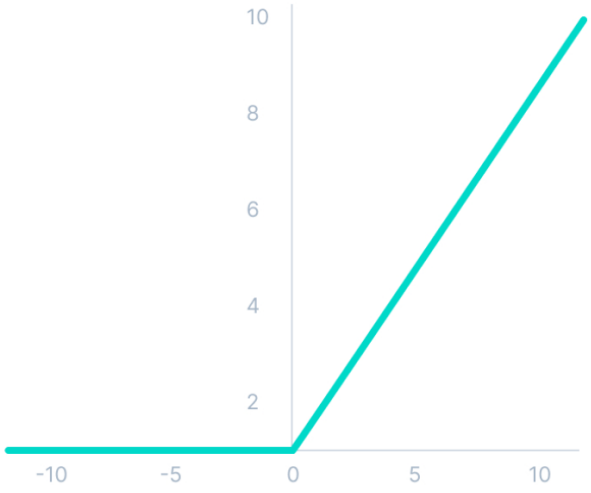

Figure 5

The issue with the ReLU is that all the negative values become zero immediately and this decreases the ability of the model to properly train (and then fit) the data. To overcome this problem, the leaky-ReLU has been introduced.

3. Leaky-ReLU

$$L\_ReLU(z) = \begin{cases} z & if\ z \geq 0 \\ \alpha z & if\ z < 0 \end{cases}$$

with derivative equal to:

$$L\_ReLU'(z) = \begin{cases} 1 & if\ z \geq 0 \\ \alpha & if\ z < 0 \end{cases}$$

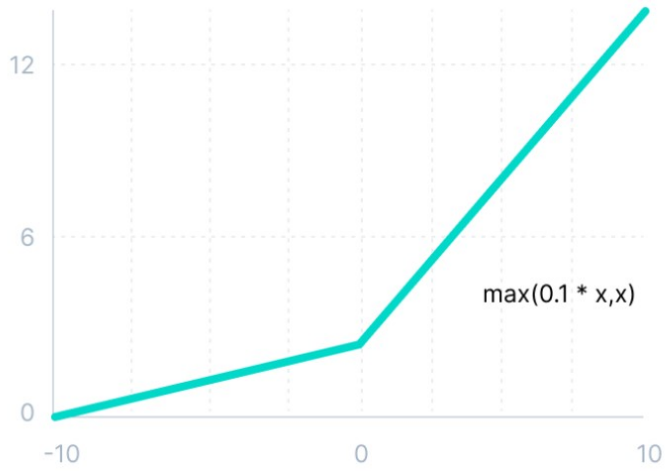

Figure 6

Usually, the value of α is 0.01. Using Leaky-ReLU the gradient never becomes zero. Empirically the Leaky-ReLU works better than the ReLU, but this may not always be the case.

4. Linear

$$f(z) = z$$

The function is just a bisector of the first and third quadrant, which gives as an output the input itself. The derivative is obviously:

$$f'(z) = 1$$

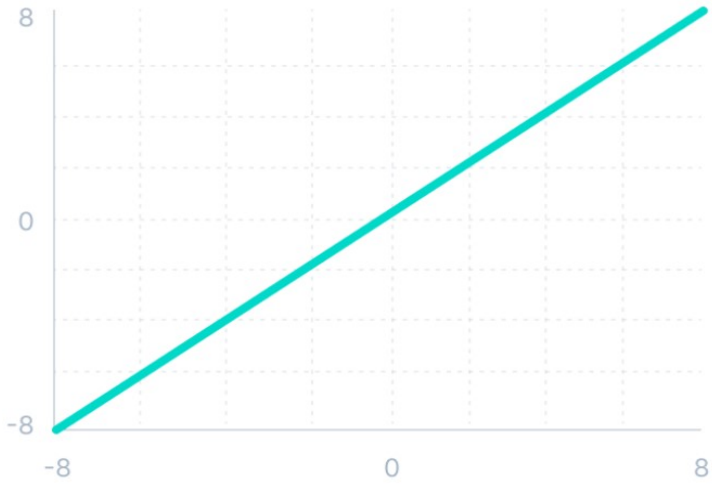

Figure 7

This type of activation function is used in the output layer of NN models when they are dealing with a regression problem. That is because the output must be a real number and not a probability that has to be decided whether it is closer to 0 or 1. In fact, the output of the function will not be confined between any range.

5. Softmax

$$f(z_i) = \frac{e^{z_i}}{\sum_{m=1}^{K} e^{z_m}}$$

with derivative with respect to z_i equal to:

$$f'(z_i) = f(z_i)(\delta_{ij} - f(z_j))$$

Softmax turns vectors into probabilities, which is exactly what the purpose of classification is. The exponential function in the formula ensures that the obtained values are non-negative. Due to the normalization term in the denominator, the obtained values sum to 1. Furthermore, all values lie between 0 and 1.

2.4.3 Back Propagation Algorithm

When training a FFNN the process starts with the random initialization of weights and biases, but in order to train properly the data, the weights need to be updated. To do so, the back propagation algorithm has been introduced. Let's define the cost function as:

$$C(\widehat{W}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - t_i)^2$$

where t_i are the targets and y_i are the outputs of the network.

Let's now define the activation of the j-th node of the l-th layer as function of the weights, the bias and the output of the previous layer:

$$z_j^l = \sum_{i=1}^{M^{l-1}} w_{ij}^l \, a_i^{l-1} + b_j^l$$

Here M^{l-1} represents the total number of nodes of the $(l-1)$-th layer.

Starting from the activation values z^l in turn it is possible to define the output of layer l as:

$$a^l = f(z^l)$$

From the definition of the activation z_j^l it can be written:

$$\frac{\partial z_j^l}{\partial w_{ij}^l} = a_i^{l-1}$$

$$\frac{\partial z_j^l}{\partial a_i^{l-1}} = w_{ij}^l$$

And then also:

$$\frac{\partial a_j^l}{\partial z_j^l} = a_j^l(1 - a_j^l) = f(z_j^l)(1 - f(z_j^l))$$

With these definitions it is possible to compute the derivative of the cost function in terms of the weigh ts.

Let's refer to the output layer $l = L$. The cost function is:

$$C(\widehat{W^L}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - t_i)^2 = \frac{1}{2}\sum_{i=1}^{n}(a_i^L - t_i)^2$$

and the derivative of this function with respect to the weights is:

$$\frac{\partial C(\widehat{W^L})}{\partial w_{jk}^L} = (a_j^L - t_j)\frac{\partial a_{jk}^L}{\partial w_{jk}^L}$$

The last partial derivative can be computed by applying the chain rule:

$$\frac{\partial a_j^L}{\partial w_{jk}^L} = \frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} = a_j^L(1 - a_j^L)a_k^{L-1}$$

Summing up, the first back propagation equation (the one referring to the output layer) is:

$$\frac{\partial C(\widehat{W^L})}{\partial w_{jk}^L} = (a_j^L - t_j)\frac{\partial a_{jk}^L}{\partial w_{jk}^L} = (a_j^L - t_j)a_j^L(1 - a_j^L)a_k^{L-1}$$

and defining:

$$\delta_j^L = a_j^L(1 - a_j^L)(a_j^L - t_j) = f'(z_j^L)\frac{\partial C}{\partial a_j^L}$$

Using the Hadamard product of two vectors this can be written as:

$$\delta^L = f'(z^L) \circ \frac{\partial C}{\partial a^L}$$

In the Hadamard product of two vectors the elements corresponding to same row and columns of given matrices are multiplied together to form a new matrix.

The first term on the right-hand side measures how fast the activation function f is changing for a given activation value z_j^L. The second term on the right-hand side measures how fast the cost function is changing as function of the j-th output activation. If, for example, the cost function does not depend much on a particular output node j, then δ_j^L will be small.

By defining δ_j^L a more compact definition of the derivative of the cost function in terms of the weights can be written:

$$\frac{\partial C(\widehat{W^L})}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}$$

Finally, generally speaking, the back propagation algorithm is:

1. Set up the input data x and initialize the weights and the biases

2. Perform a FFNN until the output layer is reached and compute all z^l and the pertinent outputs a^l for $l = 2, 3, \ldots, L$

3. Compute the error $\hat{\delta}^L$:

$$\hat{\delta}_j^L = f'(z_j^L)\frac{\partial C}{\partial a_j^L}$$

4. Compute the back propagation error for each layer

$$\hat{\delta}_j^l = \sum_k \delta_k^{l+1} \; w_{kj}^{l+1} \; f'(z_j^l)$$

5. Update the weights and biases using gradient descent for each layer

$$w_{jk}^l \leftarrow w_{jk}^l - \eta \; \delta_j^l \; a_k^{l-1}$$

$$b_j^l \leftarrow b_j^l - \eta \; \delta_j^l$$

The parameter η is the learning parameter discussed in connection with the gradient descent methods. It is convenient to use stochastic gradient descent with mini-batches. Once the back propagation algorithm reaches the first hidden layer, then a new FFNN starts, using the updated parameters. The process goes back and forth until the difference between the target and the outputs is small enough.

For the implementation in Python a class object has been created.

```
class NeuralNetwork:
    def __init__(self, X, y, n_hidden_neurons, n_categories, n_hidden_layers, epochs, batch_size, eta, lmb, activation, weight):

        self.X_full = X
        self.y_full = y

        self.n_inputs = X.shape[0]
        self.n_features = X.shape[1]
        self.n_hidden_neurons = n_hidden_neurons
        self.n_categories = n_categories
        self.n_hidden_layers = n_hidden_layers

        self.epochs = epochs
        self.batch_size = batch_size
        self.iterations = self.n_inputs // self.batch_size
        self.eta = eta
        self.lmb = lmb
        self.activation = activation
        self.weight = weight

        self.create_biases_and_weights()

    def create_biases_and_weights(self):
        self.w = []
        if self.weight == 0:
            self.w.append(np.zeros((self.n_features, self.n_hidden_neurons)))
            for i in range(1, self.n_hidden_layers):
                self.hidden_weights = np.zeros((self.n_hidden_neurons, self.n_hidden_neurons))
                self.w.append(self.hidden_weights)
            self.hidden_bias = np.zeros(self.n_hidden_neurons)
            self.output_weights = np.zeros((self.n_hidden_neurons, self.n_categories))
            self.output_bias = np.zeros(self.n_categories)
        else:
            self.w.append(np.random.randn(self.n_features, self.n_hidden_neurons))
            for i in range(1, self.n_hidden_layers):
                self.hidden_weights = np.random.randn(self.n_hidden_neurons, self.n_hidden_neurons)
                self.w.append(self.hidden_weights)
            self.hidden_bias = np.zeros(self.n_hidden_neurons) + 0.01
            self.output_weights = np.random.randn(self.n_hidden_neurons, self.n_categories)
            self.output_bias = np.zeros(self.n_categories) + 0.01
```

```
def feed_forward(self):
    # feed-forward for training
    self.a = []
    self.z = []
    self.z_h = np.matmul(self.X, self.w[0]) + self.hidden_bias
    self.a_h = self.activation(self.z_h)
    self.a.append(self.a_h)
    self.z.append(self.z_h)

    for i in range(1, self.n_hidden_layers):
        self.z_h = np.matmul(self.a[i-1], self.w[i]) + self.hidden_bias
        self.a_h = self.activation(self.z_h)
        self.a.append(self.a_h)
        self.z.append(self.z_h)

    self.z_o = np.matmul(self.a[n_hidden_layers-1], self.output_weights) + self.output_bias

    if self.n_categories > 1:
        exp_term = np.exp(self.z_o)
        self.probabilities = exp_term / np.sum(exp_term, axis=1, keepdims=True)


def feed_forward_out(self, X):
    # feed-forward for output
    a = []
    z = []
    z_h = np.matmul(X, self.w[0]) + self.hidden_bias
    a_h = self.activation(z_h)
    a.append(a_h)
    z.append(z_h)

    for i in range(1, self.n_hidden_layers):
        z_h = np.matmul(a_h, self.w[i]) + self.hidden_bias
        a_h = self.activation(z_h)
        a.append(a_h)
        z.append(z_h)

    z_o = np.matmul(a[n_hidden_layers-1], self.output_weights) + self.output_bias

    if self.n_categories > 1:
        exp_term = np.exp(z_o)
        probabilities = exp_term / np.sum(exp_term, axis=1, keepdims=True)
        return probabilities
    else:
        return z_o
```

```
def backpropagation(self):

    if self.n_categories > 1:
        error_output = self.probabilities - self.y
    else:
        error_output = self.z_o - self.y

    self.a.reverse()
    self.z.reverse()
    self.w.reverse()
    self.err_h = []

    if self.activation == sigmoid:
        error_hidden = np.matmul(error_output, self.output_weights.T) * grad_sigmoid(self.a[0])
    elif self.activation == relu:
        error_hidden = np.matmul(error_output, self.output_weights.T) * grad_relu(self.z[0])
    else:
        error_hidden = np.matmul(error_output, self.output_weights.T) * grad_leaky_relu(self.z[0])
    self.err_h.append(error_hidden)
```

```
    for i in range(1, self.n_hidden_layers):
        if self.activation == sigmoid:
            error_hidden = np.matmul(self.err_h[i-1], self.w[i-1].T) * grad_sigmoid(self.a[i])
            self.err_h.append(error_hidden)
        elif self.activation == relu:
            error_hidden = np.matmul(self.err_h[i-1], self.w[i-1].T) * grad_relu(self.z[i])
            self.err_h.append(error_hidden)
        else:
            error_hidden = np.matmul(self.err_h[i-1], self.w[i-1].T) * grad_leaky_relu(self.z[i])
            self.err_h.append(error_hidden)

    self.output_weights_gradient = np.matmul(self.a[0].T, error_output)
    self.output_bias_gradient = np.sum(error_output, axis=0)

    self.g = []
    for i in range(1, self.n_hidden_layers):
        self.hidden_weights_gradient = np.matmul(self.a[i].T, self.err_h[i-1])
        self.g.append(self.hidden_weights_gradient)

    self.hidden_weights_gradient = np.matmul(self.X.T, self.err_h[-1])
    self.g.append(self.hidden_weights_gradient)
    self.hidden_bias_gradient = np.sum(error_hidden, axis=0)
```

```
    if self.lmb > 0.0:
        self.output_weights_gradient += self.lmb * self.output_weights
        for i in range(len(self.w)):
            self.g[i] += self.lmb * self.w[i]
    self.output_weights -= self.eta * self.output_weights_gradient
    self.output_bias -= self.eta * self.output_bias_gradient

    for i in range(len(self.w)):
        self.w[i] -= self.eta * self.g[i]
    self.hidden_bias -= self.eta * self.hidden_bias_gradient

    self.a.reverse()
    self.z.reverse()
    self.w.reverse()

def predict(self, X):
    if self.n_categories > 1:
        probabilities = self.feed_forward_out(X)
        return np.argmax(probabilities, axis=1)
    else:
        values = self.feed_forward_out(X)
        return values

def predict_probabilities(self, X):
    if self.n_categories > 1:
        probabilities = self.feed_forward_out(X)
        return probabilities

def train(self):
    data_indices = np.arange(self.n_inputs)

    for i in range(self.epochs):
        for j in range(self.iterations):

            chosen_datapoints = np.random.choice(
                data_indices, size=self.batch_size, replace=False)

            self.X = self.X_full[chosen_datapoints]
            self.y = self.y_full[chosen_datapoints]

            self.feed_forward()
            self.backpropagation()
```

In the code I also added a regularization parameter lmb which has the purpose of constraining the size of the weights, so that they do not arbitrarily large to fit the training data, and in this way overfitting is reduced.

I will measure the size of the weights using the L2-norm, meaning the cost function becomes:

$$C(\Theta) = \frac{1}{N}\sum_{i=1}^{N} L_i(\Theta) \rightarrow \frac{1}{N}\sum_{i=1}^{N} L_i(\Theta) + \lambda||w||_2^2 = \frac{1}{N}\sum_{i=1}^{N} L_i(\Theta) + \lambda \sum_{ij} w_{ij}^2$$

Moreover, when dealing with a classification problem, the response vectors are transformed into matrices, following the one-hot vector representation. This means that each observation (each row of the matrix) gets as many values (columns) as the number of total classes. All the values will be 0 except for a unique value, which will be 1, and it will be found in the corresponding position of the class that each observation belongs to. This technique is especially used when dealing with multiclass problems, but I am going to implement it also in this project.

2.5 Logistic Regression

Logistic regression deals with classification problems, where the response variable is categorical and not continuous. The most common situation is encountered when the response variable presents only two possible outcomes, normally denoted as a binary or dichotomous outcome. This is the case of the problem presented in this project.

In logistic regression the probability that a data point x_i belongs to a category $y_i = \{0,1\}$ is given by the so-called logit function, which is meant to represent the likelihood for a given event.

Let's define some quantities.

$$odds = \frac{p}{1-p}$$

represents the probability of success over the probability of failure,

$$logit = log(odds) = log(\frac{p}{1-p})$$

which is the logarithm of the odds,

$$expit = \frac{e^{z_i}}{1+e^{z_i}}$$

which is the inverse of the logit and represents the probability of success.

From now on let's assume that the classification problem has two classes with y_i either 0 or 1, so the probabilities are:

$$p(y_i = 1|\beta x) = \frac{exp(\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p)}{1 + exp(\beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p)}$$

$$p(y_i = 0|\beta x) = 1 - p(y_i = 1|\beta x)$$

I will use the *Maximum Likelihood Estimation (MLE) principle* in order to define the total likelihood for all possible outcomes from a dataset $D = \{x_i, y_i\}$ with the binary outcome $y_i \in \{0,1\}$ and independently drawn data points. The aim is to maximize the probability of sampling the observed data.

Then let's approximate the likelihood in terms of the product of the individual probabilities of a specific outcome y_i, that is:

$$P(D|\beta) = \prod_{i=1}^{n} [\, p(y_i = 1|\beta x)]^{y_i} \, [1 - p(y_i = 1|\beta x))]^{1-y_i}$$

from which it is obtainable the log-likelihood and the cost function.

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

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

The maximum likelihood estimator is defined as the set of parameters that maximize the log-likelihood with respect to β. Since the cost function is just the negative log-likelihood, for logistic regression let's define the cross entropy:

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

The cross entropy is a convex function of the weights β and, therefore, any local minimizer is a global minimizer.

Minimizing this cost function with respect to β_j gives:

$$\frac{\partial C(\beta)}{\partial \beta_j} = -\sum_{i=1}^{n} (y_j x_j - x_j \frac{exp(\beta x)}{1 + exp(\beta x)})$$

which can be rewritten using a more compact form as:

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

In addition, let's define a diagonal matrix W with elements $p(y_i|\beta x)(1 - p(y_i|\beta x))$, so that a new compact expression of the second derivative can be:

$$\frac{\delta^2 C(\beta)}{\delta \beta \delta \beta^T} = X^T W X$$

This defines the Hessian matrix.

To solve these equations, the Newton-Raphson's iterative method is normally the method of choice. It requires however that the matrices that define the first and second derivatives can be computed in an efficient way, so there should not be a case of high-dimensionality. The iterative scheme is then given by:

$$\beta_{new} = \beta_{old} - (X^T W X)^{-1} (-X^T(y - p))_{\beta_{old}}$$

The right-hand side is computed with the old values of β.

Using logistic regression without any additional regularization does not always guarantee a good result because of over-fitting. This happens especially when the number of observations of the training set is not large enough, compared to the number of features. In order to get a better classifier, a regularization parameter needs to be added. In this project the parameters of logistic regression have been penalized through λ in such way:

$$\beta \leftarrow \beta - \eta \left(g + \frac{\lambda\beta}{n}\right)$$

In Python I created the class object:

```
class RegularizedLogisticRegression:
    def __init__(self, eta, num_iter, fit_intercept, reg, verbose=False):
        self.eta = eta
        self.num_iter = num_iter
        self.fit_intercept = fit_intercept
        self.reg = reg

    def __add_intercept(self, X):
        intercept = np.ones((X.shape[0], 1))
        return np.concatenate((intercept, X), axis=1)

    def fit(self, X, y):
        if self.fit_intercept:
            X = self.__add_intercept(X)

        self.theta = np.zeros(X.shape[1])

        for i in range(self.num_iter):
            z = np.dot(X, self.theta)
            h = sigmoid(z)
            gradient = np.dot(X.T, (h.ravel() - y.ravel())) / y.size
            self.theta -= self.eta * (gradient + self.reg * self.theta/y.size)


    def predict_prob(self, X):
        if self.fit_intercept:
            X = self.__add_intercept(X)

        return sigmoid(np.dot(X, self.theta))

    def predict(self, X):
        return self.predict_prob(X) > 0.5
```

3. Analysis

3.1 Stochastic Gradient Descent

The first part of the project aims at analyzing the prediction capacity of the Ordinary Least Square regression and that of the Stochastic Gradient Descent. The SGD process has been repeated five times, each time minimizing the *MSE (Mean Squared Error),* in function of different parameters. The *MSE* calculates the differences between the observed values and the predicted values raised to the power of two. This eliminates any problems with the derivatives and makes the calculations easier.

Initially, the SGD algorithm has been run with two different learning rate decay schedules, Time-Based and Drop-Based, and the best method resulted to be the Drop-Based, with a hyperparameter of decay $d = 0.3$.

	MSE	Learning Rate	Epochs	Mini-Batches Size	Decay	Alpha	Model
0	0.07513	-	-	-	-	-	OLS Reg
1	0.07608	0.01	100	8	Time-Based	-	SGD
2	0.07589	0.01	100	8	Drop-Based	-	SGD
3	0.07747	0.01	100	16	Drop-Based	-	SGD
4	0.07587	0.01	400	8	Drop-Based	-	SGD

Table 4

Furthermore, I proceeded with a grid search for the values of the learning rates, number of mini-batches and number of epochs. The results show that the best SGD model is the one having $\eta = 0.01$, 400 epochs and a mini-batch size of 8 observations. However, the classic OLS method produces a lower error: 0.0751 versus 0.0759.

It is noticeable that the model with the same hyperparameters, but with only 100 epochs, produces approximately the same error, and if it is rounded to only four digits, it is exactly the same. For computational reasons it could be better to choose the model with less iterations as the best, since the test error the more complex model produces is not significantly better. Moreover, it is also noticeable that the best learning rate value is always 0.01 in this analysis, which coincides with the default values it is usually assigned to η.

It is clear, that if a momentum parameter α is added, the error increases. Momentum, in fact, is widely used when the second derivative is more influent than the gradient and overshadows it. In this scenario, however, there are not ravine zones, so the momentum does not increase the prediction capacity of the model.

	MSE	Learning Rate	Epochs	Mini-Batches Size	Decay	Alpha	Model
0	0.07513	-	-	-	-	-	OLS Reg
1	0.07608	0.01	100	8	Time-Based	-	SGD
2	0.07589	0.01	100	8	Drop-Based	-	SGD
3	0.07747	0.01	100	16	Drop-Based	-	SGD
4	0.07587	0.01	400	8	Drop-Based	-	SGD
5	0.07936	0.001	100	8	Momentum	0.1	MSGD
6	0.07768	0.01	100	32	Momentum	0.001	MSGD

Table 5

OLS performs better in this case because the dataset has 7385 observations and the design matrix includes only 20 features, which means that the inverse matrix is easily calculated and there is no need to include a stochastic approach to the model.

Let's focus now on Ridge Regression and Stochastic Gradient Descent. The implemented SGD process is the same as in the previous part with OLS, this time with the addition of the hyperparameter λ.

	MSE	Learning Rate	Epochs	Mini-Batches Size	Decay	Alpha	Lambda	Model
0	0.07323	-	-	-	-	-	0.001	Ridge Reg
1	0.0761	0.01	100	8	Time-Based	-	0.0001	SGD
2	0.07607	0.001	100	8	Drop-Based	-	0.0001	SGD
3	0.07774	0.01	100	32	Drop-Based	-	0.0001	SGD
4	0.07585	0.01	500	8	Drop-Based	-	0.0001	SGD

Table 6

Once again, the simple Ridge Regression appears to be working better than the Stochastic Gradient Descent, with an error of 0.0732 versus 0.0759. Ridge works also better than OLS, as already pointed out in Project 1. The best model among the ones obtained via SGD is given by $\eta = 0.01$, 500 epochs, a mini-batch size of 8 observations and $\lambda = 0.001$. The penalization is very small, and the error is indeed very similar to what is obtained without a penalization parameter.

By adding a momentum parameter α, the test error worsens, and this happens for same reasons it does for models without penalization.

	MSE	Learning Rate	Epochs	Mini-Batches Size	Decay	Alpha	Lambda	Model
0	0.07323	-	-	-	-	-	0.001	Ridge Reg
1	0.0761	0.01	100	8	Time-Based	-	0.0001	SGD
2	0.07607	0.001	100	8	Drop-Based	-	0.0001	SGD
3	0.07774	0.01	100	32	Drop-Based	-	0.0001	SGD
4	0.07585	0.01	500	8	Drop-Based	-	0.0001	SGD
5	0.0794	0.001	100	8	Momentum	0.1	0.0001	MSGD
6	0.07775	0.01	100	32	Momentum	0.001	0.0001	MSGD

Table 7

3.2 Neural Network for Regression

In the second part of the project neural networks are introduced as an alternative to the simple matrix inversion and stochastic gradient descent. The aim is to test if a Multilayer Perceptron Model is able to predict with more accuracy unseen data. I am going to test the model with different parameters and approaches: the number of hidden layers, the number of hidden nodes, a regularization parameter, the activation function in the hidden layers and the initialization of the weights.

	MSE	Learning Rate	Epochs	Mini-Batches Size	Hidden Layers	Hidden Neurons	Regularization	Activation	Model
0	0.06059	0.01	25	8	1	30	0.0001	Sigmoid	NN
1	0.0665	0.001	25	8	2	30	0.0001	Sigmoid	NN
2	0.05963	0.01	25	8	3	30	0.001	Sigmoid	NN
3	0.06128	0.01	25	8	1	40	0.0001	Sigmoid	NN
4	0.06324	0.001	25	8	2	40	0.001	Sigmoid	NN
5	0.05956	0.01	25	8	3	40	0.001	Sigmoid	NN
6	0.06058	0.01	25	8	1	50	0.001	Sigmoid	NN
7	0.06275	0.001	25	8	2	50	0.0001	Sigmoid	NN
8	0.05443	0.01	25	8	3	50	0.001	Sigmoid	NN
9	0.05896	0.001	25	16	3	50	0.01	Sigmoid	NN
10	0.0565	0.001	25	32	3	50	0.0001	Sigmoid	NN
11	0.05831	0.001	25	64	3	50	0.01	Sigmoid	NN

Table 8

The first models are implemented with the sigmoid as activation function, testing on 25 epochs, 1, 2 and 3 hidden layers and 30, 40 and 50 hidden neurons.

The best model, when using the sigmoid activation function is the most complex: 50 hidden nodes, 3 activation layers and mini-batches with 8 observations each. The test produces an error of 0.05443.

For these first models, both the weights and the biases are initialized randomly. The biases have the same purpose as the intercept in a linear equation. It is an additional parameter which is used to adjust the output along with the weighted sum of the inputs to the neuron. If there was no bias, the model would train only passing through the origin, which is often not in accordance with a real-world scenario. Also with the introduction of biases, the model becomes more flexible. It is common to initialize the biases to be zero, but sometimes (especially with the ReLU activation function) it could be preferable to use a small constant value such as 0.01 for all biases, because this ensures that all neurons produce an output which can be back-propagated in the first training cycle. This second option is what is being used in these first models. The weights are initialized randomly from a normal distribution, and later in the project I will also test models with weighs initialized to zero.

Since this part of the project deals with a regression problem, the outputs are continuous numbers, that do not have to be led back to a probability, as it happens for classification. Therefore, the chosen activation function is the linear function, which leaves the values z_j^L as they are, which leads to $z_j^L = a_j^L$.

3.2.1 Activation Functions

There is not a default activation function that works for all the cases, but each dataset has to be tested using various models, which also differ from the choice of activation function in the hidden layers. I used both ReLU and Leaky-ReLU, keeping the same weight and bias initialization as in the previous models.

When these models only have 1 hidden layer, then they are able to predict fairly the test data, with similar performances to the models using sigmoid, but as soon as the complexity increases, so does the test error. With 1 hidden layer the MSE for the model with ReLU is 0.060 and 0.067 for the one with the Leaky-ReLU, but with 2 hidden layers the MSE are 0.18 and 0.15. Moreover, when using the Leaky-ReLU, the model with 3 hidden layers does not produce any output at all. Even though Leaky-ReLU usually works better than ReLU, this is an example of how there are not any rules in machine learning and each dataset and choice of parameters lead to a different optimal situation.

	MSE	Learning Rate	Epochs	Mini-Batches Size	Hidden Layers	Hidden Neurons	Regularization	Activation	Model
0	0.06059	0.01	25	8	1	30	0.0001	Sigmoid	NN
1	0.0665	0.001	25	8	2	30	0.0001	Sigmoid	NN
2	0.05963	0.01	25	8	3	30	0.001	Sigmoid	NN
3	0.06128	0.01	25	8	1	40	0.0001	Sigmoid	NN
4	0.06324	0.001	25	8	2	40	0.001	Sigmoid	NN
5	0.05956	0.01	25	8	3	40	0.001	Sigmoid	NN
6	0.06058	0.01	25	8	1	50	0.001	Sigmoid	NN
7	0.06275	0.001	25	8	2	50	0.0001	Sigmoid	NN
8	0.05443	0.01	25	8	3	50	0.001	Sigmoid	NN
9	0.05896	0.001	25	16	3	50	0.01	Sigmoid	NN
10	0.0565	0.001	25	32	3	50	0.0001	Sigmoid	NN
11	0.05831	0.001	25	64	3	50	0.01	Sigmoid	NN
12	0.06036	0.001	25	8	1	50	0.001	ReLU	NN
13	0.17562	0.00001	25	8	2	50	1.0	ReLU	NN
14	1.01841	0.01	25	8	3	50	0.1	ReLU	NN
15	0.06684	0.001	25	8	1	50	0.01	Leaky-ReLU	NN
16	0.15483	0.00001	25	8	2	50	1.0	Leaky-ReLU	NN

Table 9

3.2.2 Initialization of weights

As already mentioned, the weights and bias in the MLP model have to be initially set to given values, which are updated through the back-propagation algorithm until they reach the best values in order to be able to make the best predictions on the test set. If the algorithm goes through enough iterations, the output should not change much based on the initialized weights, sometimes, however, there could be some differences. In this part of the project I initialize both weights and biases to zero, in order to test the prediction ability of the neural network.

As it is noticeable from the table, the errors when initializing the weights and bias at zero, are way higher than the ones produced by model with randomly initialized weights. This may not always be the case, but for this specific dataset and choice of design matrix, the model probably needs more iterations to reach the optimal values for the parameters when the weights are initialized at zero.

	MSE	Learning Rate	Epochs	Mini-Batches Size	Hidden Layers	Hidden Neurons	Regularization	Activation	Model
0	0.06059	0.01	25	8	1	30	0.0001	Sigmoid	NN
1	0.0665	0.001	25	8	2	30	0.0001	Sigmoid	NN
2	0.05963	0.01	25	8	3	30	0.001	Sigmoid	NN
3	0.06128	0.01	25	8	1	40	0.0001	Sigmoid	NN
4	0.06324	0.001	25	8	2	40	0.001	Sigmoid	NN
5	0.05956	0.01	25	8	3	40	0.001	Sigmoid	NN
6	0.06058	0.01	25	8	1	50	0.001	Sigmoid	NN
7	0.06275	0.001	25	8	2	50	0.0001	Sigmoid	NN
8	0.05443	0.01	25	8	3	50	0.001	Sigmoid	NN
9	0.05896	0.001	25	16	3	50	0.01	Sigmoid	NN
10	0.0565	0.001	25	32	3	50	0.0001	Sigmoid	NN
11	0.05831	0.001	25	64	3	50	0.01	Sigmoid	NN
12	0.06036	0.001	25	8	1	50	0.001	ReLU	NN
13	0.17562	0.00001	25	8	2	50	1.0	ReLU	NN
14	1.01841	0.01	25	8	3	50	0.1	ReLU	NN
15	0.06684	0.001	25	8	1	50	0.01	Leaky-ReLU	NN
16	0.15483	0.00001	25	8	2	50	1.0	Leaky-ReLU	NN
17	0.08051	0.01	25	8	1	50	0.0001	Sigmoid	NN (w=0)
18	0.08204	0.01	25	8	2	50	0.001	Sigmoid	NN (w=0)
19	0.10076	0.1	25	8	3	50	0.0001	Sigmoid	NN (w=0)
20	1.01819	0.001	25	8	1	50	1.0	ReLU	NN (w=0)
21	1.01819	0.0001	25	8	2	50	0.01	ReLU	NN (w=0)
22	1.01819	0.0001	25	8	3	50	0.001	ReLU	NN (w=0)
23	1.01819	0.001	25	8	1	50	1.0	Leaky-ReLU	NN (w=0)
24	1.01819	0.0001	25	8	2	50	0.01	Leaky-ReLU	NN (w=0)
25	1.01819	0.0001	25	8	3	50	0.001	Leaky-ReLU	NN (w=0)

Table 10

Although, it is visible that in this case, also the model with 3 hidden layers and Leaky-ReLU as activation function provides an output, even if the test error is high.

3.3 Neural Network for Classification

For the second half of the project I am going to focus on a Classification problem and I will therefore perform the analysis on the *Wisconsin Breast Cancer Data*. I will still focus on the prediction ability of the model to choose if in this specific case a MLP works better than Logistic Regression.

Since the response variable is binary, I will use the *Accuracy* to quantify the goodness of a model, which calculates the percentage of correctly classified observations, and the goal is to maximize it.

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

Moreover, in order for the model to be able to classify the observations into two categories, the final output of the MLP has be a probability, which will be approximated to 0 if it is ≤ 0.5 and to 1 if it is > 0.5. To obtain a probability as last output a_j^L, the chosen activation function is the softmax.

	Accuracy	Learning Rate	Ripetitions	Mini-Batches Size	Hidden Layers	Hidden Neurons	Regularization	Activation	Model
0	0.69298	0.1	25 epochs	8	1	30	0.0001	Sigmoid	NN
1	0.69298	0.01	25 epochs	8	2	30	0.001	Sigmoid	NN
2	0.64035	0.01	25 epochs	8	3	30	0.01	Sigmoid	NN
3	0.68421	0.001	25 epochs	8	1	30	0.0001	ReLU	NN
4	0.71053	0.0001	25 epochs	8	2	30	0.001	ReLU	NN
5	0.71053	0.0001	25 epochs	8	3	30	0.01	ReLU	NN
6	0.68421	0.001	25 epochs	8	1	30	0.0001	Leaky-ReLU	NN
7	0.7193	0.00001	25 epochs	8	2	30	0.001	Leaky-ReLU	NN
8	0.69298	0.00001	25 epochs	8	3	30	0.01	Leaky-ReLU	NN
9	0.64035	0.01	25 epochs	8	1	50	0.0001	Sigmoid	NN
10	0.67544	0.001	25 epochs	8	2	50	0.0001	Sigmoid	NN
11	0.69298	0.01	25 epochs	8	3	50	0.001	Sigmoid	NN
12	0.69298	0.001	25 epochs	8	1	50	0.01	ReLU	NN
13	0.69298	0.001	25 epochs	8	2	50	0.0001	ReLU	NN
14	0.72807	0.00001	25 epochs	8	3	50	0.01	ReLU	NN
15	0.69298	0.001	25 epochs	8	1	50	0.01	Leaky-ReLU	NN
16	0.70175	0.00001	25 epochs	8	2	50	1.0	Leaky-ReLU	NN
17	0.71053	0.00001	25 epochs	8	3	50	0.01	Leaky-ReLU	NN

Table 11

For this classification problem, the ReLU and Leaky-ReLU work best than the sigmoid function, which turned out to be the best option for regression. However, as already mentioned, this cannot be considered a ready-made rule. The best model among the ones trained is the most complex model with ReLU as activation function and has an accuracy of 72,81%. The best learning rate is really small (10^{-5}), which means that the gradient descent moved really slowly towards the minimum. All the models with the highest accuracy have very small learning rates. The regularization parameter, on the other hand, is among the highest, which means that the weights are more penalized and converge faster to the minimum.

3.4 Logistic Regression

For the Logistic Regression the chosen cost function is the *Cross-Entropy*, which measures the difference between two probability distributions for a given random variable.

$$CrossEntropy = -\sum_{i=1}^{n} (y_i(\beta x) - log(1 + exp(\beta x)))$$

Cross-Entropy loss measures the performance of a classification model whose output is a probability value between 0 and 1. Moreover, it increases as the predicted probability diverges from the actual label. A perfect model would have a loss of 0. However, I will still use the *Accuracy Score* to test the prediction ability of the models, so that they can be compared to the ones of the MLPs.

It turns out that in this case the addition of a regularization parameter does not change the results. Moreover, there is no difference between 50.000 iterations and 100.000, so the best model will be the easiest to compute, which means the one with no regularization parameter and 50.000 iterations. However, it is often proved that a penalization parameter actually improves the goodness of prediction.

	Accuracy	Learning Rate	Ripetitions	Regularization	Model
0	0.63158	0.1	10.000 iter	0.0001	Logistic
1	0.69298	0.1	50.000 iter	0.0001	Logistic
2	0.69298	0.1	100.000 iter	0.0001	Logistic
3	0.63158	0.1	10.000 iter	0	Logistic
4	0.69298	0.1	50.000 iter	0	Logistic
5	0.69298	0.1	100.000 iter	0	Logistic

Table 12

4. Conclusions

4.1 Regression

The best model to make predictions on the *Car CO_2 Emissions* is the MLP with:

- Test Error: 0.05443
- Learning rate: 0.01
- Epochs: 25
- Minibatch size: 8
- Hidden layers: 3
- Hidden neurons: 50
- Regularization parameter: 0.001
- Activation function: Sigmoid

It is the most complex MLP among the ones trained, but it does not overfit, on the contrary, it produces the lowest test error. In this case the SGD (0.07585) works worse than the simple OLS (0.07513) and Ridge (0.07323), because of the dimensionality of the data. It is not a high-dimensionality problem, so the matrix inversion is not computationally too expensive and it is more accurate than introducing a stochastic element.

It is important to deeply understand that each dataset and choice of parameters lead to a different situation and optimal solution. This conclusion does not mean that Neural Networks *always* work better than OLS, Ridge and Gradient Descent, even though based on empirical evidence, the researchers affirm that MLPs tend to work better in a lot of situations when the aim of the study is predictions. In fact, there are occasions in which Neural Network are not the best options, for example when the purpose is to describe the data or interpret the results.

From the plot is evident the better fit of the predictions obtained with the MLP compared to the one obtained with Ridge Regression.

```
Mean Squared Error Ridge: 0.07323
Mean Squared Error NN: 0.05443
```

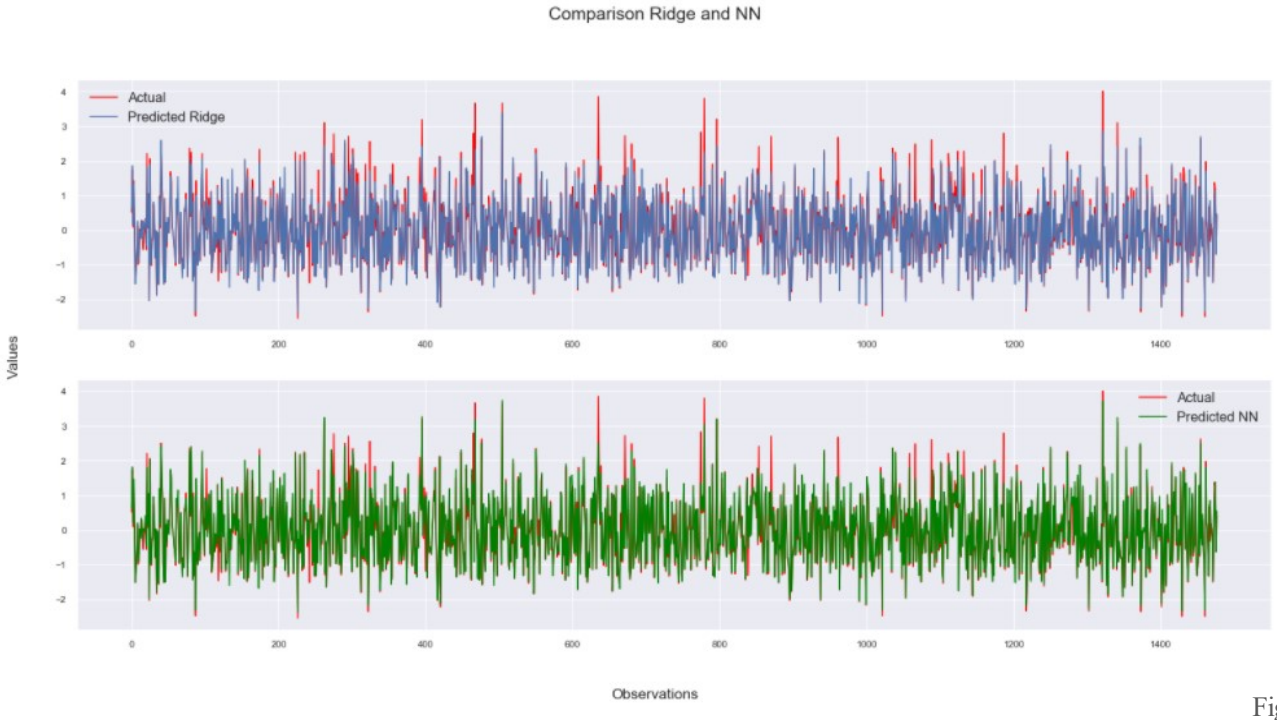

Figure 8

4.2 Classification

Also for classification the best model turned out to be a MLP. The model that best predicts the *Wisconsin Breast Cancer Data* has:

- Test Accuracy Score: 72.81%
- Learning rate: 10^{-5}
- Epochs: 25
- Minibatch size: 8
- Hidden layers: 3
- Hidden neurons: 50
- Regularization parameter: 0.01
- Activation function: Sigmoid

It is also in this case the most complex MLP among the ones trained. However, there were also other easier models that produced a high accuracy score, so based on the computational work that the researcher is willing to put up with, it can also be chosen a simpler model. For example, with just 2 hidden layers and 30 hidden nodes the accuracy is 71.05%. In this case the Logistic Regression (69.30%) works worse than almost all the trained MLPs, but the difference is not substantial, so it can be said that in this specific case the classification could also be performed via Logistic Regression.

Figure 9

.

From these plots is evident the better fit of the predictions obtained with the neural network compared to the one obtained with the logistic regression, which were able to capture more information about the classification when predicting.

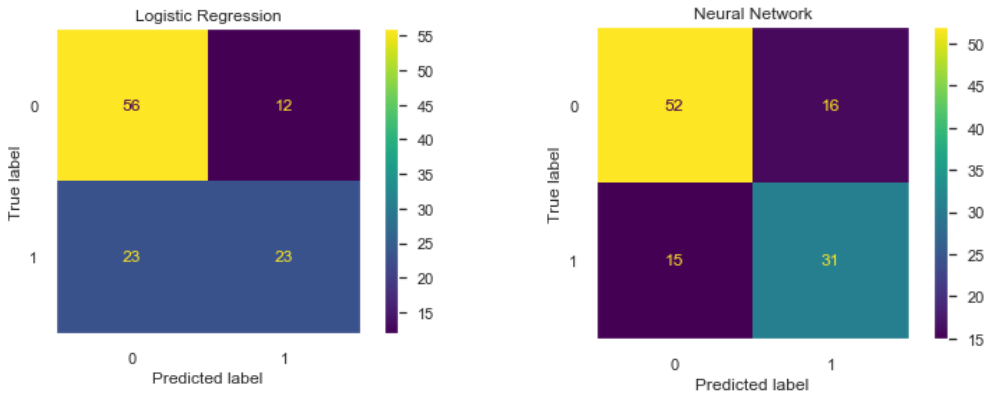

Figure 10

This result is also noticeable from the confusion matrices of the two models. The logistic model is able to classify more observations as true negatives, but less as true positive. Since this data is about predicting whether a woman has breast cancer or not, the aim is to have the least number of false negatives. These are the cases when the woman does not undergo a therapy because considered healthy, when in reality she has cancer. Based on the models trained in this project, MLP managed to have a smaller number of observations classified as false negatives.

It is not different for classification problems: this conclusion does not mean that Neural Networks *always* work better than Logistic Regression. It is the researcher's job to evaluate the main goals of the study and choose the best model accordingly.

References

[1] Goodfellow I, Bengio J, Courville A. *Deep Learning*. Cambridge (MA): MIT Press. 2016. Chapters 5-6.

[2] Hjorth-Jensen M. *Overview of course material: Data Analysis and Machine Learning*. Week 39, Week 40, Week 41. https://compphysics.github.io/MachineLearning/doc/web/course.html. Accessed: 2021-11-05.

[3] Donges N. *Gradient Descent: An Introduction to 1 of Machine Learning's Most Popular Algorithms*. https://builtin.com/data-science/gradient-descent. 2021. Accessed: 2021-11-09.

[4] Sharma S. *Activation Functions in Neural Networks*. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. 2017. Accessed: 2021-11-09.

[5] Wolberg W.H. *UCI Machine Learning Repository: Breast Cancer Wisconsin (Original) Dataset*. https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original). Accessed: 2021-11-01.

More discussions and codes can be found at my GitHub repository: https://github.com/isarositi/FYS-STK4155/blob/main/Project%202_Rositi.ipynb